PR-STM: Priority Rule Based Software Transactions for the GPU

Qi Shen, Craig Sharp(✉), William Blewitt, Gary Ushaw, and Graham Morgan

Newcastle University, Newcastle upon Tyne NE1 7RU, UK
{qi.shen1,craig.sharp,william.blewitt,
gary.ushaw,graham.morgan}@ncl.ac.uk

Abstract. In this paper we describe an implementation of a software transactional memory library for the GPU written in CUDA. We describe the implementation of our transaction mechanism which features both tentative and regular locking along with a contention management policy based on a simple, yet effective, static priority rule called Priority Rule Software Transactional Memory (*PR-STM*). We demonstrate competitive performance results in comparison with existing STMs for both the GPU and CPU. While GPU comparisons have been studied, to the best of our knowledge we are the first to provide results comparing GPU based STMs with a CPU based STM.

Keywords: Transactional memory · GPU · CUDA · Concurrency control · STM

1 Introduction

The availability of Graphics Processing Units (GPU) has recently expanded into the area of general purpose programming, giving rise to a new genre of applications known as General Purpose GPU [10] (hereafter GPGPU). The principle benefit of using the GPU is the relatively high degree of parallel computation available compared to the CPU. Furthermore, programming APIs, such as CUDA [13,14], have grown in sophistication with every new advancement in GPU design. As such, GPGPU programmers now have at their disposal tools to enable them to write complex and expressive applications which can leverage the power of modern GPUs.

As with multi-threaded applications on the CPU, GPGPU applications require synchronisation techniques to prevent corruption of shared data. As has long been experienced in the domain of CPU computing, correctly synchronising multiple threads is a difficult task to implement without introducing errors (such as deadlock and livelock) [7]. To compound matters, the high number of threads available on modern GPUs means that contention for shared data is an issue of greater potential significance than on the CPU where the number of threads is typically much lower.

J.L. Träff et al. (Eds.): Euro-Par 2015, LNCS 9233, pp. 361–372, 2015.
DOI: 10.1007/978-3-662-48096-0_28

To address the difficulties of multi-threading on the CPU, significant progress has been made in providing Concurrency Control techniques to aid the concurrent programmer. One notable technique is Transactional Memory [8] (TM), which allows the execution of transactions in both Software [2,6] and Hardware [8,15]. TM provides an intuitive interface to aid programmers of multi-threaded programs. The TM system guarantees that programs are free of data inconsistency issues while handling the intricacies of thread coordination and contention management.

At the time of writing, implementing an efficient TM technique for the GPU remains an area with much potential for development. The work in this paper aims to contribute to that development by providing the following:

- An STM algorithm for the GPU based on a simple, yet effective, static priority rule. We demonstrate that our technique can out-perform a state-of-the-art STM technique for the GPU called *GPU-STM* [19];
- Benchmarked performance figures are provided, comparing *PR-STM* with both *GPU-STM* and a widely used STM technique for the CPU, namely *TinySTM* [3]. To our knowledge this is the first time that comparisons have been produced between STM techniques for the GPU and the CPU.

We have enhanced the benchmarking software to assess the performance of all three techniques with variation on the number of threads, transaction size and the granularity of lock coverage in addition to the impact of invisible reads.

Section 2 describes the implementation of our STM and Sect. 3 surveys related work. Section 4 describes our evaluation and, finally, Sect. 5 concludes the paper and discusses future work.

2 Implementation

2.1 Overview

The operation of the GPU differs considerably from the CPU and this must be taken into account when implementing transactional algorithms on the GPU. In addition to the high degree of threads available, groups of GPU threads execute as part of a 'warp'. Threads belonging to the same warp share the same instruction counter and thus execute the same instruction in a 'lock-step' fashion. In addition to the risk of high contention given the high number of threads, deadlock and livelock are possible because threads of the same warp cannot coordinate their accesses to locks as they can on the CPU (see Fig. 1(A)).

To prevent the possibility of deadlock and livelock, we use a 'lock stealing' algorithm which requires each thread be assigned a static priority. This allows a thread with priority n to steal a lock which is currently owned by any thread with a priority less than n (see Fig. 1(B)). As every thread has a unique priority, this addresses the possibility of deadlock because any thread can always determine its next action when encountering locked data. Livelock is also addressed as threads will never attempt to perpetually steal one another's locks.

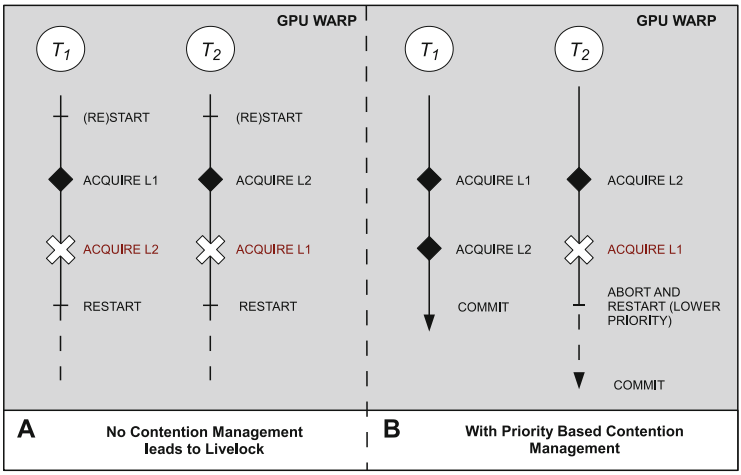

Fig. 1. Livelock and contention management in GPU transaction execution.

PR-STM implements a commit time locking approach where threads attempt to acquire locks at the end of their transactions. Before committing, threads first attempt to validate their transactions by tentatively 'pre-locking' shared data. Pre-locked data can be stolen based on the thread priority rule. If validation is successful the thread may commit its transaction. We implement invisible reads and threads maintain versions of the data they have accessed so that they can abort early if a conflict is detected. This has the benefit of reducing the costs of false conflict where a thread needlessly aborts when encountering data locked by a transaction which itself will abort in future.

2.2 Metadata

PR-STM consists of two types of metadata: a *global* metadata which is shared among all threads and a *local* metadata which is private to a single thread:

- *Global Lock Table.* A lock table is required which should be accessible to all GPU threads, hence it is located in global memory. Each word of shared data is hashable to a unique lock in the *global lock table*. To enhance the scalability of our system we can vary the number of words that are covered by a single lock. When the hashing function has a 1:1 configuration, for instance, every word of shared data has its own lock. While this configuration demands the most memory it minimises the chance of a false conflict based on shared locks. Each entry in the *global lock table* is an unsigned integer composed of version (11 bits), owner (19 bits), locked (1 bit) and pre-locked (1 bit);
- *Local Read Set* is a set of read entries each composed of a memory location, version and value read by the current thread;
- *Local Write Set* is a set of write entries recording the memory location and value written by the current thread;

– *Local Lock Set* is a set of lock indices and lock versions written by the current thread. The use of lock versioning, along with thread priorities provides the data required by our algorithm when a transaction wishes to perform lock stealing.

2.3 STM Operations

PR-STM is comprised of several functions that are executed during significant events during a transaction's execution. Specifically: *txStart*, *txRead*, *txWrite*, *txValidate* and *txCommit*. Algorithms 1 and 2 provide the pseudo code.

txStart is called before a thread begins or restarts a transaction. The function initialises the thread's local read, write and lock sets setting them to be empty (line 1). The thread then sets a local abort flag to false (line 2).

txRead is executed whenever a thread attempts to read shared data from global memory. The calling thread checks if the shared data is locked by another thread (line 3) and if so the thread aborts and restarts its transaction (line 10). If the data is not locked the thread checks to see if the data has already been added to its *local write set* (line 4) and if so, returns the stored value (line 5). If the data is not in the thread's *local write set* it retrieves the value from global memory (line 6) using an atomic read to ensure the value is up to date. The thread then adds the value read to its *local read set* along with the atomically read lock version corresponding to the shared data (lines 7–8) before it is returned.

txWrite records each write a thread wishes to make in its *local write set* . The thread first checks if the data is already locked and if so sets its abort flag to true indicating the transaction must abort and restart when the function returns (line 19). If the data is not locked the thread checks if the data is already in its *local write set* (line 14) and overwrites it. If the data has not been previously written the thread creates a new write set entry (lines 15–18).

txValidate is invoked before the transaction can commit. The thread attempts to lock all shared data that it intends to modify and performs validation of all the shared data it has read. The thread invokes *prelock* on all data read/written (line 20) to determine whether it has the highest priority value. Then the thread validates all the data in its read set by checking that their versions have not changed (lines 21–22). If validation is successful the thread will try to lock all data (line 23). If this is successful then the thread can now commit its transaction. If any of these steps fail, the transaction must abort.

txCommit is invoked only when a transaction has already successfully validated. The thread writes to all global shared data in its *local write set* (line 26) and executes a 'thread fence' (line 27). CUDA provides a thread fence function to ensure memory values modified before the fence can be seen by all other threads. Without a thread fence, the weak memory model of the GPU might cause a

Algorithm 1. PR-STM functions

```
   function txStart()
1      readSet ← writeSet ← lockTable ← ∅;
2      abort ← false;
   function txRead(Address addr)
3      if getLockBit(g_lock[hash(addr)]) = 0 then
4          if < addr, valWritten >∈ writeSet then
5              return valWritten ;
           else
6              value ← atomicRead(addr);
7              version ← getVersion(atomicRead(g_lock[hash(addr)]));
8              readSet ← readSet ∪ {< addr, value, version >};
9              return value;
       else
10         abort ← true;
11         return 0;
   function txWrite(Address addr, Value val)
12     if getLockBit(g_lock[hash(addr)]) = 0 then
13         if < addr, valWritten >∈ writeSet then
14             < addr, valWritten >←< addr, val >;
           else
15             idx ← hash(addr);
16             version ← getVersion(g_lock[idx]);
17             writeSet ← writeSet ∪ {< addr, val >};
18             lockSet ← lockSet ∪ {< idx, version >}};
       else
19         abort ← true;
   function txValidate()
20     if tryPreLock() = true then
21         for all< addr, value, version >∈ readSet do
22             if getVersion(g_lock[hash(addr)]) ≠ version then
                   return false;
23         return tryLock();
       else
24         return false;
   function txCommit()
25     for all < addr, val >∈ writeSet do
26         ∗addr ← val;
27     _threadfence();
28     for all< idx, version >∈ lockSet do
29         if version < maxVersion then
30             setVersion(g_lock[idx], version + 1);
           else
31             setVersion(g_lock[idx], 0);
```

reordering of a thread's instructions, which could lead to inconsistent shared data. The thread fence ensures that modifications to shared data are visible to all threads before any locks are released. The thread then updates the version bit in the *global lock table* for each lock in its lock set. The version bit is either incremented (line 30) or reset (line 31) if the version value has reached the maximum value.

Algorithm 2. PR-STM functions

```
    function tryPreLock()
32      for all< idx, version >∈ lockSet do
33          repeat
34              tmpLockVal ← g_lock[idx];
35              if getVersion(tmpLockVal) ≠ version
36              or getLockBit(tmpLockVal) = 1
37              or(getPreLockBit(tmpLockVal) = 1 and
                getOwner(tmpLockVal) < threadIdx) then
38                  releaseLocks();
39                  return false;
40              preLockVal ← calcPreLockedVal(version, threadIdx);
            until atomicCAS(g_lock+idx,tmpLockVal,preLockVal) = tmpLockVal;
41      return true;
    function tryLock()
42      for all< idx, version >∈ lockSet do
43          PreLockVal ← calcPreLockedVal(version, threadIdx);
44          FinalLockVal ← calcLockedVal(version);
45          if atomicCAS(g_lock+idx,PreLockVal,FinalLockVal) ≠ PreLockVal then
46              releaseLocks();
47              return false;
48      return true;
    function releaseLocks()
        for all idx ∈ PreLocked do
49          preLockVal ← calcPreLockedVal(version, threadIdx);
50          atomicCAS(g_lock+idx,preLockVal,preLockVal-1);
        for all idx ∈ Locked do
51          unLockVal ← calcUnlockVal(version);
52          g_lock[idx] ← unLockVal;
```

2.4 Contention Management Policy

In *PR-STM*, 32-bit memory words are used to represent locks. We use locks for both protecting shared data and implementing our priority rule policy. The various bits of each lock represent the following:

- The first 11 bits of a lock represent the current version of that lock. The version is incremented whenever an update transaction is successfully committed.
- Bits 12–30 represent the priority of whichever thread has currently pre-locked this lock (if such a thread exists). A lower value represents a higher priority.
- The 31st bit indicates whether this lock is pre-locked. Pre-locked locks may be stolen from threads with lower priorities and acquired by threads of higher priorities.
- The last bit represents whether the lock is currently locked. Once this bit is set, no other threads can acquire this lock.

Algorithm 2 (lines 32–52) shows three required handlers which are used to manage the locks:

tryPreLock is called whenever a thread attempts to pre-lock shared data. For each lock in its *local lock set*, the thread checks whether the lock versions are inconsistent (line 35) and whether the lock is unavailable (line 36). Finally, the

thread checks whether the lock has been pre-locked by another thread with a higher priority (line 37). If any of these conditions are true, then the thread releases all locks it has previously pre-locked and aborts (line 39) otherwise the thread attempts to pre-lock the lock using an atomic Compare and Swap (CAS). If the CAS fails then another thread must have accessed the lock. The thread must then repeat lines 35–37 until it aborts or the CAS succeeds and it has the highest priority so far of all the threads attempting to pre lock this lock.

tryLock is called when a thread successfully pre-locks every lock in its *local lock set*. The thread attempts to lock each pre-locked lock (line 45). If any CAS fails then the lock has been stolen by a higher priority thread and the original thread must then release all locks and abort (lines 46–47).

releaseLocks is called when a thread commits or aborts. All pre-locked/locked locks are released. Pre-locked locks must be released by CAS (line 50) in case the lock has been stolen by another thread.

3 Related Work

Although STM research on the GPU is a recent research area at the time of writing, numerous implementations of software transactions for the GPU have been implemented. Cederman et al. [1], for instance, implemented the first STM on the GPU that works at the granularity of a thread-block (rather than the granularity of individual threads). By using a relatively coarse 'thread-block granularity', Cederman's technique avoids dependency violations between threads within a single block. Although this reduces contention due to the typically high thread numbers used on the GPU, it does not accommodate workloads more appropriate for STM execution.

Xu et al. have implemented an approach called GPUSTM [19] which, like *PR-STM*, operates at the granularity of the thread. GPUSTM implements an approach based on a combination of timestamp-based and value-based validation called 'hierarchical based validation'. Their validation technique requires that locks are sorted whenever transactional reading takes place to avoid the possibility of livelock. The static priority rule used by *PR-STM* on the other hand avoids the need to sort locks (our threads are effectively pre-sorted by their priorities instead).

Research has also been explored in providing Hardware Transactional Memory (HTM) for the GPU. In particular, Fung et al. [4,5] proposed a technique using value based validation like Xu's work but required significant modifications to the GPU architecture. Nasre et al. have also described generic modifications to improve the performance of morph algorithms with irregular access patterns [12], and [11] explored GPU techniques to speed up execution by reducing the usage of atomic operations.

More recently, Holey et al. have provided Lightweight Software Transactions for the GPU [9]. Three variations of STM design are described, namely: ESTM (eager), PSTM (pessimistic) and ISTM (invisible reads). ESTM updates shared

memory during transaction execution while updating an undo log to remove those updates upon an abort. PSTM is a simpler version of ESTM which treats reads and writes in the same manner, hence PSTM is more effective where transactions regularly read and write to the same shared data. Like our approach, ISTM can represent invisible reads to reduce conflicts during a transaction. None of Holey's techniques allow for lock stealing based on thread priorities however. While Holey's work compares the performance of their algorithms with the CPU, they employ basic fine-grain and coarse-grain locking benchmarks. To our knowledge neither Holey's work, nor any of the other techniques described compare their performance with an actual STM implementation on the CPU.

4 Evaluation

In this section we present results from a series of benchmarks to demonstrate the performance of our system. We compare the performance of *PR-STM* against a recently developed STM system for the GPU called *GPU-STM* [19] and a widely used STM system for the CPU called *TinySTM* [3]. The tests were carried out on a desktop PC with Nvidia Fermi GPU (GeForce GTX 590) which has 16 SMs, operates at a clock frequency of 1225 MHz and has access to 1.5 GB of GDDR5 memory. All shared data and the *global lock table* are allocated in global memory, while all local meta-data is stored in local memory. The *global lock table* data accessed the L2 cache, while local memory accessed both the L1 and L2 caches. The CPU tests were carried out on 2 × dual-core 3.07 GHz Intel(R) processors with 16 GB of RAM. We used the Windows 7 Operating System. *TinySTM* used the Time Stamp Contention Management Policy [16] with the Eager Write Back configuration (with invisible reads).

The experiments use a benchmark called *bank* which accompanies *TinySTM*. A configurable array of bank accounts represents the shared data from which transactions withdraw and deposit funds. We allocated 10 MB of memory to create roughly 2.5 million accounts. We required many accounts to accommodate the presence of many more threads in the GPU. We found that this number of accounts allowed us to observe the effects of both low and high contention as we varied scenario parameters. We also added several adaptations to the base scenario, most notably the ability to vary the amount of shared data accessed within a transaction (i.e. the number of bank accounts). This allowed us to vary the likelihood of contention caused by longer transactions. We also implemented changes to the hashing function used in all three STM systems so that we could control the amount of shared data covered by a single lock to experiment with the degree of false-sharing. Finally, we included results where the number of threads are increased to observe the contention caused by high numbers of threads featured in GPU applications.

In the following graphs we present results where: (i) all threads perform update transactions (i.e. read and write operations) and (ii) 20 % of the threads in the scenario execute read-only transactions. This was included to observe the impact of invisible reads on the scenario. Each test lasted for 5 s and was executed 10 times with the average results presented.

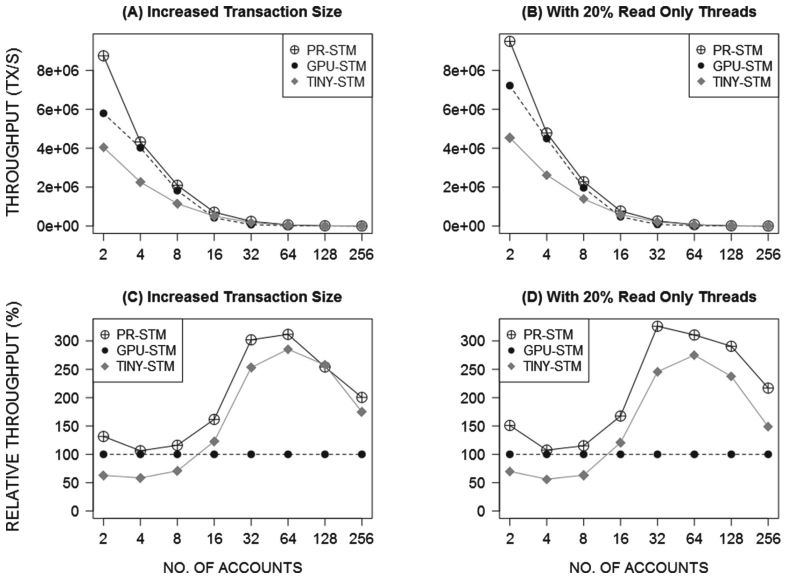

Fig. 2. Average throughput with increasing transaction size

4.1 Transaction Throughput

Figure 2 shows the degree of transaction throughput when the number of accounts accessed per transaction is increased. The number of threads used was kept constant at 512 threads for the GPU and 8 threads for the CPU. These values were used as they provided the best performance in each system. In Figs. 2(A) and (B), Y-axes show the number of transactions committed per second and X-axes show the number of bank accounts accessed in each transaction. As the GPU has many more threads than the CPU both *PR-STM* and *GPU-STM* outperform *TinySTM* when the number of accessed accounts is low (below 16). As expected, when the transaction size increases the throughput of all three STMs drops because inter-transaction conflicts are now more likely. The sharpest drop in performance is witnessed in *GPU-STM* as the higher thread numbers exacerbate the degree of conflicts. In the results with 20 % read only transactions (Figs. 2(B) and (D)) throughput is marginally better. This is because fewer locks are acquired and so fewer conflicts occur.

Figures. 2(C) and (D) show normalised throughput instead of the absolute values shown in Figs. 2(A) and (B). This helps to differentiate the performance when the transaction size increases beyond 16 accounts, where the values are too close to read in absolute terms. Y-axes show the relative throughput of *PR-STM* and *TinySTM* if we treat *GPU-STM* as 100 %. With more accounts accessed we can see both *PR-STM* and *TinySTM* outperform *GPU-STM*. One possible reason for this is that our algorithm does not have to sort the local lock array at every read or write step (like *GPU-STM*) while the higher number of

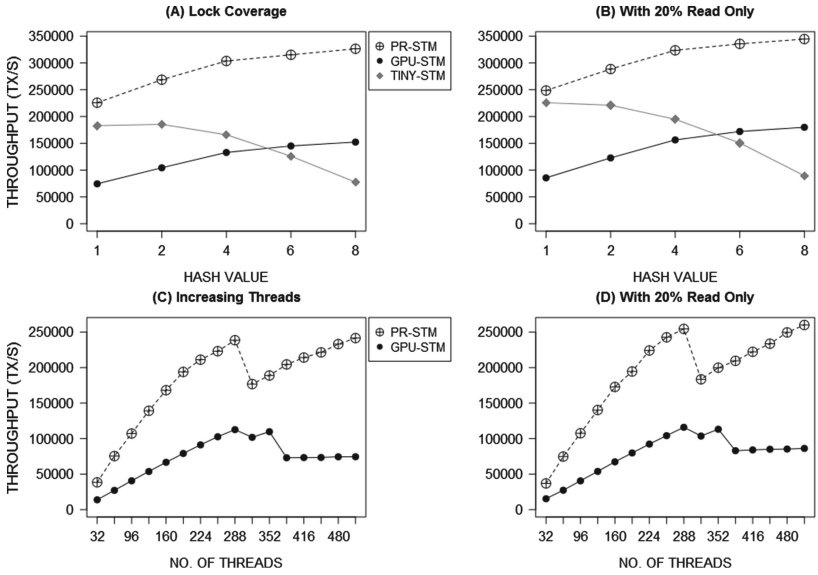

Fig. 3. Average throughput with increasing lock coverage and increased threads

threads enjoyed by *PR-STM* remains a benefit to performance rather than a hindrance.

4.2 STM Scalability

Figures. 3(A) and (B) show the degree of transaction throughput when the hash function is modified. The hash function determines the number of accounts covered by a single lock; the lower the hash value the less chance that threads will try to access the same lock when reading or writing to different shared data. Both the number of threads used and the transaction size were kept constant at 512(GPU)/8(CPU) and 128 respectively. Once again the Y axes show the throughput in transactions per second and the X-axes show the hash function value as the number of accounts covered by a single lock.

Figures 3(A) and (B) provide comparison between *PR-STM*, *GPU-STM* and *TinySTM* with different hash values. As the hash value increases the performance of *TinySTM* deteriorates due to the increased likelihood of false conflicts. Both *PR-STM* and *GPU-STM*, however, show increased throughput. This is because *PR-STM* and *GPU-STM* can both take advantage of reduced lock-querying (due to their lock-sets) and memory coalescing to reduce bus traffic when querying the status of locks held. In Fig. 3(B), with 20 % read only threads, performance is only slightly improved in all three techniques, but mostly in *TinySTM* which gains the most benefit from invisible reads.

In Figs. 3(C) and (D), we increase the number of threads. In these two graphs we only compare the performance of *PR-STM* and *GPU-STM* because

TinySTM is limited by the relatively small number of threads afforded by the CPU. Transaction throughput rises until 258 threads are used where inter-thread conflicts begin to occur at a substantial rate. Below 258 threads, the possibility of conflict is negligible because the high number of accounts used reduces the probability that threads will access the same account. As the number of threads increases, however, so too increases the rate of conflict and therefore the throughput decreases markedly. As thread numbers increase, however, *PR-STM* begins to improve once again, whereas *GPU-STM* levels out. The benefit of the work produced by extra threads is cancelled out by the overhead caused by inter-transactional contention. In Fig. 3(D) we can see that performance improves marginally with the introduction of 20 % read only threads. All other factors being equal, improvements in terms of read only transactions have little effect on the GPU.

5 Conclusion

In this paper we have presented *PR-STM*, a new scalable STM technique for the GPU which uses static thread ranking/priority to efficiently resolve contention for shared locks. We have demonstrated the performance of our approach against both GPU (*GPU-STM*) and CPU (*TinySTM*) software transactional memory libraries which, to our knowledge, is the first time such testing has been done. Results for transactional throughput and scalability demonstrate that our approach performs better than both *GPU-STM* and *TinySTM* in almost all cases.

We believe there exists much scope for expanding our approach. In the short-term we would like to enhance our Contention Management Policy to accommodate dynamic priorities and application semantics (this has been shown to provide substantial performance improvements [17,18]). In the long-term we would like to experiment with combining the GPU and the CPU within a heterogeneous transaction manager. The results suggest that the GPU is particularly effective at processing large numbers of short transactions, while the presence of read-only transactions provides only a small improvement to GPU performance. Further testing will allow us to formulate transaction allocation strategies, assigning work to either the CPU or the GPU based on the effectiveness of each processing element to execute that work.

References

1. Cederman, D., Tsigas, P., Chaudhry, M.T.: Towards a software transactional memory for graphics processors. In: EGPGV, pp. 121–129 (2010)
2. Dice, D., Shalev, O., Shavit, N.: Transactional locking II. In: Dolev, S. (ed.) DISC 2006. LNCS, vol. 4167, pp. 194–208. Springer, Heidelberg (2006)
3. Felber, P., Fetzer, C., Riegel, T.: Dynamic performance tuning of word-based software transactional memory. In: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 237–246. ACM (2008)

4. Fung, W.W., Aamodt, T.M.: Energy efficient GPU transactional memory via space-time optimizations. In: Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture, pp. 408–420. ACM (2013)
5. Fung, W.W., Singh, I., Brownsword, A., Aamodt, T.M.: Hardware transactional memory for GPU architectures. In: Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture, pp. 296–307. ACM (2011)
6. Harris, T., Marlow, S., Peyton-Jones, S., Herlihy, M.: Composable memory transactions. In: Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 48–60. ACM (2005)
7. Herlihy, M.: Wait-free synchronization. ACM Trans. Program. Lang. Syst. (TOPLAS) **13**(1), 124–149 (1991)
8. Herlihy, M., Moss, J.E.B.: Transactional memory: architectural support for lock-free data structures, vol. 21. ACM (1993)
9. Holey, A., Zhai, A.: Lightweight software transactions on GPUs. In: 2014 43rd International Conference on Parallel Processing (ICPP), pp. 461–470. IEEE (2014)
10. Luebke, D., Harris, M., Govindaraju, N., Lefohn, A., Houston, M., Owens, J., Segal, M., Papakipos, M., Buck, I.: GPGPU: general-purpose computation on graphics hardware. In: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing, p. 208. ACM (2006)
11. Nasre, R., Burtscher, M., Pingali, K.: Atomic-free irregular computations on GPUs. In: Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units, pp. 96–107. ACM (2013)
12. Nasre, R., Burtscher, M., Pingali, K.: Morph algorithms on GPUs. In: ACM SIGPLAN Notices, vol. 48, pp. 147–156. ACM (2013)
13. Nvidia, C.: Compute unified device architecture programming guide (2007)
14. Nvidia, C.: Programming guide (2008)
15. Rajwar, R., Goodman, J.R.: Transactional lock-free execution of lock-based programs. ACM SIGOPS Oper. Syst. Rev. **36**(5), 5–17 (2002)
16. Scherer III, W.N., Scott, M.L.: Advanced contention management for dynamic software transactional memory. In: Proceedings of the Twenty-fourth Annual ACM Symposium on Principles of Distributed Computing, pp. 240–248. ACM (2005)
17. Sharp, C., Blewitt, W., Morgan, G.: Resolving semantic conflicts in word based software transactional memory. In: Silva, F., Dutra, I., Santos Costa, V. (eds.) Euro-Par 2014 Parallel Processing. LNCS, vol. 8632, pp. 463–474. Springer, Heidelberg (2014)
18. Sharp, C., Morgan, G.: Hugh: a semantically aware universal construction for transactional memory systems. In: Wolf, F., Mohr, B., an Mey, D. (eds.) Euro-Par 2013. LNCS, vol. 8097, pp. 470–481. Springer, Heidelberg (2013)
19. Xu, Y., Wang, R., Goswami, N., Li, T., Gao, L., Qian, D.: Software transactional memory for GPU architectures. In: Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization, p. 1. ACM (2014)