This is author's own version. The authoritative version will appear in CGO 2014.

Software Transactional Memory for GPU Architectures

Yunlong Xu[⋆], Rui Wang[†], Nilanjan Goswami[‡], Tao Li[‡], Lan Gao[†], 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, lan.gao}@jsi.buaa.edu.cn
nil@ufl.edu, taoli@ece.ufl.edu, depeiq@xjtu.edu.cn

ABSTRACT

Modern GPUs have shown promising results in accelerating computation intensive and numerical workloads with limited dynamic data sharing. However, many real-world applications manifest ample amount of data sharing among concurrently executing threads. Often data sharing requires mutual exclusion mechanism to ensure data integrity in multithreaded environment. Although modern GPUs provide atomic primitives that can be leveraged to construct fine-grained locks, lock-based synchronization requires significant programming efforts to achieve functional correctness. The massive multithreading and SIMT execution paradigm of GPUs further extend the challenges of GPU locks.

To make applications with dynamic data sharing 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 massive 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. We build our GPU-STM prototype based on the commercially available GPU platform and runtime. Our real system based evaluation shows that GPU-STM outperforms coarse-grain locks on GPUs by up to 20x.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel programming*

General Terms

Algorithms, Design

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
CGO '14, February 15 - 19 2014, Orlando, FL, USA
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2670-4/14/02$15.00.
http://dx.doi.org/10.1145/2544137.2544139.

Keywords

General-Purpose GPU Computing, Software Transactional Memory, Parallel Programming

1. INTRODUCTION

Recently, the graphics processing unit (GPU) has been widely adopted for general-purpose computing due to its massive multithreading capability and cost-effectiveness. The general-purpose GPU computing is conventionally used for parallel computation that has limited dynamic data sharing. Emerging workloads are showing more involved dynamic data sharing among the threads. The synchronizations on shared memory locations among individual threads are typically addressed in two ways. Firstly, programmers can avoid synchronizations via ad-hoc implementations in some cases [10, 14, 16]. However, the ad-hoc implementations are not generic in nature. Secondly, the synchronizations can be implemented using locks that are constructed from atomic primitives [9]. However, the lock-based synchronization requires significant programming efforts to achieve functional correctness and desirable performance. The problem becomes worse on GPUs due to the concurrent execution of a large number of threads. Furthermore, the SIMT execution paradigm of GPUs further extends 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) [13, 11, 21] could be a promising alternative to simplify such complex concurrency issues. TM enables atomic operations on an arbitrary set of memory locations. 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 [3, 7], yet these proposals either allow limited concurrency [3], or cannot run on existing GPUs [7]. Cederman et al. [3] propose two software transactional memory (STM) systems (blocking/non-blocking) on GPUs. However, the STMs proposed can only support transactional execution at the granularity of thread blocks (i.e. per-thread-block transactions) instead of individual threads (i.e. per-thread transactions). As a result, the STMs allow limited transaction concurrency, which results in low utilization of the massive multithreading capability of GPUs. Fung et al. propose KILO TM [7], a hardware transactional memory (HTM) for GPU architectures. Though the KILO TM supports per-thread transactions, it requires hardware level modification and cannot run

on existing GPUs.

In this paper, we propose a highly scalable, livelock-free software transactional memory (STM) system for GPUs, which supports per-thread transactions. A STM system that supports per-thread transactions faces new challenges due to the distinct characteristics of GPUs. The major challenges include ensuring good scalability of the STM system with respect to the massive multithreading of GPUs, and preventing livelocks caused by the SIMT execution paradigm of GPUs. To address the two challenges, we propose (1) a hierarchical validation technique and (2) an encounter-time lock-sorting mechanism, respectively. The hierarchical validation technique combines timestamp-based validation with value-based validation to eliminate false conflicts of timestamp-based validation and high overhead of value-based validation. When coupled with commit-time locking, the encounter-time lock-sorting mechanism allows all transactions to grab the locks in the same order to avoid livelocks.

We build our GPU-STM prototype based on the commercially available GPU platform and runtime. Therefore, it allows programmers to develop GPU applications with transactional memory without obligations from GPU hardware vendors. The GPU hardware vendors can use this to gather developer's feedbacks before committing to build special hardware for TM.

The rest of the paper is organized as follows. In Section 2, we briefly introduce the GPU microarchitecture, and describe the pitfalls of GPU locks in detail. We present our new GPU-STM algorithms in Sections 3, and evaluate their effectiveness in Section 4. Section 5 discusses related works. Section 6 concludes the paper.

2. GPU AND GPU LOCKS

2.1 GPU Microarchitecture

As a general-purpose processor, 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 is the main SIMD processing engine and has several functional blocks, such as integer/floating point ALUs, load/store units, special functional blocks.

Multiple threads (Nvidia *warp*) execute simultaneously in a lockstep fashion. On control flow divergences, some thread lanes are masked off. The execution model is called Single Instruction Multiple Threads (SIMT). At each scheduling cycle, the fetch/schedule unit chooses an instruction program counter (PC) based on warp formation and scheduling policy. Due to the lockstep execution paradigm, only one instruction is decoded per warp. Based on the type of the instruction, appropriate functional units are exercised 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 obtain data from off-chip memory. The load/store units also implement atomic operations and memory access coalescing mechanism. Multiple consecutive accesses to same DRAM row are coalesced to generate a single memory request.

Execution of general-purpose programs on heterogeneous GPU/CPU architectures is realized by various application programming interfaces (API) such as CUDA (Nvidia) [17], and OpenCL [1]. Using these APIs a programmer can launch thousands of parallel threads onto GPU device from the host CPU. Recent GPU architectures supports thread launch from GPU kernels as well. Several warps form a thread-block and synchronize either implicitly at the end of execution or explicitly after a barrier instruction. Thread-blocks group together to form a thread-grid that executes a GPU kernel (set of GPU instruction sequence).

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 shown in Scheme #1 of Algorithm 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 Algorithm 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 Algorithm 1). Scheme #3 executes correctly when each thread acquires a single lock. However, when each thread acquires multiple locks in uncertain order, it could result in livelock due to circular locking phenomenon within individual warps. For example, consider two threads from the same warp attempting 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.

Algorithm 1 Lock implementations on GPUs

```
    /* Schemes #1 and #3 are originally discussed in [7, 19], and
    scheme #2 is adapted from [20] */
    Scheme #1 [7, 19]: spinlocks
 1: repeat locked ← CAS(&lock, 0, 1)
 2: until locked = 0
 3: critical section...
 4: lock ← 0

    Scheme #2 [20]: serialization within each warp
 5: for i ← 1 to WARP_SIZE do
 6:     if threadIdx.x%WARP_SIZE = i then
 7:         repeat locked ← CAS(&lock, 0, 1)
 8:         until locked = 0
 9:         critical section...
10:         lock ← 0

    Scheme #3 [7, 19]: diverging on locking failures
11: done ← false
12: while done = false do
13:     if CAS(&lock, 0, 1) = 0 then
14:         critical section...
15:         lock ← 0
16:         done ← true
```

3. A SCALABLE, LIVELOCK-FREE STM FOR GPUS

GPUs exhibit three major distinct characteristics: massive multithreading, SIMT execution, and memory

access coalescing. Such characteristics should be considered when designing a STM system for GPUs. Otherwise, the STM system would incur poor scalability, livelocks or higher overhead.

3.1 STM Infrastructure

GPU-STM is a word- and lock-based STM system. In response to the characteristics of GPUs, GPU-STM integrates three novel ideas: (1) hierarchical validation, (2) encounter-time lock-sorting and (3) coalesced read-/write-set organization.

Hierarchical validation – Value-based validation (VBV) [18, 5] and timestamp-based validation (TBV) [6] are two common conflict detection strategies used in STM systems.

VBV records the actual values of locations read by a transaction and checks them to detect conflicts. To ensure opacity [8] (which requires transactions observe a consistent view of memory), STM systems that adopt VBV alone have to perform incremental validation: a transaction has to validate all past transactional reads after each new read. This can introduce nontrivial performance overhead. A single global sequence lock (as used in NOrec [5]) can be used to filter out unnecessary validations. Combined with the single lock, VBV can lead to fast systems on CPUs (e.g. NOrec [5]), since it does not need to access other shared metadata. However, this scheme cannot scale well on GPUs, because (1) the single lock is updated frequently by thousands of transactions, and (2) during commit, memory updates of all transactions are serialized by the single 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 reduce the number of compare instructions and off-chip memory traffic, thus reduce performance overhead. However, transactions that access locations managed by the same version lock may incur false conflicts, which can be avoided by using VBV. False conflicts can hamper the scalability of TM systems. Compared with independent thread execution on CPUs, the lockstep execution of GPUs exacerbates the side effect of false conflicts. On conflicts some thread lanes have to be masked off, and the transactions of the masked-off threads would have to be re-executed later. 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 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, our proposed 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 the transaction observes a consistent view of memory. 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 locations accessed is still consistent. In case of validation failure, the transaction is aborted. We argue that HV is nearly comparable with TBV in common cases, and can deal with the corner cases efficiently without increasing the number of global version locks. For applications with small amount of shared data, the execution path of HV is mostly the same as that of TBV, and when a large amount of shared data is concurrently accessed, HV can exploit VBV to avoid false conflicts.

Encounter-time lock-sorting – GPU-STM uses locks to ensure the isolation of validation and memory updates of individual transactions during commit. Note that although the scheme #3 described in Section 2.2 can be used to acquire the locks, it may incur livelocks. To avoid this, a common practice is to use the exponential backoff strategy. This strategy requires a transaction that incurs locking failure to wait for a random, exponentially increasing delay before retrying, thus can practically avoid livelocks. However, the exponential backoff cannot work on GPUs, because transactions within the same warp cannot wait for different delays due to the lockstep execution.

To address the livelocking issue, we propose encounter-time lock-sorting coupled with commit-time locking. To be more specific, each transaction maintains a local lock-log. On each read/write, a lock is inserted into a corresponding position in an already-sorted lock-log. The order that the locks are sorted by is derived from the ordinal relation of addresses being read or written during transaction execution. For example, if the size of the global lock-table is 2^{20}, and for a 32-bit address, we use the 2nd-21th of address bits to identify the lock that manages the address. We sort the locks by their IDs, so that a global order can be obtained when acquiring locks. The time complexity of encounter-time lock-sorting is $O(n^2)$ where n is the number of locks it encountered, because each incoming lock is compared with the locks that have already existed in the lock-log. Therefore, lock-sorting would introduce nontrivial overhead for transactions with large read-/write-sets.

To reduce the overhead, we organize local lock-logs in order-preserving hash tables. An incoming lock is hashed into a bucket, and inserted into a corresponding position afterwards. Eventually, the number of comparison steps is reduced. Note that a lock is not inserted when the same lock already exists in the local lock-log. Thereby, duplication of locks is avoided. When a transaction commits, it sequentially processes each bucket and the locks within each bucket. In this way, a global order of lock acquisition is maintained among all transactions. Hence, livelock-freedom is ensured, and no backoff mechanism is required. Such livelock-freedom guarantees system-wide progress in GPU-STM: some thread always makes progress.

Coalesced read-/write-set organization – Similar to KILO TM [7], GPU-STM leverages the memory access coalescing mechanism to reduce the overhead of transaction bookkeeping. The read-/write-sets of all transactions within each warp are merged in a way so that the transactions can access consecutive locations. Each thread of a warp executes a transaction at a time, and 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 $(i \ mod \ 32) = j$.

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 miscellany (i.e. memory fences, and register checkpoint).

3.2.1 Metadata

GPU-STM comprises two sets of metadata (as listed in Algorithm 2): the global metadata that is shared among transactions, and the local metadata that is private to each transaction. The global lock 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 the rest of the bits indicating 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 the transactions within each warp have coalesced organization as described in Section 3.1. Each transaction maintains a hash table to sort the locks encountered during execution. Each entry of the hash 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. The two bits are referred to as write-bit and read-bit, respectively.

Algorithm 2 GPU-STM metadata

1: global unsigned *g_clock*
2: global $\langle Version, Lock\text{-}Bit \rangle$ *g_lockTab*[ ]
3: local unsigned *snapshot*
4: local $\langle Address, Value \rangle$ *reads*[ ]
5: local $\langle Address, Value \rangle$ *writes*[ ]
6: local Hash$\langle IndexToGlobalLock, WR, RD \rangle$ *l_lockTab*[ ]

Using the global version locks, GPU-STM can detect conflicts between transactional accesses. However, the conflicts between transactional and non-transactional accesses cannot be detected, since global version locks of the STM system do not protect the latter. This mechanism offers weak isolation [2].

3.2.2 Algorithm

TXBegin – Each transaction begins by reading the global clock at the point when it starts (line 4 in Algorithm 3)[1]. 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 to the location (line 22). Here, a bloom filter for each transaction is used to compress the write-set. If the location has been written, it returns value from the write-set. Otherwise, it (1) reads a value from memory (line 24), (2) logs the address/value pair to read-set for future validation (line 25), (3) checks for the consistency of all memory reads performed by the transaction (line 27-33), (4) computes a global lock index based on the address it reads from, inserts the index into the local lock-table for commit-time locking, and sets the read-bit of the local lock (line 34).

We explain the consistency checking in details below. The read barrier first reads the global version of the location, and checks whether the location has been locked. Since GPU-STM adopts commit-time locking, the location can only be locked by a committing transaction. If the location has been locked, the read barrier waits until the location is released (line 27-29). Note that, a location locked by a committing transaction cannot be released until all the memory updates of the transaction have been committed (see *TXCommit* below). This guarantees that all of the memory updates of the committing transaction can be seen by current transaction during consistency checking.

When the location is unlocked, the read barrier compares the corresponding global version with the local snapshot (line 31). If the local snapshot is valid, it suggests that the location has not been written by any other transaction since the snapshot was verified last time. In this case, consistency is ensured and no further validation is needed. However, even if the snapshot is out-of-date, the transaction may still be consistent, because the locations within the read-set of the transaction may have not been updated by any other transaction. To confirm this, the read barrier performs post-validation (line 32). During post-validation, the read barrier uses value-based validation (VBV) to check whether the locations have been updated by any other transaction (line 9-11). If the value of a location has been changed, the transaction is aborted.

However, a simple VBV is insufficient. We also need to ensure that the locations validated during VBV had not been updated by other concurrent transactions. Therefore, the read barrier compares the global versions of the locations with a particular snapshot (line 13-19) after VBV is passed. The particular snapshot is obtained before the value of the first location is checked. If a location is found locked, or the snapshot is out-of-date (line 17), the VBV is restarted. This is because, in these cases, a location possibly had been updated by other concurrent transactions during VBV. If post-validation is passed, the transaction is verified to be consistent (at the time of the particular snapshot). Otherwise, the transaction is aborted.

Since the hardware SIMT stack of GPUs is not manageable from software, GPU-STM requires each transaction to maintain an opacity flag (line 3 and 33) to support transaction aborts. Programmers can access the flag and take measure to abort a running transaction that observes an inconsistent view of memory. This programming burden can be eliminated by future compiler or hardware supports.

TXWrite – The write barrier (1) updates the write-set (line 37), (2) computes a global lock index based on the address it writes to, inserts the index into the local lock-table, and sets the write-bit of the local lock (line 38).

TXCommit – A read-only transaction does not require validation, since it linearizes at the time of the last read (line 68). Otherwise, a transaction first tries to acquire the global locks indicated by its local lock-table (line 73). On locking failure, it retries after transactions within the same warp finish committing. If the read-bit of a local lock-entry is set, the transaction also compares the corresponding global version with the local snapshot (line 50). If the snapshot is out-of-date, it sets a flag to trigger VBV later (line 51). When the transaction has successfully acquired all of its locks, it validates read locations using VBV only if any previous timestamp-based validation fails (line 76). If validation is passed, the transaction (1) makes its speculative updates visible by looping through its write-set (line 80-81), (2) increases the global clock by one (line 83), (3) updates corresponding global version locks with new global clock,

[1] In the rest of Section 3, all lines specified in brackets indicate lines in Algorithm 3.

Algorithm 3 GPU-STM algorithm

```
 1: void TXBegin()
 2:     reads ← writes ← l_lockTab ← ∅
 3:     isOpaque ← passTBV ← true
 4:     snapshot ← g_clock
 5:     __threadfence()

 6: bool PostValidation(Unsigned version)
 7:     snapshot ← version
 8:     loop:
 9:         for all ⟨addr, val⟩ ∈ reads do
10:             if *addr ≠ val then
11:                 return false
12:         __threadfence()
13:         for all ⟨addr, val⟩ ∈ reads do
14:             versionLock ← g_lockTab[hash(addr)]
15:             isLocked ← (versionLock&1)
16:             tmpVer ← (versionLock >> 1)   ▷ >>: right-shift
17:             if (isLocked ≠ 0) ∨ (tmpVer > snapshot) then
18:                 snapshot ← tmpVer
19:                 goto loop
20:         return true

21: Value TXRead(Address addr)
22:     if ⟨addr, valWritten⟩ ∈ writes then
23:         return valWritten
24:     val ← *addr
25:     reads ← reads ∪ {⟨addr, val⟩}
26:     __threadfence()
27:     do
28:         versionLock ← g_lockTab[hash(addr)]
29:     while {(versionLock&1) ≠ 0}
30:     version ← versionLock >> 1
31:     if version > snapshot then
32:         if ¬PostValidation(version) then
33:             isOpaque ← false              ▷ tx should be aborted
34:     l_lockTab ← l_lockTab ∪ {⟨hash(addr), 0, 1⟩}
35:     return val

36: void TXWrite(Address addr, Value val)
37:     writes ← writes ∪ {⟨addr, val⟩}
38:     l_lockTab ← l_lockTab ∪ {⟨hash(addr), 1, 0⟩}

39: Atomic_or(Unsigned *addr, Unsigned val)
40:     atomic{old ← *addr; *addr ← old | val; return old;}

41: Atomic_inc(Unsigned *addr)
42:     atomic{old ← *addr; *addr ← old + 1; return old;}

43: bool GetLocksAndTBV()
44:     for all ⟨i, wr, rd⟩ ∈ l_lockTab do
45:         versionLock ←Atomic_or(g_lockTab[i], 1)
46:         if (versionLock&1) ≠ 0 then
47:             ReleaseLocks()
48:             return false
49:         if rd = 1 then
50:             if (versionLock >> 1) > snapshot then
51:                 passTBV ← false
52:     return true

53: void ReleaseLocks()
54:     for all global_lock i acquired do
55:         g_lockTab[i] ← g_lockTab[i] − 1

56: void ReleaseAndUpdateLocks(Unsigned version)
57:     for all ⟨i, wr, rd⟩ ∈ l_lockTab do
58:         if wr = 1 then
59:             g_lockTab[i] ← version << 1        ▷ <<: left shift
60:         else
61:             g_lockTab[i] ← g_lockTab[i] − 1

62: bool VBV()
63:     for all ⟨addr, val⟩ ∈ reads do
64:         if *addr ≠ val then
65:             return false
66:     return true

67: bool TXCommit()
68:     if read-only transaction then
69:         return true
70:     loop:
71:         if ¬VBV() then ▷ optional, to reduce lock contention
72:             return false
73:         if ¬GetLocksAndTBV() then
74:             goto loop
75:         if passTBV ≠ true then
76:             if ¬VBV() then
77:                 ReleaseLocks()
78:                 return false
79:         __threadfence()
80:         for all ⟨addr, val⟩ ∈ writes do
81:             *addr ← val
82:         __threadfence()
83:         version ← Atomic_inc(g_clock) + 1
84:         ReleaseAndUpdateLocks(version)
85:         return true
```

and releases the locks acquired (line 84). Otherwise, it releases the previously locks acquired and aborts (line 77-78).

Unlike CPU STM systems where only the locations being written to is locked during commit, the GPU-STM system locks all read/write locations of the transaction before validation. Otherwise, some transactions can never succeed due to lockstep execution. For example, consider the following situation in which locations being read are left unlocked during commit. Transactions *T1* and *T2* within the same warp share two locations X and Y. *T1* reads Y and updates X, while *T2* reads X and updates Y. Since *T1* and *T2* are within the same warp, they execute in lockstep fashion. During commit, *T1* and *T2* first lock their write accesses (X for *T1*, and Y for *T2*), respectively. Afterwards, *T1* and *T2* validate their read accesses (Y for *T1*, X for *T2*), and find that their read accesses are locked by other transactions. As a result, *T1* and *T2* both abort and restart later, and can never commit.

3.2.3 Miscellany

Memory fences – STM systems in weak memory models (such as GPU memory model) typically need fences to ensure that subsequent accesses are not hoisted [22]. CUDA provides *__threadfence* [17], which stalls current thread until its prior shared memory accesses (global memory accesses) are visible to all threads in the thread block (all threads in the kernel). The fences are used in 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 (line 5). In *TXRead*, two fences are used. The first one is set in between the program data access and consistency checking (line 26). The second

is set in between value-based validation and version checking in $PostValidation$ (line 12). In $TXCommit$, two fences are put before and after memory updates of write-set (line 79 and 82), respectively.

Register checkpoint – GPU-STM does not checkpoint registers currently. Similar to KILO TM [7], we observe that the original values in registers are rarely used when a transaction restarts and do not need to be restored. Only one of our evaluation workload requires restoring one register for each transaction. Other evaluation workloads do not require any register restoration upon transaction aborts. However, if necessary, register restoration can be realized as follows: (1) programmers can insert code to checkpoint and restore register values if a small number of registers need to be restored; or (2) a compiler can determine which registers are both read and written within a transaction and insert code to checkpoint and restore them.

3.3 STM Correctness

Opacity [8] is a crucial correctness criterion for STM systems without sandboxing [4]. It defines a form of strict serializability in which all transactions would appear to occur in some global total order. As stated in [8], opacity captures the following requirements: (1) all operations performed by every committed transaction appear as if they occurred at some single, indivisible point, (2) memory updates made by aborted transactions are not visible to other transactions, and (3) every transaction always observes a consistent view of memory during execution.

We briefly explain how GPU-STM follows such opacity requirements. (1) In GPU-STM, a committed transaction appears as if it occurred at the point when the global clock was increased (line 83). The indivisibility of committed transactions is ensured by using locks: a transaction locks all locations involved before validation and memory updates during commit. (2) In GPU-STM, aborted transactions can never be visible due to lazy updates. (3) GPU-STM checks for consistency (line 27-33) after each transactional read to ensure that every transaction always observe a consistent view of memory.

4. EVALUATION

In this section, we evaluate the performance of GPU-STM from different perspectives. First, we present the overall performance results of five workloads with different characteristics. Second, we compare the proposed hierarchical validation (HV) technique and the timestamp-based validation (TBV) technique in more details using a micro-benchmark. Finally, we evaluate the overhead of STM system by examining the execution time breakdown of a single-thread.

4.1 Experimental Setup

Three micro-benchmarks and three workloads ported from STAMP benchmarks [15] are used in the evaluation. The workloads together exhibit comprehensive transactional characteristics, as presented in Table 1. The micro-benchmarks used are *random array* (RA), *hashtable* (HT), and *EigenBench* (EB) [12]. In RA, each transaction randomly accesses multiple locations of a shared array. In HT, each transaction inserts multiple elements into a shared hash table. The EB micro-benchmark is used only for HV and TBV comparison due to its reconfigurability. The STAMP workloads used are *labyrinth* (LB), *genome* (GN), and *k-means* (KM). We select these three STAMP workloads because they are suited for GPU computing. The data structures of the three workloads can be easily replaced with arrays, which is usually required for GPU computing.

We implement GPU-STM on top of CUDA [17] runtime on a Nvidia 1.15 GHz C2070 Fermi GPU with 14 streaming multiprocessors (448 processing elements). The local and global metadata of GPU-STM are both stored in the global memory (a form of off-chip memory in Nvidia's term). The local metadata is cached at the L1 and L2 levels. The global metadata is only cached at the L2 level, because the L1 cache on Fermi GPUs is not coherent and cannot be used for caching globally shared data. We selectively bypass the L1 cache by using "*volatile*" [17]. Though implemented on the Nvidia GPU and runtime, the design of GPU-STM is relatively general and applies to any similar GPU architecture. We transactify applications using GPU-STM runtime APIs. Figure 1 presents a transactional CPU-host/GPU-kernel code example using GPU-STM. Compiler support can further reduce the complexity of GPU-STM programming: (1) log operations and opacity checking can be automatically inserted, and (2) explicit calls to TXRead/Write can be replaced by simple atomic annotations.

4.2 Overall Performance Comparison

We implement four basic STM variants that adopt NOrec-like [5] value-based validation (STM-VBV), TBV with encounter-time lock-sorting (STM-TBV-Sorting), and HV with encounter-time lock-sorting (STM-HV-Sorting) and with backoff (STM-HV-Backoff), respectively. Both HV and TBV use 1M global version locks. STM-VBV does not need to prevent livelocks, since only a single global lock is used. STM-HV-Backoff adopts a backoff mechanism specific to GPU to prevent livelocks. 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 (inactive). Transactions that have acquired locks successfully perform validation and memory updates in parallel before the failed transactions retry. Livelocking within a warp is avoided by sequentially locking in the second step. However, such sequentially locking could cause a bottleneck during commit and thus degrade performance.

In addition, we implement an optimized STM variant (STM-Optimized), which adaptively makes a selection between HV and TBV at runtime according to the amount of shared data of a STM program. STM-Optimized selects HV when the amount of shared data is larger than that of global version locks. Otherwise, it selects TBV, since false conflicts are rare and value-based validation is unnecessary in this case. For GPU programs, usually the amount of shared data can be easily obtained by counting the elements of arrays before transaction kernels start. To prevent livelocks, STM-Optimized adopts encounter-time lock-sorting.

We also compare the STM variants with the existing blocking GPU STM [3] (STM-EGPGV). The non-blocking one exhibits similar performance, as the limited concurrency poses as major obstacle. We do not compare GPU-STM with KILO TM [7], because KILO TM cannot run on existing GPUs. The STM implementations for CPUs (e.g. NOrec [5], JudoSTM [18] and TL2 [6]) are not used for comparison as they cannot run on GPUs. However, they

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
EB	1M-64M	32	32	1M	High	High-Low

[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.

```
/*This example implements the random array micro-benchmark, each thread executes a transaction. The STM code is marked in bold font.*/

__host__ void randomarray () {    //CPU-host code
    int *d_array; cudaMalloc((void **)&d_array, SIZE * sizeof(int));    // allocate global memory on the GPU
    STM_STARTUP();    // allocate memory space for and initialize global metadata
    randomarray_core <<<BLOCKS, BLOCK_SIZE>>>(d_array);    //call a GPU kernel
    STM_SHUTDOWN();    //free global metadata
    cudaFree(d_array);    // free global memory on the GPU
}

__global__ void randomarray_core (int *array) {    //GPU-kernel code
    //allocate memory for a warp of transactions, executed by a single thread within each warp
    Warp *warp = STM_NEW_WARP();
    done = false;
    while(!done){
        TXBegin(warp);    //each thread initializes a transaction
        for(int i = 1; i < ACTIONS_PER_TX; i++){
            action = generateAction(); index = generateAddr();    //generate random action and address
            if(action == do_read) TXRead(&array[index], warp);
            //if opacity during transaction execution is required, check the opaque flag
            if(!warp->isOpaque[threadIdxInWarp])break;
            if(action == do_write) TXWrite(&array[index], val, warp);
        }
        done = TXCommit(warp);    //each thread commits a transaction
    }
    STM_FREE_WARP(warp);
}
```

Figure 1: A code example using GPU-STM.

can be roughly represented by STM-VBV, STM-HV-Backoff and STM-TBV-Sorting (though lock-sorting is not used in CPU STMs). The STM performance is measured in terms of the speedup on GPU transaction kernels over coarse-grained locking (CGL), which serializes critical sections with a single lock, on GPUs. Fine-grained locking is not implemented due to the complexity and even infeasibility (for RA, HT, and LB).

Figure 2 presents an overview of STM performance across the five workloads. Among all STM implementations, STM-Optimized, which adaptively selects HV or TBV, and adopts encounter-time lock-sorting, is either the fastest or ties to the fastest. The performance of STM-EGPGV is constrained by its limited concurrency. STM-VBV yields undesirable performance on workloads with a large number of transactions (e.g. RA, HT, GN, and KM) due to its limited scalability. STM-TBV-Sorting and STM-HV (both Sorting and Backoff) exhibit noticeable speedup over CGL on 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.

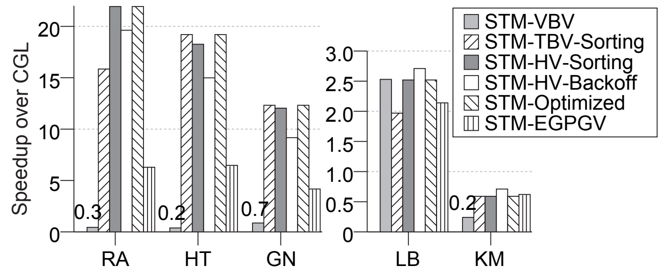

Figure 2: Performance comparison between STM variants and coarse-grained locking on GPU.

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). In addition, STM parallelization is also crucial to the workloads with less transaction time (e.g. LB), since critical sections would have to be serialized otherwise.

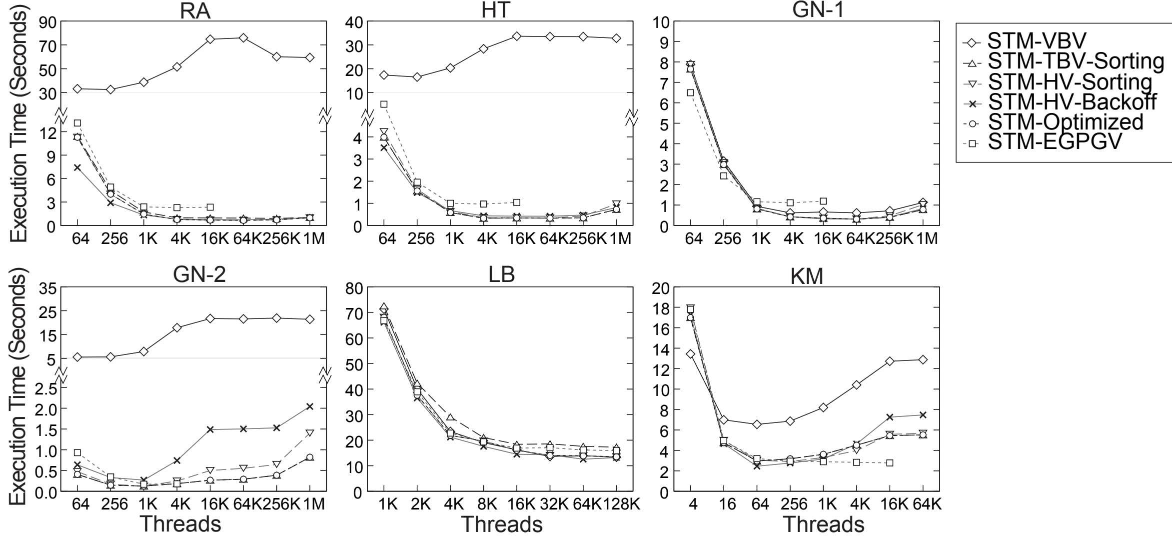

Figure 3: The scalability of STM variants.

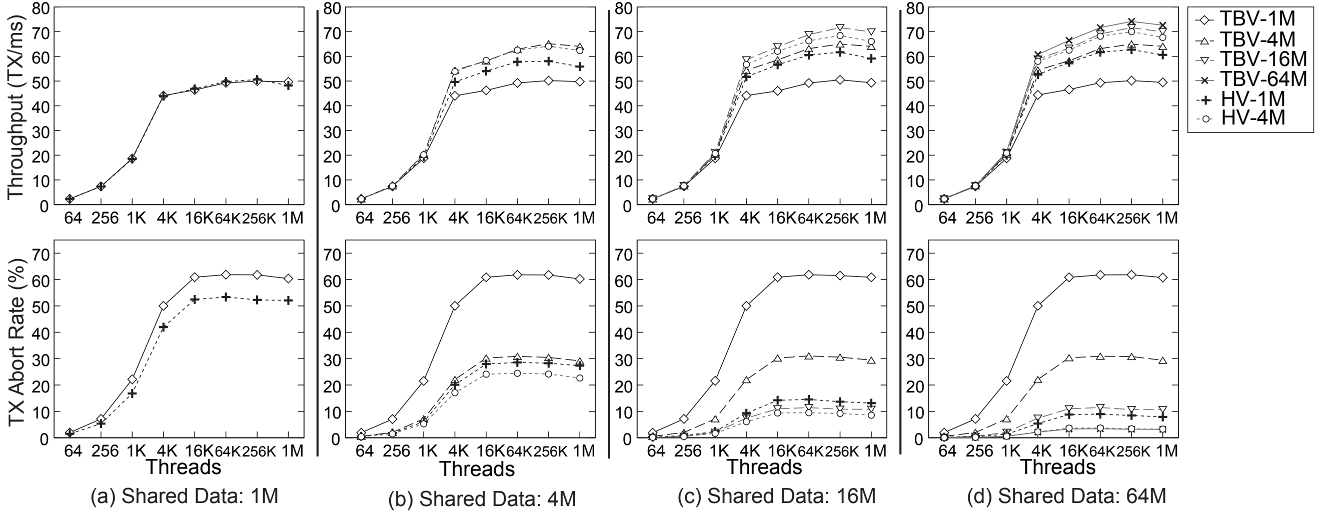

Figure 4: Comparison between HV and TBV with different number of global version locks.

Table 2: Launch configurations of workloads when STM-Optimized achieves optimal performance.

	RA	HT	GN-1, GN-2	LB	KM
Thread-blocks	256	256	256, 16	512	64
Threads per Block	256	256	256, 64	256	4

Furthermore, STM-HV-Sorting outperforms STM-TBV-Sorting on 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 insinuates the TBV implemented by STM-TBV-Sorting suffers from false conflicts. For rest of the workloads, STM-HV-Sorting incurs a small slowdown compared with STM-TBV-Sorting due to unnecessary value-based validation. Compared with the above two variants, STM-Optimized yields better overall performance due to its adaptability. When the amount of shared data is large, STM-Optimized is comparable with STM-HV-Sorting. Otherwise, STM-Optimized is comparable with STM-TBV-Sorting. STM-HV-Sorting outperforms STM-HV-Backoff on workloads with low conflict rates except LB. LB does not benefit from lock-sorting, since there is only one thread that executes transactional code within each thread block, and sorting cannot reduce inactive thread lanes. Also KM does not benefit from lock-sorting, since the inactive thread lanes poses as major obstacle.Therefore, adaptive selection between lock sorting and backoff may yield better overall performance. We leave this as future work.

Table 2 presents the launch configurations of the workloads when STM-Optimized achieves the optimal performance. GN has two transaction kernels, and KM cannot fully utilize the SIMT lanes due to high conflict rate.

Figure 3 presents the scalability of STM implementations in the same workload configurations. Note that we

did not evaluate CGL as it does not scale. STM-EGPGV crashes at relatively small numbers of threads because it does not support per-thread transactions. As expected, STM-VBV does not scale well due to contention on the single global lock it uses. The rest of the STM variants scale quite well, since they use a relatively large number of global locks. Note that the performance does not improve consistently with the increasing number of threads. This is due to the limitation of the hardware resources of GPUs. Meanwhile, the increasing number of threads can result in more conflicts among transactions thus higher abort rates. This is a tradeoff between concurrency and efficiency, and this tradeoff encourages identifying the optimal number of concurrent threads. Therefore, a transaction scheduler that dynamically adjusts concurrency would simplify the optimization of GPU-STM programs. We leave this adaptive transactional scheduler as our future work.

4.3 Comparison between HV and TBV

In this section, we compare HV and TBV in more details from scalability perspective. The evaluation workload used is *EigenBench* (EB) [12] micro-benchmark. Due to its reconfigurability, this micro-benchmark allows us to compare the two validation techniques under different conditions (i.e., the amount of shared data, global version locks and concurrent threads). The amount of shared data in our experiments varies from 1M to 64M.

Figure 4 shows a comparison between HV and TBV with different number of global version locks (from 1M to 64M). HV and TBV yield comparable performance when the number of concurrent threads is small (less than 1K), because the execution path of HV is mostly the same as that of TBV when the conflict is low. HV and TBV also yield comparable performance when the amount of shared data is small (as shown in Figure 4(a)), because exploiting value-based validation (VBV) cannot reduce conflicts. When the amount of shared data is large (as shown in Figures 4(b)-(d)), TBV benefits significantly from increasing the number of global version locks due to the reduced conflicts. Note that HV also benefit from increasing the number of global version locks due to lesser lock contention. More importantly, when many threads concurrently access a large amount of shared data, HV can yield good performance with relatively smaller number of global version locks. As shown in Figures 4(b)-(d), HV yields near optimal performance with 4M locks, since it can exploit VBV to reduce false conflicts.

As shown in Figures 4(b)-(d), HV can reduce the transaction abort rates significantly even with a small number of global version locks. For the EB micro-benchmark, the performance does not benefit significantly from the lower abort rates. Since the threads access global memory most of the time, the long latency of memory accesses hides the benefit of the lower abort rates. However, for real-world workloads, the lower abort rates may yield good performance. Interestingly, as shown in Figure 4(a)-(b), even without false conflicts, TBV results in higher abort rates than HV when they use the same number of global version locks. The reasons are: (1) to reduce lock contention, a transaction is aborted after several lock-acquisition attempts in practical implementations; and (2) HV can filter out conflicts using VBV before acquiring locks to reduce lock contention.

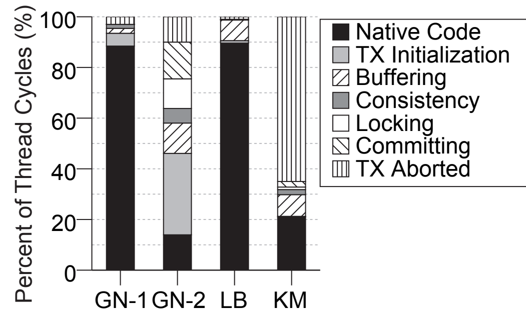

Figure 5: Execution time breakdown of a single-thread, which includes: native-code execution, transaction initialization, buffering, consistency checking, acquiring/releasing locks, committing, and aborted transactions.

4.4 Execution Time Breakdown

This section evaluates the overhead of GPU-STM by examining the execution time breakdown of a single-thread in STM-Optimized, as shown in Figure 5. In the micro-benchmarks (RA, HT and EB), almost all operations of the GPU kernels are synthetic transactional operations. Therefore, the breakdown of these benchmarks cannot tell the overhead of GPU-STM, and is not presented. 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. Therefore, the STM overhead (especially transaction initialization overhead) of GN-2 is difficult to be amortized by the native code execution. Workloads with larger read- and write-sets (e.g. LB and KM) incur higher buffering overhead. Note that the single-thread overhead of GPU-STM can be amortized by the additional scalability it provides. For instance, GN-2 exhibits ∼20x speedup even with significant overhead. However, workloads with limited inherent scalability (e.g. KM) are not suited to GPU-STM, since a very high proportion of transactions are aborted due to high conflict rates.

5. RELATED WORKS

Cederman et al. [3] proposed two STMs on GPUs. The STMs do not incur livelocking issue, because they only support a single transaction per thread block. However, they are not scalable and only allow limited concurrency. Fung et al. proposed KILO TM [7], a HTM 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 cannot be used on existing GPUs, because it requires hardware level modification. Moreover, KILO TM uses several commit units to detect conflicts and order memory updates. The concurrency in commit phase is constrained by the number and capacities of commit units. On the contrary, GPU-STM allows much higher concurrency by using a large number of global version locks. Timestamp-based validation (TBV) was used by TL2 [6], and value-based validation (VBV) was used by JudoSTM [18] and NOrec STM [5]. To ensure the scalability of GPU-STM, we combine TBV and VBV, and propose hierarchical validation (HV), where the interaction between TBV and VBV is carefully designed.

GPU-STM has been presented in our prior work [23]. This paper extends the prior work mainly in two ways. First, we propose a new GPU-STM implementation, which adaptively makes a selection between HV and TBV at runtime based on the amount of shared data of a STM program. Second, we present detailed description and analysis of GPU-STM algorithm, and perform comprehensive evaluation.

6. CONCLUSION

Simplified fine-grained synchronization is crucial for the evolution of the GPU as a massive dynamic data-sharing enabled general-purpose computing architecture. To this end, we propose GPU-STM, a novel STM system for GPU architectures. The proposed STM system can scale to thousands of concurrent threads and ensure livelock-freedom. We build GPU-STM infrastructure on commercially available GPUs and runtime. Our prototype-based evaluation shows that dynamic data sharing among individual threads on GPUs can be efficiently expressed with GPU-STM, and our proposed technique outperforms coarse-grain locks on GPUs significantly.

7. ACKNOWLEDGMENTS

This work was partially supported by 863 Programs of China under grants 2012AA010902 and 2012AA010901, and NSF China under grants 61128004, 61133004 and 61073011. We thank the anonymous reviewers for their valuable comments and advice on improving this paper.

8. REFERENCES

[1] Khronos OpenCL, http://www.khronos.org/opencl/, 2013.

[2] C. Blundell, E. C. Lewis, and M. M. K. Martin. Subtleties of transactional memory atomicity semantics. *IEEE Computer Architecture Letters (CAL)*, 5(2), 2006.

[3] 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 (EGPGV)*, 2010.

[4] L. Dalessandro and M. L. Scott. Sandboxing Transactional Memory. In *Proc. of 21th Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT)*, 2012.

[5] L. Dalessandro, M. F. Spear, and M. L. Scott. NOrec: Streamlining STM by Abolishing Ownership Records. In *Proc. of the 15th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP)*, pages 67–78, 2010.

[6] D. Dice, O. Shalev, and N. Shavit. Transactional Locking II. In *Proc. of the 20th Intl. Symp. on Distributed Computing (DISC)*, pages 194–208, 2006.

[7] W. W. L. Fung, I. Singh, A. Brownsword, and T. M. Aamodt. Hardware Transactional Memory for GPU Architectures. In *Proc. of the 44th Annual IEEE/ACM Intl. Symp. on Microarchitecture (MICRO)*, pages 296–307, 2011.

[8] R. Guerraoui and M. Kapalka. On the Correctness of Transactional Memory. In *Proc. of the 13th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP)*, pages 175–184, 2008.

[9] P. Harish, V. Vineet, and P. J. Narayanan. Large graph algorithms for massively multithreaded architectures. Technical Report IIIT/TR/2009/74, IIIT Hyderabad, INDIA, 2009.

[10] B. He and J. X. Yu. High-throughput transaction executions on graphics processors. In *Proc. of the VLDB Endowment (PVLDB)*, pages 314–325, 2011.

[11] M. Herlihy and J. E. B. Moss. Transactional memory: architectural support for lock-free data structures. In *Proc. 20th Annual Intl. Symp. on Computer Architecture (ISCA)*, pages 289–300, 1993.

[12] S. Hong, T. Oguntebi, J. Casper, N. Bronson, C. Kozyrakis, and K. Olukotun. Eigenbench: A simple exploration tool for orthogonal TM characteristics. In *Proc. of IEEE Intl. Symp. on Workload Characterization (IISWC)*, 2010.

[13] D. B. Lomet. Process structuring, synchronization, and recovery using atomic actions. In *Proc. of the ACM Conference on Language Design for Reliable Software*, pages 128–137, 1977.

[14] M. Mendez-Lojo, M. Burtscher, and K. Pingali. A GPU Implementation of Inclusion-based Points-to Analysis. In *Proc. of the 17th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP)*, pages 107–116, 2012.

[15] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford Transactional Applications for Multi-Processing. In *Proc. of IEEE Intl. Symp. on Workload Characterization (IISWC)*, 2008.

[16] R. Nasre, M. Burtscher, and K. Pingali. Atomic-free Irregular Computations on GPUs. In *Proc. of the 6th Workshop on General Purpose Processor Using Graphics Processing Units (GPGPU)*, pages 96–107, 2013.

[17] NVIDIA CUDA. CUDA C Programming Guide Version 4.2.

[18] M. Olszewski, J. Cutler, and J. G. Steffan. JudoSTM: A Dynamic Binary-Rewriting Approach to Software Transactional Memory. In *Proc. of 16th Intl. Conf. on Parallel Architecture and Compilation Techniques (PACT)*, 2007.

[19] A. Ramamurthy. Towards Scalar Synchronization in SIMT Architectures. Master's thesis, University of British Columbia, 2011.

[20] J. Sanders and E. Kandrot. *CUDA by Example, An Introduction to General Purpose GPU Programming*, chapter 9. Addison-Wesley Professional, 2010.

[21] N. Shavit and D. Touitou. Software transactional memory. In *Proc. of the 14th ACM Symp. on Principles of Distributed Computing (PODC)*, pages 204–213, 1995.

[22] M. F. Spear, M. M. Michael, M. L. Scott, and P. Wu. Reducing Memory Ordering Overheads in Software Transactional Memory. In *Proc. of the 7th annual IEEE/ACM Intl. Symp. on Code Generation and Optimization (CGO)*, pages 13–24, 2009.

[23] Y. Xu, R. Wang, N. Goswami, T. Li, and D. Qian. Software Transactional Memory for GPU Architectures. *IEEE Computer Architecture Letters (CAL)*, 2013.