KILO TM: HARDWARE TRANSACTIONAL MEMORY FOR GPU ARCHITECTURES

PROGRAMMING GPUS IS CHALLENGING FOR APPLICATIONS WITH IRREGULAR FINE-GRAINED COMMUNICATION BETWEEN THREADS. TO IMPROVE GPUS' PROGRAMMABILITY AND THUS EXTEND THEIR USAGE TO A WIDER RANGE OF APPLICATIONS, THE AUTHORS PROPOSE TO ENABLE TRANSACTIONAL MEMORY (TM) ON GPUS VIA KILO TM, A NOVEL HARDWARE TM SYSTEM THAT SCALES TO THOUSANDS OF CONCURRENT TRANSACTIONS.

Wilson W.L. Fung
Inderpreet Singh
University of British Columbia

Andrew Brownsword
Electronic Arts

Tor M. Aamodt
University of British Columbia

Applications that benefit from running on GPUs tend to contain considerable data parallelism coupled with regular memory access patterns that ensure efficient use of off-chip memory bandwidth. An important question for GPU manufacturers is whether the market for GPUs can expand to include a wider range of applications. Increasing the range of applications that can exploit GPUs' higher peak performance and power efficiency requires support for synchronization mechanisms. Writing and debugging code that involves fine-grained communication among thousands of threads is challenging on current hardware because it could require lock-based programming. It is well-known in the supercomputing field that dealing with large problem sizes and a large number of threads causes highly nondeterministic interactions, exacerbating the programming challenge.[1]

In this article, we explore how to support a transactional memory (TM) programming model on a GPU to ease the challenge of writing code that requires synchronization between thousands of threads.[2,3] TM simplifies software development for parallel architectures by providing the programmer with the illusion that code blocks, called transactions, execute atomically. For example, with TM, the programmer does not need to write code with locks to ensure mutual exclusion. The underlying TM system optimistically executes transactions in parallel to improve performance. If data accesses from any two transactions cause data races (known as *conflicts* in TM), the system will restart one of the transactions to ensure that each transaction's execution appears atomic in a global serial order.

Faster return on investment

In the context of GPUs, we believe TM will encourage programmers to explore more efficient algorithms that require synchronization. The early development stage of a GPU application requiring complex fine-grained synchronization is likely to be dominated by debugging the application's functionality. During this phase (denoted by "?" in Figure 1), uncertainty about the

0272-1732/12/$31.00 © 2012 IEEE Published by the IEEE Computer Society

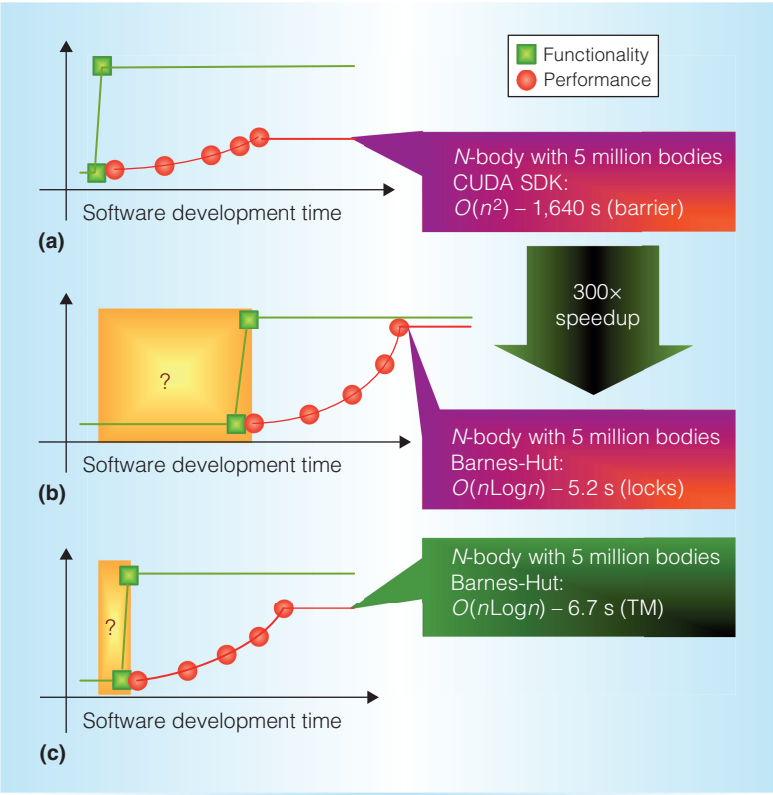

Figure 1. Illustration of software development time for GPU applications: coarse-grained or no synchronization (a), fine-grained locking (b), and transactional memory (TM) (c). We use a CUDA implementation of the Barnes-Hut algorithm by Burtscher and Pingali to demonstrate the performance benefit with fine-grained synchronization.[4] We estimate the TM execution time by scaling the iterative tree-building kernel's measured execution time by 2.74× (0.87 seconds → 2.38 seconds). This kernel is the part of Barnes-Hut algorithm that requires fine-grained locking. In our evaluation, this kernel runs 2.74× slower with Kilo TM than with fine-grained locking. (SDK: software development kit.)

application's viability and performance could jeopardize a software project. With TM support on the GPU, the programmer can create correctly working code sooner, reducing the overall software development risk. More development time can then be devoted to optimizing performance instead of debugging. Although TM might pose a performance overhead, we believe that it could help improve application performance in time-constrained development.

GPU architecture

Figure 2 shows the high-level organization of our GPU architecture. It consists of a collection of single-instruction, multiple-thread (SIMT) cores connected by an interconnection network to multiple memory partitions. Each SIMT core is heavily multithreaded and contains a private, noncoherent Level-1 (L1) data cache for access to global memory space and thread-private local memory space. The SIMT cores access a distributed, shared, read/write last-level (L2) cache and off-chip DRAM via the on-chip network.

A CUDA program starts on a CPU and then launches computing kernels onto a GPU.[5] Each kernel launch dispatches a hierarchy of threads (a grid of blocks of scalar-threads running the same computing kernel) onto the GPU. Blocks are allocated as a single work unit to a SIMT core. Threads within a block can communicate through an on-chip, per-core shared memory. The SIMT execution model manages scalar threads as a single-instruction, multiple-data (SIMD) execution group called a *warp* (or *wavefront* in AMD terminology). Each warp contains 32 scalar threads. Different warps can execute different paths concurrently. Each warp has a SIMT stack that serializes the execution of different thread subsets that diverge to alternate control flow paths.[6]

TM performance potential

Figure 3 compares the performance of a set of GPU TM applications running on an ideal GPU TM system against the applications' fine-grained lock versions.[7] (Table 1 lists these applications.) The ideal TM system has no overheads for detecting conflicting transactions. We normalize the performance to that obtained by serializing all transaction executions via one global lock. On average, the applications running on ideal TM achieve 279× speedup over serializing all transactions, and are 24 percent faster than their fine-grained locking counterparts. Our study's goal was to get as close to this ideal as possible.

Kilo TM: Reconciling TM and GPU goals

Many hardware TM (HTM) proposals are designed around multicore CPUs with tens of cores, each running a single thread.[8,9] In contrast, GPUs are designed to exploit fine-grained data parallelism via thousands of concurrent threads. This difference in

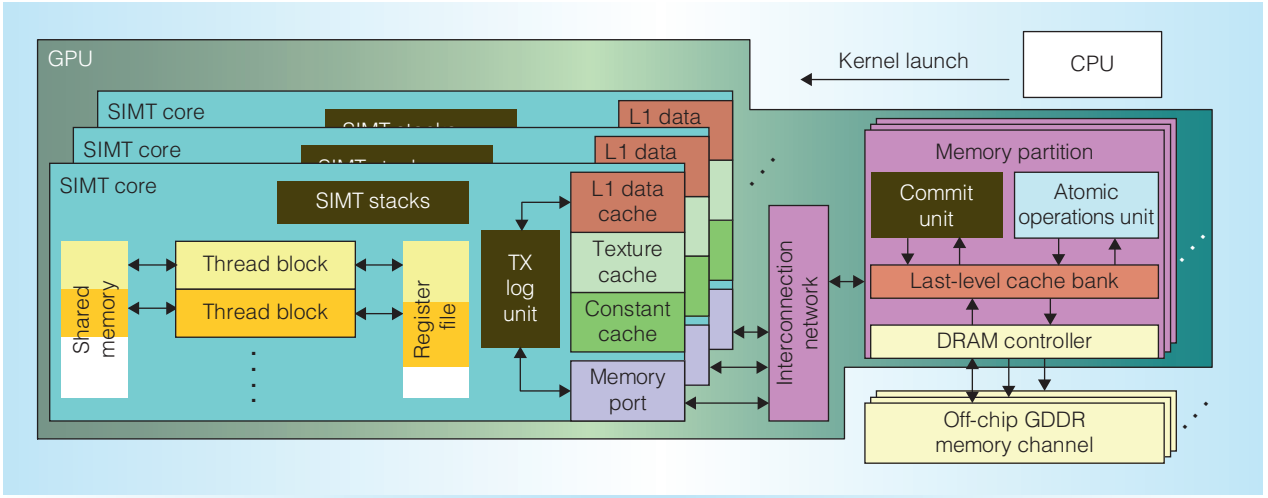

Figure 2. High-level GPU architecture and GPU computing programming model. Kilo TM adds a transaction (TX) log unit and a commit unit, and requires minor modification to the single-instruction, multiple-thread (SIMT) stack hardware. (GDDR: graphics double data rate; L1: Level 1.)

design focus raises many challenges in adopting existing HTM proposals for GPUs.

To address these challenges, we designed Kilo TM, a novel TM system scalable to thousands of concurrent transactions. Kilo TM features lazy conflict detection and lazy version management,[3] and it supports unbounded transactions. These transactions are weakly isolated,[3] and they don't perform irreversible operations such as system calls and I/O operations. Each transaction has a single entry and a single exit, matching `atomic{}` semantics in common TM language extensions.[3] Transaction boundaries are conveyed to hardware with `tx_begin` and `tx_commit` instructions in the computing kernel. Nested transactions are flattened into a single transaction.[3]

Figure 2 highlights the changes required to implement Kilo TM on our baseline GPU architecture.

SIMT stack extension

The active mask of the top-of-stack (TOS) entry represents the subset of threads in the warp that executes the current path, as indicated by the program counter (PC). When a warp finishes a transaction, each of its active threads will try to commit. Some threads might abort and need to re-execute their transactions because of conflicts, whereas other threads might succeed in

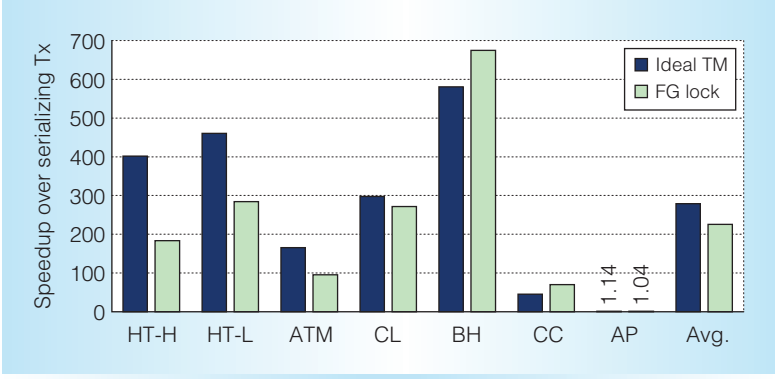

Figure 3. Performance comparison between applications running on an ideal TM system and their respective fine-grained (FG) locking version. The massive speedups for both cases illustrate the performance advantage of fine-grained synchronizations for GPU applications. (Details of these applications are available in our full paper.[7]) (Tx: transaction.)

Table 1. GPU transactional memory (TM) applications for our performance evaluation.

Application	Description
AP	Association rule mining
ATM	Bank account transactions simulation
BH	Barnes-Hut algorithm for *N*-body simulation
CC	Graph cuts for image segmentation
CL	Cloth physics simulation
HT-H	Hash table construction (high contention)
HT-L	Hash table construction (low contention)

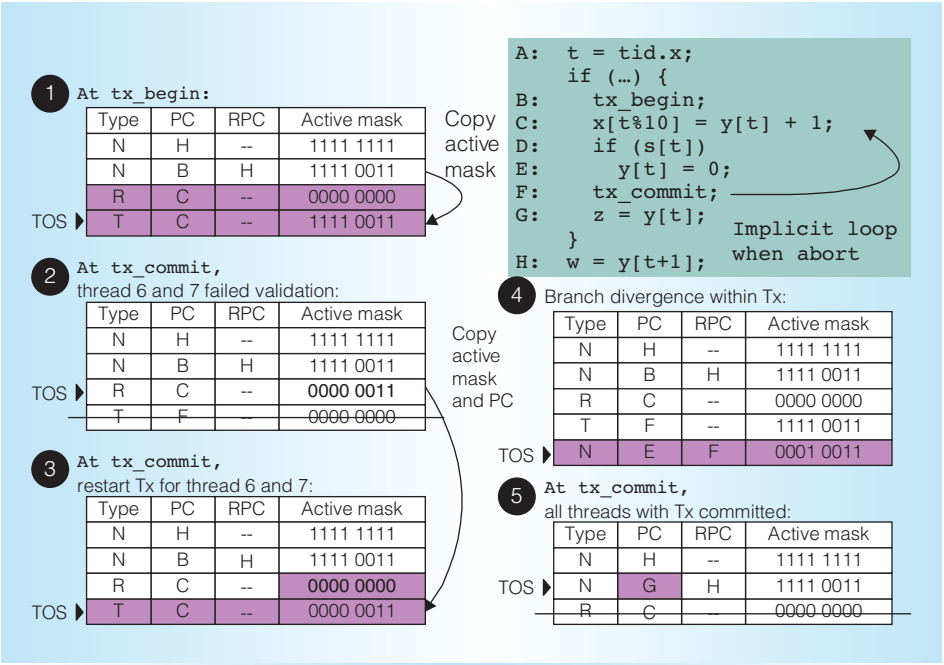

Figure 4. SIMT stack handling divergence of a warp due to transaction aborts (conflicts detected). Threads 6 and 7 have aborted and are restarted. The stack entry types are normal (N), transaction retry (R), and transaction top (T). B, C, E, F, H are program counters referring to different lines in the example program in the figure. For each scenario, added entries or modified fields are shaded. The active mask of the top-of-stack (TOS) entry represents the subset of threads in this warp that executes the current path indicated by the program counter (PC). The TOS entry is popped when PC = RPC or at `tx_commit`. (RPC: reconvergence program counter.)

committing their transactions. Because this outcome might not be unanimous across the entire warp, a warp could diverge afterwards.

Figure 4 shows how to extend the SIMT stack 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 (scenario 1). The first entry of type R (transaction retry) 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 (transaction top) tracks the current transaction attempt. The R entry handles transaction aborts triggered by a watchdog timer required to satisfy opacity.[10] At `tx_commit` (line F), any thread that is aborted sets its mask bit in the R entry. The T entry is popped when the warp finishes the commit process (that is, its active threads have either committed or aborted) (scenario 2). The warp then pushes a new T entry onto the stack using the active mask and PC from the R entry to restart the aborted threads. Then, the active mask in the R entry is cleared (scenario 3). Branch divergence of a warp within a transaction is handled in the same way as nontransactional divergence (scenario 4). If the active mask in the R entry is empty, both the T and R entries are popped, revealing the original N (normal) entry (scenario 5). Its PC is then set to the instruction right after `tx_commit`, and the warp resumes normal execution.

Software register checkpoint

To execute thousands of threads concurrently, GPUs spend significant hardware resources on register storage. For example, each Nvidia Fermi GPU contains 2 Mbytes of registers.[5] Naively checkpointing all registers used by a thread at transaction boundaries,

as proposed in many HTM publications, is too expensive.

Kilo TM uses software for version management of registers and local memory space. We observed in our applications that the original values in many registers are rarely used when a transaction restarts, so restoration is unnecessary. A compiler could determine which registers are both read and written within a transaction and insert code to checkpoint and restore them before and after a transaction. The worst of our benchmarks requires restoring two registers per transaction on average, whereas other benchmarks require none.

Linear transaction logs

Many other HTM proposals leverage cache coherence for conflict detection, while assuming each transaction owns a private L1 cache. Although recent GPUs have caches,[5] the local caches at each SIMT core aren't coherent, and there are fewer cache lines than scalar threads (more than 1,000) sharing the L1 data cache on each SIMT core.

Kilo TM manages transactional global-memory accesses in hardware. These accesses skip the noncoherent L1 data cache. Each transaction buffers its saved read-set values and memory writes in a read log and a write log (in address value pairs) in local memory, located in off-chip DRAM and cached in the per-core L1 data cache. This lets Kilo TM support unbounded transactions.[7] Each transaction can use a small bloom filter to detect whether it is reading from its write set. A hit in the filter triggers a search through the write log.

Value-based conflict detection

Detecting conflicts between thousands of concurrent transactions efficiently is challenging. Naive broadcast-based detection scales poorly. Many proposed TMs use global metadata, such as a cache coherence directory, to eliminate unnecessary traffic. GPUs such as Fermi do not have a coherent, private cache for each thread.[5] Signature-based HTMs can operate independently of caches.[9] We experimented with an ideal version of a signature-based HTM and found that storing a signature for each thread requires 3.8 Mbytes of total storage to achieve a reasonably low false conflict rate.

Typical conflict detection used in HTMs checks the existence of conflicts and identifies the specific conflicting transactions. Many software TMs, such as RingSTM,[11] detect only the existence of conflicts between a committing transaction and transactions that have already committed. Kilo TM uses value-based conflict detection to exploit this insight.[12] It detects conflicts without using any global metadata or cache coherence protocol; only values from global memory are used.

Each transaction stores the value of each global memory read in its read log (in address-value pairs) during execution. Upon its completion, the transaction performs validation by comparing the saved values of its read set against the latest values in memory. A changed value indicates a conflict with one or more committed transactions. Transactions with detected conflicts can self-abort without interfering with other running transactions (shown in Figure 5a). Unlike atomic compare-and-swap (CAS) operations used in nonblocking algorithms, value-based conflict detection can tolerate the ABA problem (see the "Correctness Discussion" sidebar).[13]

Each transaction normally validates only once before it commits. A transaction is *doomed* if it has observed an inconsistent view of memory (for example, if between two memory reads, another transaction has committed and updated the accessed locations). These doomed transactions could enter an infinite loop. To ensure that doomed transactions are eventually aborted, we use a watchdog timer to trigger a validation. This satisfies opacity with minimum overhead for GPUs.[10]

Distributed validation-commit pipeline

Despite its merits, using value-based conflict detection in Kilo TM leads to more challenges, because a naive implementation serializes transaction commits. This serialization can seem unavoidable, because a transaction's memory updates (its write set) are invisible to others until the transaction commits. Two conflicting transactions validating concurrently observe no changes to their read sets, and will subsequently update memory

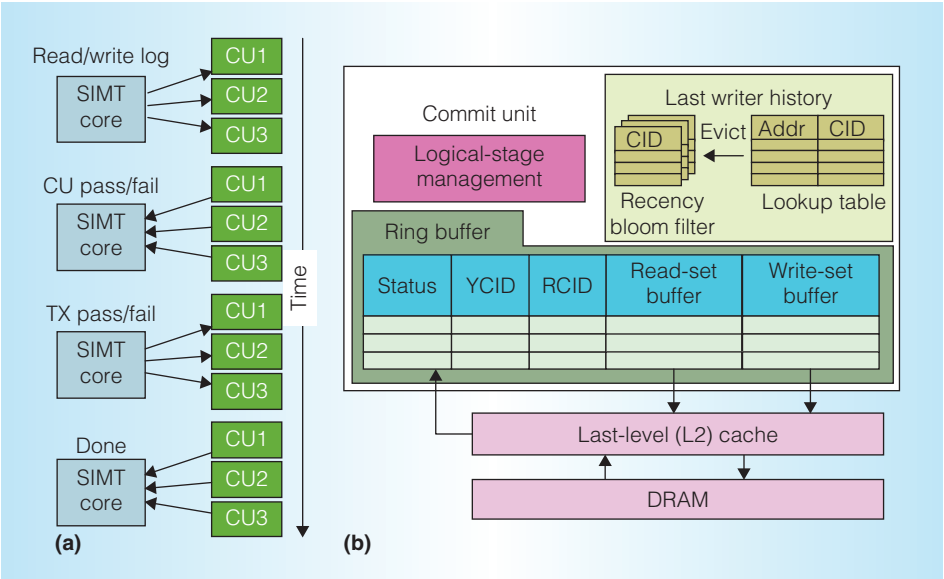

Figure 5. Commit units (CUs) in Kilo TM: communication flow with the transaction at the SIMT core (a), and overview of a commit unit (b). A committing transaction in Kilo TM communicates exclusively with the commit units and does not interfere with other running transactions. (CID: commit ID; RCID: retired CID; YCID: youngest CID.)

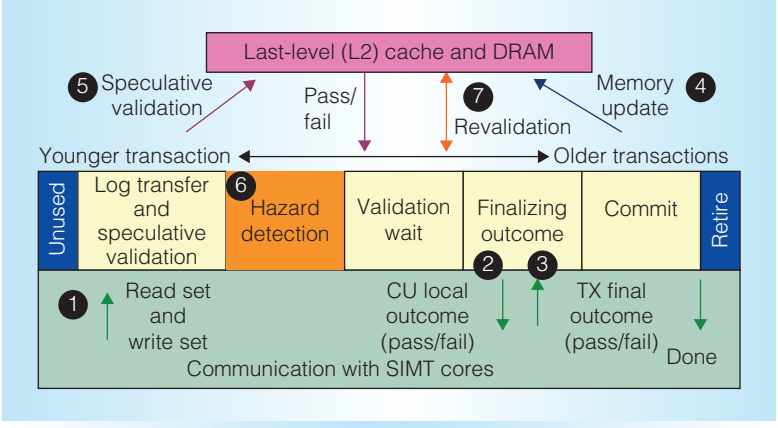

Figure 6. Logical-stage organization and communication traffic of the bounded ring buffer stored inside each commit unit. This organization lets the commit units support thousands of concurrently committing transactions while keeping the design simple.

with their contradicting write sets. Although serializing all commits prevents this potential data race, it also prevents nonconflicting transactions from committing in parallel.

Kilo TM first increases commit parallelism through a set of commit units (shown in Figure 5b). Each memory partition has a commit unit that handles validations and commits of transactional accesses to that partition. Before a transaction starts its validation-commit process, it acquires a commit ID (CID) from a central ID vendor (similar to Scalable TCC [Transactional Memory Coherence and Consistency][8]). The CID ensures that the conflict resolution is unanimous among all commit units, and it associates the transaction's state to a commit entry in each commit unit. Each commit unit has a ring buffer of commit entries,[11] organized into the logical stages depicted in Figure 6. When a transaction finishes executing, it sends its read log and write log to the commit units (step 1) for conflict detection (validation). Each unit replies with the outcome (pass/fail) back to the transaction at the core (step 2). If all commit units report no conflict detected, the transaction permits the commit units to publish the write log to memory (steps 3 and 4).

To further improve commit parallelism for nonconflicting transactions, each commit unit speculatively validates a subset of transactions in parallel through value-based conflict detection (step 5). This validation is speculative because it detects conflicts only with the committed transactions, and the transactions being validated could have conflicts with one another. To detect potential

Correctness Discussion

A general concern for the correctness of value-based conflict detection, which is used in Kilo TM, is the possibility of subtle bugs due to the ABA problem. We show that value-based conflict detection can tolerate the ABA problem, and like other software transactional memories (TMs) such as Norec (No Ownership Records),[1] we can create a logical order for Kilo TM in which each transaction's validation and commit are indivisible. (A full discussion is available elsewhere.[2])

Tolerance to the ABA problem

Researchers have found examples of ABA problems for published nonblocking algorithms. These generally result from an implicit assumption that it is possible to infer atomicity of a high-level operation on a concurrent data structure as long as a guard variable's value is the same after a sequence of low-level instructions for implementing the operation. This fallacy results in subtle bugs.[3]

We argue that a transactional memory (TM) system with value-based conflict detection isn't vulnerable to the ABA problem as long as it ensures that each transaction's read set has observed a consistent view of memory (that is, conflict detection isn't comparing values with a partially committed transaction).

Consider a TM system with address-based conflict detection (full knowledge of which locations have been modified by other transactions). If any location read by a committing transaction has been changed and restored to its original value since the transaction originally read it, aborting the transaction and rerunning it instantaneously with the updated memory would yield the same result (same addresses and values in its write set). This situation summarizes the behavior of a transaction in Kilo TM that has passed value-based conflict detection, in which other conflicting transactions have changed and then restored the values of the transaction's read set when the transaction is running. Committing the transaction directly without rerunning it effectively serializes it behind the last committed transaction. This requires the transaction's read set to observe a consistent view of memory, and to commit before other transactions update locations in the transaction's write set.

Logical validation-commit indivisibility

In Kilo TM, each transaction T_X is given a unique commit ID prior to validation and commit, and this ID defines the commit order of T_X. T_X will obtain a new commit ID for each execution attempt (that is, a new ID is assigned every time T_X is aborted). Given two transactions, T_X and T_Y, with commit IDs X and Y, where $X < Y$, Kilo TM's implementation guarantees the following partial ordering:

- Validation of each word w by T_Y always happens after any write to w by T_X.
- Any write to w by T_Y always happens after any write to w by T_X.
- Validation of w by T_X always happens before any write to w by T_Y.

These guarantees order validations and writes at each memory location (word) in ascending commit order. With this per-word order, we can show that transactions committed by Kilo TM satisfy conflict serializability.[4] All conflict relations produced from the accesses at each word obey the commit order. Hence, a conflict graph created from these conflict relations is always acyclic.

Conflict serializability implies a logical timeline in which validation and commit of each transaction are indivisible (performed without being interleaved by other transactions). The logical timeline also implies that validation of each transaction always observes a consistent view of memory.

References

1. L. Dalessandro, M.F. Spear, and M.L. Scott, ''NOrec: Streamlining STM by Abolishing Ownership Records,'' *Proc. 15th ACM SIGPLAN Symp. Principles and Practice of Parallel Programming* (PPoPP 10), ACM Press, 2010, pp. 67-78.
2. W.W.L. Fung et al., *KILO TM Correctness: ABA Tolerance and Validation-Commit Indivisibility,* tech. report, Dept. of Electrical Eng., University of British Columbia, to be published in 2012.
3. M.M. Michael, ''Practical Lock-Free and Wait-Free LL/SC/VL Implementations Using 64-Bit CAS,'' *Proc. 18th Int'l Symp. Distributed Computing* (DISC 04), LNCS 3274, Springer, 2004, pp. 144-158.
4. G. Weikum, and G. Vossen, *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery,* Morgan Kaufmann, 2001.

conflicts (hazards) among this limited set of transactions (step 6), the commit units use a combination of an exact address-based filter and a novel bloom filter structure, which we call a *recency bloom filter* (see the last writer history unit in Figure 5b). Kilo TM resolves a hazard by deferring one of the conflicting transactions with a younger CID and revalidating this transaction's read set after the other transaction has updated global memory (step 7). Revalidation serializes the validation-commit process among conflicting transactions. (Details of this design are available in our paper for the 44th Annual IEEE/ACM International Symposium on Microarchitecture.[7])

Concurrency control

Running fewer transactions concurrently can speed up high-contention applications

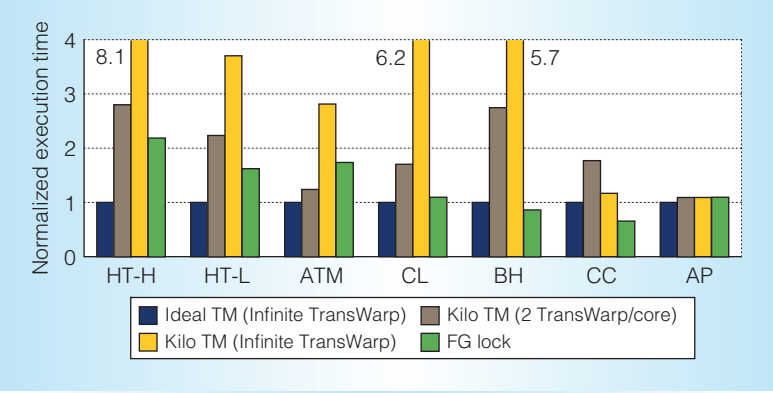

Figure 7. Execution time of our applications with Kilo TM and fine-grained locking (FG lock), normalized to the execution time of ideal TM. Lower is better. (Infinite TransWarp: unlimited transaction concurrency; 2 TransWarp/core: limiting each core to two concurrent transaction warps.)

(with transactions that are likely to abort) and lower 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.

Evaluation

We evaluated Kilo TM's performance using a set of GPU TM applications with a modified version of GPGPU-Sim 3.0.[14] Our modified GPGPU-Sim is configured to model a GPU similar to Nvidia's Quadro FX 5800, extended with L1 data caches and an L2 unified cache similar to Nvidia's Fermi.[5] We validated GPGPU-Sim 3.0 with the FX 5800 configuration, with no cache extensions and using Parallel Thread Execution (PTX, the pseudo-assembly language used in CUDA) instead of SASS (the assembly language executed natively by the hardware GPU), against a real FX 5800. We observed an instruction-per-cycle (IPC) correlation of about 0.93 for a subset of the CUDA software development kit (SDK) benchmarks. GPGPU-Sim incorporates a configurable interconnection network simulator,[15] and it models an out-of-order memory access scheduler with detailed GDDR3 (graphics double data rate 3) timing. Atomic CAS operations on GPGPU-Sim have about 4× higher throughput than on an Nvidia Fermi GPU, making our comparison to fine-grained locks conservative. (Details of our simulation configuration and benchmarks are available in our full paper.[7])

Figure 7 shows the execution time of our applications with Kilo TM and fine-grained locking (FG lock), normalized to the execution time of ideal TM. With unlimited transaction concurrency (infinite TransWarp), Kilo TM is on average 4.1× slower than ideal TM. The HT-H, CL, and BH applications are affected the most. These applications have many concurrent transactions with high contention.

Limiting each core to two concurrent transaction warps (2 TransWarp/core in Figure 7, 1,920 concurrent transactions across the GPU) reduces contention in HT-H, CL, and BH, and improves their performance with Kilo TM by 2× to 3×. The performance of HT-L improves by 66 percent. ATM, a low-contention application that requires acquiring multiple locks to enter a critical section, speeds up by 2.3× because of fewer false hazards detected (fewer unnecessary serializations). Kilo TM outperforms FG lock for this application. The performance of CC drops by 34 percent, but further analysis shows that thread-level concurrency control can remove this penalty. Overall, Kilo TM with transaction concurrency limited to two warps per core captures 52 percent of ideal TM and 59 percent of fine-grained locking performance, and is 128× faster than executing transactions serially on the GPU.

We find that AP, a TM application ported from CPU-optimized versions, performs poorly on GPUs regardless of the data synchronization mechanism used (FG lock or TM). Optimizing applications such as AP for GPUs would involve redesigning the algorithm. TM allows for creating the redesigned algorithm's functional versions far sooner, thus lowering the risk of this investment.

Breaking down Kilo TM's performance by execution time indicates that future refinements could reduce its performance overhead.[7] We estimate that implementing Kilo TM on an Nvidia Fermi GPU would increase chip area by only 0.5 percent.[5]

The growth in using GPUs for nongraphics computation illustrates that programmers are willing to redesign algorithms

to exploit GPUs' high computational efficiency. Enhancing GPU architectures to support robust synchronization mechanisms such as transactional memory will lower the risk of employing GPUs for more complex applications. Some of the insights and innovations gained from this work will likely also apply to multicore CPUs as they scale to support more concurrent threads. Our evaluation with an ideal TM system motivates us to make further improvements to Kilo TM. Future work is also needed to explore performance-tuning techniques for GPU TM applications. Insights gained from these explorations could contribute to a GPU-optimized, intuitive TM interface that doesn't undermine programmers' ability to enhance application performance. MICRO

Acknowledgments

We thank Doug Carmean and Mark Hill for their enthusiastic support of our work. Specifically, we are very grateful for Doug Carmean's insightful feedback on the final version of our article, and Mark Hill's effort in overseeing our discussion to address the correctness concerns with Kilo TM. We also thank Jaewoong Chung, Mike O'Connor, Arrvindh Shriraman, Andrew Boktor, Ali Bakhoda, Henry Wong, and the anonymous reviewers for their valuable comments. This work was partially supported by the Natural Sciences and Engineering Research Council of Canada.

References

1. D. Arnold et al., ''Stack Trace Analysis for Large Scale Debugging,'' *Proc. IEEE Int'l Parallel and Distributed Processing Symp.* (IPDPS 07), IEEE CS, 2007; doi:10.1109/IPDPS.2007.370254.
2. M. Herlihy and J.E.B. Moss, ''Transactional Memory: Architectural Support for Lock-Free Data Structures,'' *Proc. 20th Ann. Int'l Symp. Computer Architecture* (ISCA 93), ACM, 1993, pp. 289-300.
3. T. Harris, J. Larus, and R. Rajwar, *Transactional Memory,* 2nd ed., Morgan and Claypool, 2010.
4. M. Burtscher and K. Pingali, ''An Efficient CUDA Implementation of the Tree-Based Barnes Hut n-Body Algorithm,'' *GPU Computing Gems,* Emerald ed., Morgan Kaufmann, 2011, pp. 75-92.
5. ''NVIDIA's Next-Gen CUDA Compute Architecture: Fermi,'' white paper, Nvidia, Oct. 2009; http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf.
6. W.W.L. Fung et al., ''Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow,'' *Proc. 40th Ann. IEEE/ACM Int'l Symp. Microarchitecture,* IEEE CS, 2007, pp. 407-420.
7. W.W.L. Fung et al., ''Hardware Transactional Memory for GPU Architectures,'' *Proc. 44th Ann. IEEE/ACM Int'l Symp. Microarchitecture,* ACM, 2011, pp. 296-307.
8. H. Chafi et al., ''A Scalable, Non-blocking Approach to Transactional Memory,'' *Proc. IEEE 13th Int'l Symp. High Performance Computer Architecture* (HPCA 07), IEEE CS, 2007, pp. 97-108.
9. L. Yen et al., ''LogTM-SE: Decoupling Hardware Transactional Memory from Caches,'' *Proc. IEEE 13th Int'l Symp. High Performance Computer Architecture* (HPCA 07), IEEE CS, 2007, pp. 261-272.
10. R. Guerraoui and M. Kapalka, ''On the Correctness of Transactional Memory,'' *Proc. 13th ACM SIGPLAN Symp. Principles and Practice of Parallel Programming* (PPoPP 08), ACM, 2008, pp. 175-184.
11. M.F. Spear, M.M. Michael, and C. von Praun, ''RingSTM: Scalable Transactions with a Single Atomic Instruction,'' *Proc. 20th Ann. Symp. Parallelism in Algorithms and Architectures* (SPAA 08), ACM, 2008, pp. 275-284.
12. L. Dalessandro, M.F. Spear, and M.L. Scott, ''NOrec: Streamlining STM by Abolishing Ownership Records,'' *Proc. 15th ACM SIGPLAN Symp. Principles and Practice of Parallel Programming* (PPoPP 10), ACM, 2010, pp. 67-78.
13. M.M. Michael, ''Practical Lock-Free and Wait-Free LL/SC/VL Implementations Using 64-Bit CAS,'' *Proc. 18th Int'l Symp. Distributed Computing* (DISC 04), LNCS 3274, Springer, 2004, pp. 144-158.
14. A. Bakhoda et al., ''Analyzing CUDA Workloads Using a Detailed GPU Simulator,'' *Proc. IEEE Int'l Symp. Performance Analysis of Systems and Software* (ISPASS 09), IEEE, 2009, pp. 163-174.

15. W.J. Dally and B.P. Towles, *Principles and Practices of Interconnection Networks*, Morgan Kaufmann, 2004.

Wilson W.L. Fung is a PhD student in the Department of Electrical and Computer Engineering at the University of British Columbia. His research interests include parallel-programming models and many-core accelerators. Fung has an MASc in computer engineering from the University of British Columbia.

Inderpreet Singh is pursuing an MASc in the Department of Electrical and Computer Engineering at the University of British Columbia. His research interests include programming models for many-core accelerators. Singh has a BASc in engineering physics from the University of British Columbia.

Andrew Brownsword is a software architect at Intel. He performed the work described in this article while he was a senior software engineer at Electronic Arts. His research interests include improving parallel-programming models on massively parallel machines. Brownsword has a BSc in computer science from the University of British Columbia.

Tor M. Aamodt is an associate professor in the Department of Electrical and Computer Engineering at the University of British Columbia. His research interests include many-core accelerators (such as GPUs), heterogeneous multicore processors, and analytical performance modeling of processor architectures. Aamodt has a PhD in electrical and computer engineering from the University of Toronto. He is a member of IEEE and the ACM.

Direct questions and comments to Wilson W.L. Fung, 2332 Main Mall, Vancouver, BC, Canada V6T 1Z4; wwlfung@ece.ubc.ca.

Selected CS articles and columns are also available for free at http://ComputingNow.computer.org.