IEEE COMPUTER ARCHITECTURE LETTERS, VOL. XX, NO. XX, XX XX

Software Transactional Memory for GPU Architectures

Yunlong Xu*, Rui Wang†, Nilanjan Goswami‡, Tao Li‡, Depei Qian*,†
*School of Electronic and Information Engineering, Xi'an Jiaotong University, Xi'an, China
†School of Computer Science and Engineering, Beihang University, Beijing, China
‡ECE Department, University of Florida, Gainesville, USA
xjtu.ylxu@stu.xjtu.edu.cn, rui.wang@jsi.buaa.edu.cn, nil@ufl.edu, taoli@ece.ufl.edu, depeiq@xjtu.edu.cn

Abstract—To make applications with dynamic data sharing among threads benefit from GPU acceleration, we propose a novel software transactional memory system for GPU architectures (GPU-STM). The major challenges include ensuring good scalability with respect to the massively multithreading of GPUs, and preventing livelocks caused by the SIMT execution paradigm of GPUs. To this end, we propose (1) a hierarchical validation technique and (2) an encounter-time lock-sorting mechanism to deal with the two challenges, respectively. Evaluation shows that GPU-STM outperforms coarse-grain locks on GPUs by up to 20x.

Index Terms—Multicore Processors, Parallel Programming, Run-time Environments, SIMD Processors.

1 Introduction

The increasing computing requirement adds more general-purpose applications to the arena of GPU computing, including applications with dynamic data sharing. The synchronizations on shared memory locations among individual threads are typically implemented using locks constructed from atomic primitives. However, the lock-based synchronization requires a great amount of programming efforts to achieve functional correctness and desirable performance. Furthermore, the SIMT execution paradigm of GPUs exacerbates the challenges of GPU locks (see more in Section 2.2). Therefore, to ensure that GPUs benefit a wide range of real-world workloads, it is imperative to simplify the concurrent programming paradigm of GPU architectures.

Transactional memory (TM) [6] enables atomic operations on an arbitrary set of memory locations. Code that must execute atomically and in isolation is protected by TM infrastructures. Transactions eliminate many pitfalls commonly associated with locks (e.g. deadlock, livelock). Recent efforts have integrated TM into GPUs to simplify fine-grained synchronization on GPUs [1], [4], yet these proposals either allow limited concurrency [1], or require hardware level modification and cannot run on existing GPUs [4]. In this paper, we propose a highly scalable, livelock-free software transactional memory (STM) system for GPUs.

2 GPU and GPU Locks

2.1 GPU Microarchitecture

General-purpose GPU exposes its shader cores as SIMD processing engine, which is augmented with control flow divergence support and memory access optimization mechanism. The shader core has several functional blocks, such as integer/floating point ALUs, load/store units, special functional blocks, and atomic units. Multiple threads (Nvidia *warp*) execute simultaneously in lockstep fashion. On control flow divergences, some thread lanes are masked off. The execution paradigm is called SIMT. Every scheduling cycle, the fetch/schedule unit chooses an instruction PC based on warp formation and scheduling policy. Due to lockstep execution paradigm, only one instruction is decoded per warp. Based on the type of the instruction, appropriate functional blocks are executed simultaneously for all the threads in the warp. Load/store units access the on-chip shared memory or travel through the on-chip interconnect and memory controller to get data from off-chip memory. Load/store units also implement atomic read-modify-write operations and memory access coalescing mechanism. Multiple consecutive accesses to same DRAM row are coalesced to generate a single memory request.

Manuscript submitted: 09-Dec-2012. Manuscript accepted: 11-Feb-2013. Final manuscript received: 13-Feb-2013.

2.2 Pitfalls of GPU Locks

Besides the traditional challenges of lock-based synchronization, concurrency bugs can manifest in new manners due to the SIMT execution paradigm of GPUs.

Consider the situation in Scheme #1 of Fig. 1, here two threads within a warp compete for a spinlock, which is implemented using the compare-and-swap (CAS) primitive. One of them acquires the lock, and waits at the start of critical section for re-convergence, while the other spins forever; eventually it leads to a deadlock. Such deadlock can be avoided using one of the two methods. Firstly, using Scheme #2 of Fig. 1, we can perform serialization within each warp. Unfortunately, this could lead to extremely low hardware utilization. Secondly, instead of spinning, threads within each warp diverge on lock acquisition failures (see Scheme #3 of Fig. 1). Scheme #3 executes correctly when each thread acquires a single lock. However, when each thread acquires multiple locks in uncertain order, it results in livelock due to circular locking phenomenon within individual warps. For example, in a case that two threads from the same warp attempt to acquire two locks in reverse orders, when a thread incurs locking failure, it releases the locks acquired and retries. Since warps execute the same instruction in lockstep fashion, those two threads loop forever. Due to this livelocking issue, fine-grained locking on GPUs is extremely challenging, even impossible in some cases.

3 A Scalable, Livelock-free STM for GPUs

Massively multithreading of GPUs requires the scalability of proposed STM system, which should support concurrently execution and committing of 1000s of transactions. Also, the

```
/*Schemes #1 and #3 are originally discussed in [4], [9], and
scheme #2 is adapted from [10]*/
Scheme #1 [4], [9]: spinlocks
1.    while (CAS(&lock,0,1)==1), spin
2.    //critical section...
3.    CAS(&lock,1,0)

Scheme #2 [10]: serialization within warps
1.    for (int i=0; i<WARP_SIZE; i++)
2.        if (threadIdx.x%WARP_SIZE == i)
3.            while (CAS(&lock,0,1)==1), spin
4.            //critical section...
5.            CAS(&lock,1,0)

Scheme #3 [4], [9]: diverging on locking failures
1.    done = false
2.    while (done==false)
3.        if (CAS(&lock,0,1)==0)
4.            //critical section...
5.            CAS(&lock,1,0), done = true
```

Fig. 1: Lock implementations on GPUs

characteristics of GPUs (i.e. SIMT execution, and memory access coalescing) should be considered. Otherwise, the proposed STM would incur incorrect execution or higher overhead.

3.1 STM Infrastructure

GPU-STM is a word- and lock-based STM, which integrates three novel ideas: (1) hierarchical validation; (2) encounter-time lock-sorting; and (3) coalesced read/write buffers.

Hierarchical validation – Value-based validation (VBV) and timestamp-based validation (TBV) are two common conflict detection strategies used in STM systems.

VBV records the actual values of locations read by a transaction and checks these to detect conflicts. To ensure opacity [5], STM systems that adopt VBV alone have to validate incrementally after each transactional read. This can introduce nontrivial performance overhead. A single global sequence lock can be used to filter out unnecessary validations. Together with the single lock, VBV can lead to fast systems on CPUs (e.g. NOrec [2]) since it does not need to access other shared metadata. However, this scheme cannot scale well on GPUs, because (1) the lock would be updated frequently by 1000s of transactions, and (2) during committing, memory updates of all transactions are serialized by the lock.

Unlike VBV, TBV uses global version locks to manage the entire memory. Each version lock indicates the version of a memory stripe. A transaction is invalidated when its snapshot (the version of memory it accessed) is found out of date. Comparing with VBV, TBV can result in less number of compare instructions and less off-chip memory traffic. However, transactions that access locations managed by the same lock may incur false conflicts, which can be avoided by using VBV. False conflicts can hamper the scalability of TM systems. The SIMT execution paradigm of GPUs exacerbates the side effect of false conflicts. On conflicts some thread lanes have to be masked off. The thread lanes would have been masked off unnecessarily if the conflicts were false; this results in low hardware utilization and thus degrades performance. False conflicts can be reduced by increasing the number of global version locks. However, the number of global version locks should not be too large, otherwise the storage overhead and the performance impact of metadata-induced cache pressure would be significant even for a small workload.

To ensure scalability, GPU-STM adopts hierarchical validation (HV) that combines TBV and VBV. GPU-STM validates transactions in two scenarios. In addition to the commit-time validation, it performs post-validation after each read to ensure opacity. In both scenarios, a transaction first compares the corresponding global version locks with its snapshot. Only if the snapshot is out of date, it performs VBV to confirm that the memory locations accessed is still consistent. In case of validation failure, the transaction is aborted. We argue that HV is comparable with TBV in common cases, and can deal with corner cases efficiently without increasing the number of global locks accordingly. Because for applications with small amount of shared data, the execution path of HV is mostly the same as that of TBV, and HV can exploit VBV to avoid false conflicts when a large amount of shared data is concurrently accessed.

Encounter-time lock-sorting – GPU-STM uses locks to ensure isolation of validation and memory updates of individual transactions. Scheme #3 described in Section 2.2 is used to acquire the locks. However, this scheme may incur livelocks. The typical solution is exponential backoff strategy. When applying to GPU-STM, it requires a transaction that incurs locking failure to wait for a random, exponentially increasing delay before retrying, thus can practically avoid livelocks. However, this strategy along with the SIMT execution can result in a lot of inactive thread lanes on GPUs.

To address the livelocking issue, we propose encounter-time lock-sorting which is coupled with commit-time locking. Each transaction maintains a local lock-log. On each read/write, a lock is inserted into the correct position in an already sorted lock-log. The time complexity of lock-sorting is $O(n^2)$, where n is the number of locks it encountered. Thus, lock-sorting would introduce non-trivial overhead. To reduce the overhead, we organize local lock-logs in order-preserving hash tables. The order that the locks are sorted by is derived from the ordinal relation of addresses being read or written during transaction execution. An incoming lock is hashed into a bucket, and inserted into a correct position afterwards. Eventually, the number of comparison steps is reduced. Here the well-known insertion sort algorithm is used for lock-sorting within each bucket. When a transaction commits, it sequentially processes each bucket and each lock within each bucket. Thereby, a global order of lock acquisition is maintained among all transactions, and livelock-freedom is ensured.

Coalesced read/write buffers – GPU-STM leverages the memory access coalescing mechanism to reduce the overhead of transaction bookkeeping. The read-/write-sets of transactions within the same warp are merged in a way so that the transactions within a warp can access consecutive locations. Each thread of a warp executes a transaction at a time, and each thread uses its index within the warp to access an independent partition of the merged read-/write-set. Usually 32 threads form a warp, each thread has a unique index (from 0 to 31), then entry i of a merged read-/write-set belongs to thread j if $(i \ mod \ 32) = j$. However, accesses to the local lock-logs are not coalesced. The penalty of uncoalesced memory accesses can be reduced by multi-level caches of GPUs.

3.2 STM Implementation

This section describes the implementation of GPU-STM, which consists of three components: the STM metadata, the STM runtime algorithms, and the memory fences used.

Metadata – GPU-STM comprises two sets of metadata (as listed in Fig. 2): global metadata shared among transactions, and local metadata private to each transaction. The global lock

```
global unsigned global_clock
global <Version, Lock-Bit> global_locktable[]
local unsigned snapshot
local <Address, Value> reads[]
local <Address, Value> writes[]
local Hash<IndexToGlobalLock, WR-Bit, RD-Bit> local_locktable[]
```

Fig. 2: GPU-STM metadata

table is an array of version locks, each of which is an unsigned integer with the least significant bit indicating whether a stripe of memory is locked, and with the rest bits indicate the version of a memory stripe. Each transaction maintains a thread local snapshot of the global clock. Each transaction has its own read-set and write-set. Read-/write-sets of transactions within the same warp have coalesced organization. Each transaction maintains a hash table to sort the locks it encountered during execution. Each entry of a local lock-table indexes to a global lock. The lowest two bits of each entry indicate whether the transaction has written to, or read from the memory stripe managed by the global lock, thus the two bits are referred to as write-bit and read-bit, respectively.

TXBegin – Each transaction begins by reading the global clock. This snapshot value indicates the most recent time when the transaction was known to be consistent.

TXRead – Read barrier first checks whether the transaction has written the location. If the location has been written, it returns value from write-set. Otherwise, it (1) reads a value from memory, (2) logs the address/value pair to read-set for future validation, (3) checks for opacity, (4) computes a global lock index based on the address it reads from, (5) inserts the index into the local lock-table for commit-time locking, and (6) sets the read-bit of the local lock. For opacity checking, the read barrier compares the corresponding global version with the local snapshot. If the snapshot is out of date, it performs value-based validation (VBV) to confirm that the transaction is still consistent. If validation is passed, it updates the snapshot with the global version. Otherwise, the transaction is aborted.

TXWrite – The write barrier (1) updates the redo-logs with the value written, (2) computes a global lock index based on the address it writes to, (3) inserts the index into the local lock-table, and (4) sets the write-bit of the local lock.

TXCommit – A read-only transaction does not require validation since it linearizes at the time of the last read. Otherwise, a transaction first tries to acquire the global locks indicated by its local lock-table. On locking failure, it retries after transactions within the same warp finish committing. If the read-bit of a local lock is set, the transaction compares the corresponding global version with the local snapshot. If the snapshot is out of date, it sets a flag to trigger VBV later. When the transaction has successfully acquired all of its locks, it validates read locations using VBV only if any previous timestamp-based validation fails. If validation is passed, the transaction (1) makes its speculative updates visible by looping through its write-set, (2) increases the global clock by one, (3) updates corresponding global version locks with new global clock, and (4) releases the locks acquired. Otherwise, it releases the locks acquired and aborts.

Memory fences – STM systems typically need fences in weak memory models (e.g. GPU memory model) to ensure that subsequent accesses are not hoisted [11]. CUDA provides $\_\_threadfence$ that can stall current thread until its prior global memory and shared memory accesses complete. The fences are used in three runtime functions ($TXBegin$, $TXRead$ and $TXCommit$) of GPU-STM to ensure the order between accesses to metadata and main memory. In $TXBegin$, each transaction reads the global clock at beginning time. A fence is used to order such read prior to transaction execution. In $TXRead$, a fence is put between the access of program data and opacity checking. In $TXCommit$, two fences are put before and after memory updates of write-set, respectively.

TABLE 1: Application Characteristics.[a]

Name	Shared Data	RD /TX	WR /TX	TX /Kernel	TX Time	Conflicts
RA	8M	16	16	1M	High	Low
HT	256K	8	8	1M	High	Medium
GN[b]	16K /1M	1	1	4M /1M	Medium /High	Low /Medium
LB	1.75M	352	352	512	Low	Low
KM	2K	32	32	64K	Medium	High

[a] Shared data: number of data shared among TXs; RD/TX, WR/TX: number of shared data read, written by each TX; TX/kernel: number of TXs per kernel; TX time: proportion of time spent in TXs; conflicts: the probability of conflict.
[b] GN has two transaction kernels.

3.3 Semantics

GPU-STM provides weak isolation, where transactions are isolated from other transactions rather than both other transactions and concurrent non-transactional accesses.

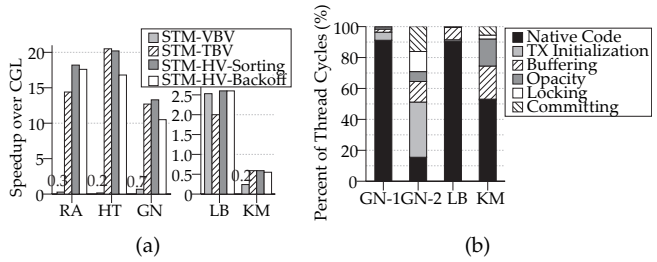
Because the hardware SIMT stack of GPUs is not manageable from software, GPU-STM requires each transaction to maintain a failure-flag to support transaction aborts. When opacity is required, programmers should access the flag and take measure to abort a transaction. This programming burden can be eliminated by future compiler or hardware supports.

4 Evaluation

In this section, we present a preliminary performance evaluation of GPU-STM. We implement GPU-STM on top of CUDA runtime on an Nvidia 14-shader 1.15 GHz C2070 Fermi GPU, and transactify applications using GPU-STM runtime APIs. The local metadata and global metadata of GPU-STM are both stored in the global memory. The local metadata is cached at the L1 and L2 levels; while the global metadata is only cached at the L2 level, since the L1 cache on Fermi GPUs is not coherent. Two micro-benchmarks and three workloads ported from STAMP benchmarks [7] are used in the evaluation. In $randomarray$ (RA) micro-benchmark, each transaction randomly accesses multiple locations of a shared array. In $hashtable$ (HT) micro-benchmark, each transaction inserts multiple elements into a shared hash table. The STAMP workloads used are $labyrinth$ (LB), $genome$ (GN), and $k\text{-}means$ (KM). Table 1 presents the characteristics of the workloads.

4.1 Performance Comparison

We implement four STM variants which adopt NOrec-alike value-based validation (STM-VBV), timestamp-based validation (STM-TBV), and hierarchical validation with encounter-time lock-sorting (STM-HV-Sorting) and with backoff (STM-HV-Backoff), respectively. STM-HV-Backoff adopts a backoff scheme specific to GPU to reduce inactive thread lanes caused by exponential backoff. When lock acquisition is required, transactions within a warp try to acquire locks in parallel in the first attempt. Those who incur locking failures try to acquire locks again sequentially, while the others are waiting. Transactions that have acquired locks successfully perform validation



(a) (b)

Fig. 3: (a) Performance comparison between STM variants and CGL on GPU. (b) Execution time breakdown of a single-thread, includes: native-code execution, TX initialization, buffering, opacity checking, acquiring/releasing locks, and committing.

and memory updates in parallel before the failed transactions retry. Both hierarchical and timestamp-based validation use 1M global version locks. The STM performance is measured in speedup over coarse-grained locking (CGL), which serializes critical sections with a single global lock, on GPUs. Due to the complexity and even impossibility (for RA, HT, and LB), we do not implement fine-grained locking.

As shown in Fig. 3 (a), STM-VBV yields undesirable performance for workloads with a large number of transactions (e.g. RA, HT, GN, and KM) due to its limited scalability. STM-TBV and STM-HV (both Sorting and Backoff) exhibit noticeable speedup over CGL for workloads with modest conflicts among transactions (e.g. RA, HT, GN, and LB). KM does not benefit from STM parallelization due to high conflict rate, which is caused by relatively small amount of shared data competed by many transactions. Moreover, the benefit of STM parallelization varies with the proportion of time spent in transactions. Workloads with higher proportion of transaction time exhibit higher speedup (e.g. RA, HT, and GN). Nevertheless, STM parallelization is also crucial to the workloads with less transaction time (e.g. LB), since critical sections would have to be serialized otherwise. Furthermore, STM-HV-Sorting outperforms STM-TBV (which also sorts locks) for workloads with relatively large amount of shared data (e.g. RA and LB). On these workloads, the amount of shared data (8M for RA, 1.75M for LB) is larger than that of global version locks (1M); this makes timestamp-based validation of STM-TBV suffers from false conflicts. For the rest workloads, STM-HV-Sorting is comparable with STM-TBV. STM-HV-Sorting outperforms STM-HV-Backoff for almost all evaluation workloads.

4.2 Execution Time Breakdown

This section evaluates the overhead of STM system by examing the execution time breakdown of a single-thread, as presented in Fig. 3 (b). The second kernel of GN (GN-2) incurs significant STM overhead, since it spends a large proportion of time in transactions, and transactional reads and writes account for a large proportion of the transactions. This makes STM overhead, especially transaction initialization overhead, difficult to be amortized by the native code execution. The rest three kernels incur less overhead (∼20% on average). Workloads with larger transaction read- and write-sets (e.g. LB and KM) incur higher buffering overhead. KM also incurs high opacity overhead due to large read-sets as well as small amount of shared data. This is because opacity-checking after each read incurs frequent failures in snapshot validation, and the system selects value-based validation instead. Note that the single-thread overhead of GPU-STM can be amortized by the additional scalability it enables. For instance, though with significant overhead, GN-2 still exhibits ∼20x speedup. However, workloads with limited inherent scalability (e.g. KM) are not suited for GPU-STM.

5 Related Works

Cederman et al. [1] propose lock-based STMs on GPUs. Compared with the STMs, our GPU-STM supports per thread transactions rather than per thread-block transactions. Further, GPU-STM uses hierarchical validation to ensure scalability, and encounter-time lock-sorting to ensure livelock freedom. Fung et al. propose KILO TM [4], a hardware transactional memory for GPU architectures. Similar to GPU-STM, KILO TM also supports per thread transactions, employs value based conflict detection, and ensures logs are coalesced. Unlike GPU-STM, KILO TM proposes mechanisms for SIMT control flow divergence handling related to transaction execution. GPU-STM is implemented entirely in software, which can be used on existing GPUs without incurring any hardware overhead. Moreover, in KILO TM, the concurrency in commit phase is constrained by the number and capacities of commit units. GPU-STM allows much higher concurrency. Timestamp-based validation is also used by TL2 [3], and value-based validation is used by JudoSTM [8] and NOrec STM [2].

6 Conclusion

We propose a scalable and livelock-free STM for GPU architectures to enable GPU as a further general-purpose computing platform. Evaluation shows that dynamic data sharing on GPUs can be easily expressed with GPU-STM. Preliminary results show that GPU-STM outperforms coarse-grain locks on GPUs significantly.

Acknowledgments

This work is supported by NSF of China under grant 61133004, 61128004 and 61073011, and 863 Program of China under grant 2012AA010902.

References

[1] D. Cederman, P. Tsigas, and M. T. Chaudhry, "Towards a Software Transactional Memory for Graphics Processors," in *Proc. of the Eurographics Symp. on Parallel Graphics and Visualization*, 2010.
[2] L. Dalessandro, M. F. Spear, and M. L. Scott, "NOrec: Streamlining STM by Abolishing Ownership Records," in *PPoPP'10*.
[3] D. Dice, O. Shalev, and N. Shavit, "Transactional Locking II," in *DISC'06*.
[4] W. W. L. Fung, I. Singh, A. Brownsword, and T. M. Aamodt, "Hardware Transactional Memory for GPU Architectures," in *MICRO'11*.
[5] R. Guerraoui and M. Kapalka, "On the Correctness of Transactional Memory," in *PPoPP'08*.
[6] M. Herlihy and J. E. B. Moss, "Transactional memory: architectural support for lock-free data structures," in *ISCA'93*.
[7] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun, "STAMP: Stanford Transactional Applications for Multi-Processing," in *IISWC'08*.
[8] M. Olszewski, J. Cutler, and J. G. Steffan, "JudoSTM: A Dynamic Binary-Rewriting Approach to Software Transactional Memory," in *PACT'07*.
[9] A. Ramamurthy, "Towards Scalar Synchronization in SIMT Architectures," Master's thesis, University of British Columbia, 2011.
[10] J. Sanders and E. Kandrot, *CUDA by Example: An Introduction to General Purpose GPU Programming*. Addison-Wesley Professional, 2010, ch. 9.
[11] M. F. Spear, M. M. Michael, M. L. Scott, and P. Wu, "Reducing Memory Ordering Overheads in Software Transactional Memory," in *CGO'09*.