OFG-STM: Transactional Memory for GPUs based on Obstruction-Free STM algorithms

Tiago Perlin
PPGC - Universidade Federal de Pelotas
Pelotas, Brazil
tiago.perlin@inf.ufpel.edu.br

Gerson Cavalheiro
PPGC - Universidade Federal de Pelotas
Pelotas, Brazil
gerson.cavalheiro@inf.ufpel.edu.br

André Rauber Du Bois
PPGC - Universidade Federal de Pelotas
Pelotas, Brazil
dubois@inf.ufpel.edu.br

Abstract

Transactional memory is a high-level programming abstraction for synchronizing concurrent tasks investigated in different architectures, such as multicores, distributed systems and GPUs. Software transactional memory systems (STMs), including those for GPUs, are usually implemented using a lock-based system, where the locks that protect memory locations being accessed by a transaction are acquired, either at commit time or during transaction execution, to guarantee the atomicity of the transaction's changes to memory. Transactional memory systems can also be lock-free or obstruction-free. Still, such STM algorithms are complex to implement in languages that do not support garbage collection, as they rely on a structure called *locator*, which must be used to acquire ownership of transactional objects. Every time a transaction needs to access an object in write mode, a new locator is created to substitute the current locator of the object. As transactions can be aborted at any time, and concurrent transactions must read the locators of objects to access their current versions, it is difficult to know when exactly such a locator is not being used and can be freed, a common problem in designing lock-free data structures. This paper presents OFG-STM, an STM system for GPUs inspired by obstruction-free STM algorithms. OFG-STM takes advantage of the fact that GPU threads executing the same kernel execute the same code, and threads in a warp execute a kernel in lockstep, to introduce a garbage-collection phase every time a number of transactions has committed. This paper presents the design and implementation of the OFG-STM system, as well as experiments demonstrating its usability.

CCS Concepts: • **Computing methodologies → Parallel programming languages**.

Keywords: transactional memory, GPU, parallel programming

1 Introduction

Transactional memory (TM) is a high-level programming abstraction for synchronizing concurrent accesses to shared-memory. In systems implemented using TM, accesses to shared memory are performed inside of transactions that are guaranteed by the TM system to execute atomically with respect to other concurrent executing transactions. The TM abstraction can be implemented in hardware or software, and has been investigated in different architectures such as CPUs, distributed systems, and GPUs.

Software Transactional Memory (STM) systems for CPUs are usually implemented using lock-based algorithms such as TL2 [6], tinySTM[8] and SwissTM[7]. In such systems, the locks of objects being accessed by transactions must be acquired, either at first encounter or at commit time, in order to guarantee that updates made to objects seem to be atomic. Objects in memory are also tagged with version numbers, which are used to validate data read by transactions: a transaction can only commit if the versions of the values read have not changed during its execution. Some STM algorithms are also called *time-based*, as they use a global shared clock to provide version numbers. In such systems, every memory location is tagged with the time that they were last written, and when a transaction starts, it records its start time from the global clock to guarantee that, during execution, it will only read memory locations that were updated before it started. When a transaction commits, it must first lock all locations to be written (if they are not already locked), then it validates its reads by checking that the versions of the memory locations read have not changed, and finally, it updates the version number of locations to be written before their locks are released. Software Transactional Memory systems for GPUs use algorithms inspired by the CPU ones, so they are also implemented using variations of the lock-based/time-based approach, e.g., [19, 29, 34].

In STM systems for CPUs, the use of a global clock can be a source of contention [1], especially in large systems with frequent commits [25]. Such a problem would be even worse in GPUs, where many threads compete to access the global clock.

Looking at the literature on STM, it is possible to find other techniques for implementing the TM abstraction. Since the objective of TM is to avoid the problems introduced by the use of locks, there are many TM algorithms that are Lock-Free or Obstruction-Free, e.g., [15, 16, 21, 32]. Obstruction-free STM algorithms are complex to implement in languages that do not support garbage collection as they rely on a structure called *locator* (see Section 2.2) that must be used to acquire ownership of transactional objects. Every time a transaction needs to access an object in write mode, a new locator is created to substitute the current locator of the object. As transactions can be aborted at any time, and concurrent transactions must read the locators of objects to access their current versions, it is difficult to know when exactly such a locator is not being used and can be freed, a common problem in designing lock-free data structures [18].

In this paper, we look at obstruction-free STM algorithms as an inspiration for developing an STM system for GPUs. We present OFG-STM, a software transactional memory system for GPUs based on obstruction-free STM algorithms. The main idea of the system is to take advantage of the fact that threads executing a kernel execute the same code, and threads in a warp execute in lockstep, to introduce a garbage collection phase every time several transactions have been committed. This paper describes the design and implementation of OFG-STM, and presents experiments with the current prototype of the system, implemented in CUDA, using the Bank benchmark [23, 29], a benchmark used to evaluate other STM systems for GPUs.

This paper is organized as follows: Section 2 presents some background on transactional memory and obstruction-free STM algorithms. Section 3 describes the main contribution of this paper, the OFG-STM system. Section 4 presents experiments comparing OFG-STM with an implementation of the JVSTM algorithm for GPUs using the Bank Benchmark. Finally, related works (Section 5) and Conclusions (Section 6) are discussed.

2 Background

2.1 Transactional Memory

As multiprocessor systems became more and more available, researchers started to think about new abstractions to simplify concurrent programming. One such abstraction is Transactional Memory (TM) [13, 14], which is a concurrency abstraction that was first proposed as a hardware feature [17] and was investigated on different computing architectures such as CPUs [6–8], distributed systems [24, 26, 31], and GPUs [19, 23, 29, 34]. The main idea is that programmers should access shared memory through transactions, similar to database transactions, and the transactional runtime system should guarantee that these accesses to shared memory appear to be performed atomically concerning other concurrent memory transactions executing in the system.

In an STM system, memory transactions can execute concurrently, and if finished without conflicts, a transaction may commit. Conflict detection may be eager if a conflict is detected the first time a transaction accesses a value, or lazy when it occurs only at commit time. With eager conflict detection, to access a value, a transaction must acquire ownership of the value, preventing other transactions from accessing it, also called pessimistic concurrency control. With optimistic concurrency control, ownership acquisition, and validation occur only when committing. These design options can be combined for different kinds of accesses to data, e.g., eager conflict detection for write operations and lazy for reads. STM systems also differ in the granularity of conflict detection, being the most common word-based and object-based.

STM systems need a mechanism for version management. With eager version management, values are updated directly in memory, and a transaction must maintain an undo log that keeps the original values. If a transaction aborts, it uses the undo log to copy the old values back to memory. With lazy version management, all writes are buffered in a redo log, and readers must consult this log to see earlier writes. If a transaction is committed, it copies these values to memory, and the redo log can be discarded if it aborts.

2.2 Obstruction-Free STM algorithms

In Obstruction-Free STM systems, every transacted memory location is accessed through an abstraction called *transactional object* (TMObject) [16] or *versioned box* [10]. Data that transactions will access must be encapsulated in transactional objects, which, during transaction execution, must be open for reading or writing before their content can be accessed. A transaction typically opens a number of versioned boxes in read/write mode, performs operations on their values, and then commits. As data is accessed only through versioned boxes, the underlying STM system can guarantee consistent and atomic access to it.

Internally, each transactional object points to a *locator* (see Figure 1). Each locator has three fields: the `owner` that points to the state of the transaction that last opened the object in write mode; the `new version`, which is the owner's view of the object; and `old version`, that is the version of the object before the owner opened it. The current version of the object can be determined by the owner's state: if the transaction is COMMITTED, the current version is the new version, and if the transaction is ABORTED or ACTIVE, the current version is in its old version field.

Before a transaction can use a transactional object, it must explicitly open it in write or read mode. When opening a TMObject in write mode, the transaction must acquire ownership of the object by substituting the current locator with a new one. In the new locator, the `owner` field points to the state of the transaction opening the object. The `new version` and `old version` of the new locator contain a copy of the object's current version. A CAS (compare-and-swap) operation is used to substitute the current locator with the new one. If the old owner is ACTIVE, then the transactional system asks a contention manager to decide which transactions should be aborted. A contention manager is a black box that uses some heuristic to decide which transactions to abort [13, 14, 28].

When a transaction opens a TMObject in read mode, it simply consults the object's locator to get its current version and adds the object to its read set. Read sets are used to

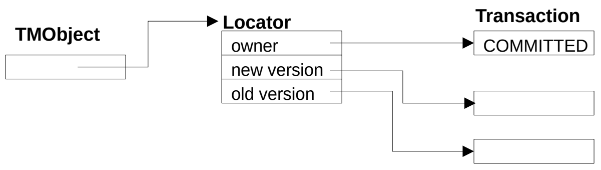


Figure 1. Transactional object and locator

validate the state of the transaction to guarantee that it has seen a consistent state and can commit.

This design makes it easy to update objects atomically at commit time. As all objects opened by a transaction in write mode point to the transaction's state, we just need to atomically change the transaction's status from ACTIVE to COMMITTED using a CAS operation to update the objects. In the same way, if there is a conflict between two transactions, we can abort one of them by atomically changing its state from ACTIVE to ABORTED.

The ideas presented in this Section were first proposed by Herlihy et al. in [16]. Variations of this design were implemented in many STM systems, e.g., [15, 16, 21, 32].

3 OFG-STM: STM for GPUs inspired by Obstruction-Free STM algorithms

The OFG-STM system is inspired by the Obstruction-Free STM algorithms described in Section 2.2. Each GPU thread can execute several transactions, one at a time, and as transactions are garbage collected (see Section 3.4), all transactions executed on the same thread use the same space for metadata. Each transaction has a read-set and a write-set, which are allocated to the thread's shared memory, which is faster than the global memory. The read-set is used for validating reads, i.e., to check if a transaction has seen a consistent view of memory. To ensure that the state of a transaction is always consistent, the read-set is validated each time a new object is opened. Unlike the Obstruction-Free STMs, presented in Section 2.2, transactions also have a write-set. The write-set is used to garbage-collect transactions so that their metadata can be reused by other transactions executing on the same thread (see Section 3.3). As transactions must be able to see the state of other transactions to decide the correct versions of objects, the state of all transactions is kept in an array on global memory. Each transaction has a unique ID that can be used to index the array of states to find its current state. Every thread also has a queue of free locators accessed by transactions whenever they need to open an object in write mode. Since all transactions must view locators, they are allocated on the global memory. The typical workflow of a transaction is first to initialize its metadata, then a number of objects are opened in read (Section 3.2) or write mode (Section 3.1). Once objects are open, the transaction can read and write to them freely with no overhead, and when finished computing, it tries to commit (Section 3.3).

3.1 Opening objects for writing

In Figure 2, we can see the algorithm for opening a transactional object for writing. Initially (lines 3 and 4), the transaction gets the current `locator` of the object to be opened and asks the memory management system for a new locator (`n_locator`) that will be used to substitute the current one. If the current transaction is already the object's owner, we return the object's current version (line 6). Otherwise, the new locator's state depends on the current owner's state. If the current owner is COMMITTED, the current version of the object is in its locator's `new_version` field, so the new locator is initialized accordingly (lines 11 and 12). If the current owner is ABORTED, the current version of the object is in the locator's `old_version` field. If the current owner is ACTIVE (line 18), there is a conflict between the two transactions, so we ask a contention manager (see Section 3.5) to decide which transaction to abort. If the current transaction was not aborted (line 21), it means that the enemy transaction was, so the new locator is initialized with the `old_version` field of the owner. If the current transaction is aborted, the new locator created is useless, so it is returned to the memory management system (line 26). Finally, after the new locator was initialized, `__threadfence()` is called to guarantee that the updates done in the new locator are visible to other transactions (line 31) and we try to substitute the old locator for the new one using a compare and swap operation (line 32). If successful, we validate the read set to guarantee that the transaction still has a consistent view of the memory and return the new version of the object. Otherwise, the transaction is aborted.

3.2 Opening an object for reading

When opening an object for reading, a transaction must check the state of the locator to get the correct version of the object (Figure 3). If the object's current owner has committed, the current version is in the `new_version` field. Otherwise,

```
TX_Open_Write(stm_data, tx_data, object)
{
  locator = get_locator(object)
  n_locator = get_new_locator()
  if (locator->owner == tx_data->tr_id)
     return locator->new_version;
  n_locator->owner = tx_data->tr_id;
  n_locator->object = object;
  switch (get_tr_state(locator->owner)) {
  case COMMITTED:
   n_locator->old_version = locator->new_version;
   n_locator->new_version = n_locator->old_version;
   break;
  case ABORTED:
   n_locator->old_version = locator->old_version;
   n_locator->new_version = n_locator->old_version;
   break;
  case ACTIVE:
   TX_contention_manager(stm_data, tx_data,
   n_locator->owner, locator->owner))
   if(get_tr_state(tx_data->tr_id) != ABORTED)
   {
    n_locator->old_version = locator->old_version;
    n_locator->new_version = n_locator->old_version;
   } else {
    free_locator(n_locator);
   }
   break;
  }
  if(get_tr_state(tx_data->tr_id) != ABORTED)
  { __threadfence();
    if(atomicCAS(stm_data->vboxes[object],
        locator, n_locator))
    {
       if(TX_validate_readset(stm_data, tx_data))
        {  add_write_set(n_locator, object)
           return n_locator->new_version;
        } else {
            abort_tr(stm_data, tx_data->tr_id);}
    } else {
        free_locator(n_locator);
    }
 return NULL;
}
```

Figure 2. Opening an object for writing

```
TX_Open_Read(stm_data, tx_data, object)
{
    locator = get_locator(object)
    switch (get_tr_state(locator->owner)) {
            case COMMITTED:
              version =  locator->new_version;
              break;
            case ABORTED:
              version =  locator->old_version;
              break;
            case ACTIVE:
              version = locator->old_version;
              break;
    }
   if(TX_validate_readset(stm_data, tx_data))
    {
      add_read_set(tx_data, object, locator, version);
      return version;
    }
    abort_tx(tx_data);
    return NULL;
}
```

Figure 3. Opening an object for reading

```
TX_commit(stm_data, tx_data)
{
  __threadfence();
  if(atomicCAS(tr_state(tx_data->tr_id),
               ACTIVE ,
               COMMITTED))
     {
      TX_free_locators(stm_data, tx_data);
      return 1;
     }
  return 0;
}
```

Figure 4. Committing a transaction

if the transaction is active or aborted, the current version is in the old_version field. After discovering the object's current version, the transaction validates its read set to verify that the objects it has read so far have not changed, in which case the transaction has a consistent view of memory and can proceed. Otherwise, the transaction aborts.

3.3 Committing a transaction

The commit operation in obstruction-free STMs is very cheap compared to other lock-based algorithms. As in OFG-STM every time a transaction opens an object in write mode, it has to validate its read set; when a transaction opens its last object for writing, the transaction is in a state where it owns all objects to be written and also has a consistent view of memory, and that is the transactions serialization point in time. Hence, to commit a transaction, we only have to change its state from ACTIVE to COMMITTED, using a compare-and-swap operation (Figure 4). If the transaction is still active, the CAS operation will succeed, and its changes to memory will be automatically available to all other transactions. If the CAS operation fails, it means that the transaction is no longer active and was aborted by other transactions due to conflicts.

To help OFG-STM's garbage-collection (see Section 3.4), when a transaction commits, it must also *free* the locators it has created, which are present in its write-set. This process changes the *owner* field of the locators to point to the state of a dummy transaction, which is always committed. This process guarantees there are no references to the committing transaction, and its memory space can be used to execute another transaction in the same thread.

3.4 Garbage-Collection

Obstruction-Free STMs are challenging to implement in languages with no garbage-collection due to the difficulty in managing locators. When a transaction opens an object in write mode, it must substitute the current locator of the TMObject. Although the old location is now garbage, we can not reuse it as other transactions might have it in their read set. This is a common problem in designing lock-free/obstruction-free algorithms for CPUs [18], and there are many solutions to it, like reference counting [20], hazard pointers [22] and epoch-based reclamation [2]. The main problem with these techniques is that they are heavily based on atomic operations, which are very expensive for GPUs.

In OFG-STM, we take advantage of the fact that on the GPU, threads executing the same kernel execute the same code, and threads in the same warp execute in lockstep, to introduce a garbage-collecting phase during execution. The idea is that threads synchronize to garbage collect their locators after executing several transactions. Synchronization is achived using CUDA'S Cooperative Groups [1]. Each thread in the STM system starts with a fixed number of locators, which can be used during the lifetime of a thread to perform transactions. These locators are stored in a queue, one for each thread. When a transaction calls `get_new_locator()` in the `TX_Open_Write` primitive (see Figure 2), the locator at the top of the queue is returned, and the queue advances. If a transaction is aborted, in some cases, it can return an unused locator to the queue (see lines 26 and 41 of Figure 2).

Locators were extended with an extra field `object` (see Figure 6) that points to the TMObject that the locator protects to allow garbage collection. Garbage collection proceeds by passing through the used locators in the queue of locators, checking if each TMObject pointed by the locators still contains the respective locator. If not, the locator is garbage and can be reused.

3.5 Contention Management

A contention manager (CM) is a black box responsible for deciding which transaction should continue when two transactions conflict. There are only two requisites for contention managers in obstruction-free algorithms [27]: they must be non-blocking, and a transaction must, after a finite number of tries, eventually be granted permission to abort an enemy transaction.

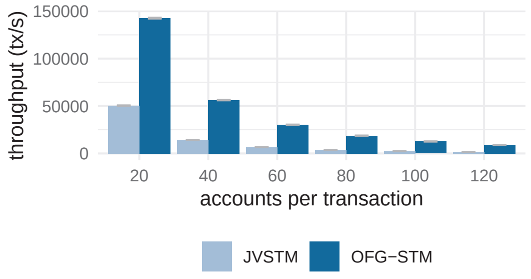
In the current prototype of OFG-STM, we employ a simple contention manager that favors the transaction that has done more work. For that, it compares the size of the write set of both transactions and aborts the transaction with the smaller set. If both write sets are the same size, it aborts the transaction that has been aborted fewer times up until now. There are some contention managers explicitly designed for GPUs, e.g., [29, 30], and we plan, in the future, to investigate how these techniques could be applied in the OFG-STM system.

[1] https://developer.nvidia.com/blog/cooperative-groups/

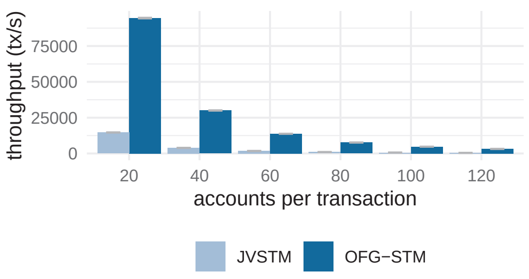
4 Experiments

To evaluate the current prototype of OFG-STM, written in CUDA, we have implemented the Bank benchmark, a benchmark used evaluate transactional memory for GPUs in other works [23, 29]. The bank benchmark simulates a bank that manages a number of accounts, that begin with an initial balance, and also provides two operations: (i) transfer, which transfers an amount of money between two accounts, and (ii) balance, which reads the balance of an account. To stress the system, in all experiments presented, OFG-STM each thread performs garbage collection after every transaction committed. Also, each thread starts with 4.500 locators to be used by its transactions. This value was obtained by executing the worst case scenarios and counting how many locators were needed. The reader should notice that, to execute 1024 threads, the total size of the global memory occupied by locators is approximately 27.5 Megabytes. The experiments were carried out on an AMD Ryzen 7 5700X 8-Core Processor, with an NVIDIA GP102 TITAN Xp, 12 GB, 3840 cores, running Ubuntu Server Linux kernel 6.5.0-28, and nvcc version V12.0.140. As a baseline for comparison, we are using an implementation of the JVSTM transactional memory algorithm [10] modified for GPU execution, taken from [23]. The JVSTM algorithm was designed for applications in which most transactions are read-only and uses locks at commit time.

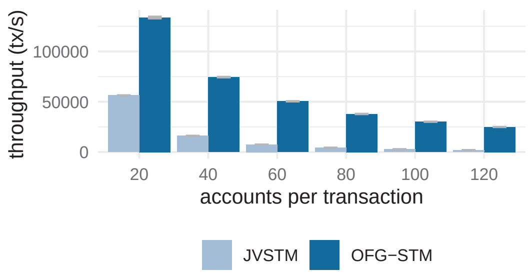
The application is executed for 5 seconds for each case scenario, and then the throughput, given in commits per second, is computed. Each application instance was executed 30 times, and the average throughput is used in the graphs. In the experiments of Figure 5, 1024 threads are concurrently executing transactions that modify the state of the bank in a grid comprised of 32 blocks with 32 threads. The experiment is executed in two scenarios, a high-contention one with 100K bank accounts and a low-contention scenario with 10M accounts. The experiment shows how throughput varies when we increase the number of bank accounts accessed per transaction. As expected, it is possible to see that when contention rises, i.e., each transaction accesses more objects, the throughput decreases. The reader should notice that, with 100K bank accounts, when transactions access 120 objects in the worst-case scenario, the total number of accounts concurrently accessed by transactions is higher than the



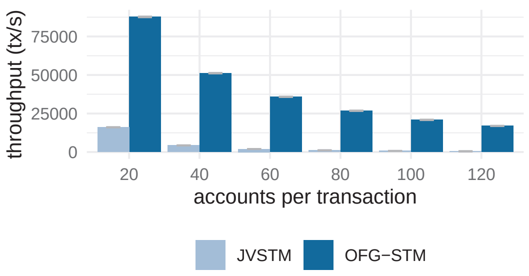
(a) 100,000 accounts and 80% read-only transactions



(b) 100,000 accounts and 20% read-only transactions



(c) 10,0000,000 accounts and 80% read-only transactions



(d) 10,0000,000 accounts and 20% read-only transactions

Figure 5. Throughput of the bank application varying the number of objects accessed by each transaction.

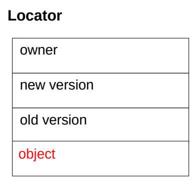


Figure 6. New locator structure

total number of bank accounts. In lower contention, both systems present higher throughput.

In the experiments of Figure 7, the number of bank accounts accessed by each transaction was fixed at 30, and we varied the number of transactions executed concurrently using the high and low-contention scenarios. In general, when the number of concurrent threads increases, also the throughput increases.

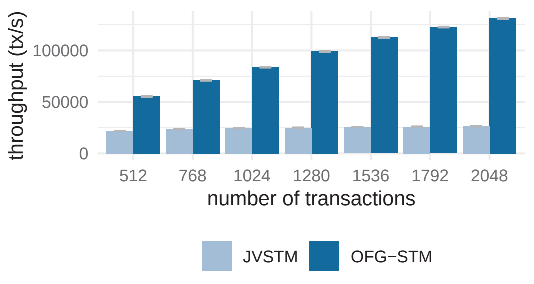
As the JVSTM algorithm was designed for read-dominant scenarios, in the last experiment, we wanted to compare both systems when we increased the percentage of read-only transactions. The experiments were carried out with 1024 transactions, each accessing 30 bank accounts, and the results are presented in Figure 8. As it is possible to see in the graphs, the difference between OFG-STM and JVSTM decreases as we increase the number of read-only transactions, but JVSTM is faster than OFG-STM only with 100% read-only transactions.

5 Related Work

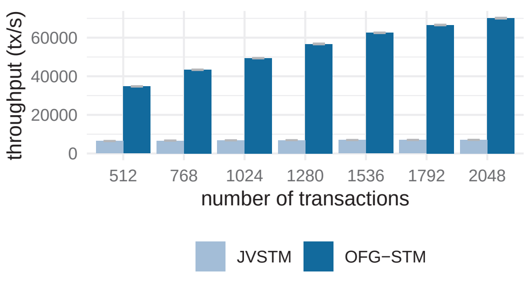
The first STM system for GPUs was [3], which was limited because it only allowed one transaction for each thread block. The main problem with the approach is that it can not deal with the large workloads expected in a GPU.

GPUSTM [34] is a lock-based STM system for GPUs. It combines timestamp-based and value-based validation, which they call *hierarchical-based validation*. The system is inspired by STM algorithms proposed initially for the CPU [6, 9].

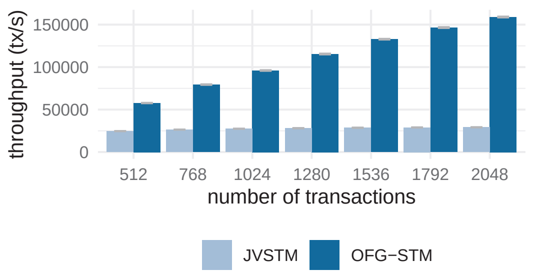
PR-STM [29] implements a typical lock-based STM system for GPUs that uses version numbers to validate reads. Each transactional value in memory contains a version number



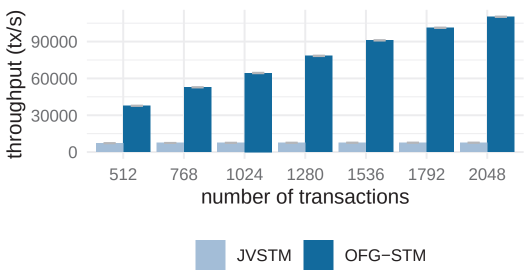
(a) 100,000 accounts and 80% read only transactions



(b) 100,000 accounts and 20% read only transactions

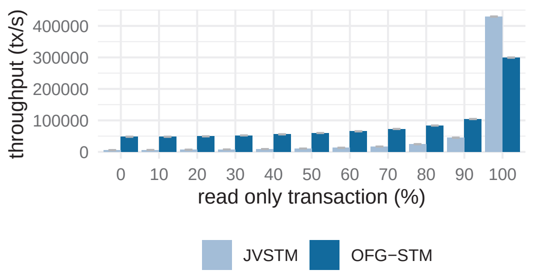


(c) 10,0000,000 accounts and 80% read only transactions

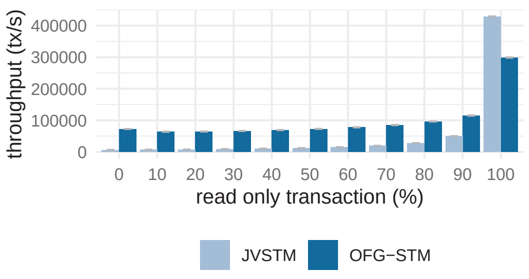


(d) 10,0000,000 accounts and 20% read only transactions

Figure 7. Throughput of the bank application varying the number of threads.



(a) 100,000 accounts



(b) 10,0000,000 accounts

Figure 8. Throughput of the bank application varying the probability of executing read-only transactions.

that is updated every time a committing transaction modifies the value. Its main difference compared to STMs for the CPU is that it uses a priority-based contention manager. Each transaction has a different priority, and when a conflict occurs, the transaction with a higher priority is always favored. It uses a pre-locking scheme where transactions register their intention to lock an object, and locks can be stolen by higher-priority transactions. Also, a contention manager designed for the system was investigated in [30].

In [19], the authors present the Lightweight Software Transactions system for the GPU. The paper describes three different variations of lock-based STM systems for the GPU. ESTM is an eager system in which transactions acquire locks of memory locations to be written at the first encounter. PSTM treats reads in the same way as writes, acquiring locks to secure that values read by a transaction are not modified. ISTM makes reads invisible to other transactions so that conflicts are reduced.

In [23], the authors present different implementations of the JVSTM [10] transactional memory algorithm for the GPU. The main idea of the JVSTM algorithm is to maintain a linked list of older versions of an object at each transactional object so that read-only transactions are never aborted. It uses timestamps for version numbers, and a transaction uses its starting time to know which versions of objects it can access. In the paper, different versions of the system were presented, one that uses locks at commit time (which was used in the experiments of Section 4) and a client-server variation of the algorithm that uses the system in [33] for message-passing. Although the proposed scheme is good for read-only transactions, it does not scale well when the number of updated transactions increases. In the experiments presented in [23], transactions would update a maximum of two memory locations.

The works mentioned in this section are all software-based approaches for transactional memory on GPUs. There are also several hardware based TMs for GPUs, e.g., [4, 5, 11, 12]. These systems try to avoid overheads imposed by STMs through special-purpose hardware. The drawback of these approaches is that they can not be executed on existing GPUs.

6 Conclusions and Future Work

This paper presented OFG-STM, a transactional memory system for GPUs inspired by obstruction-free STM algorithms. The main difficulty in implementing obstruction-free STM algorithms in languages with no automatic garbage collection is managing locators, a structure used to acquire ownership of objects. When transactions open objects in write mode, they must substitute the locator of the object being opened by a new one. Different transactions may point to a locator, so it is difficult to know when they are garbage. To solve this problem, we take advantage of the fact that all threads executing a kernel execute the same code, some of them even in lockstep, to introduce a garbage collection phase where each thread collects its own locators.

As future work, we would like to investigate other contention management policies, including those specifically designed for GPUs, e.g., [29, 30], in conjunction with OFG-STM. Contention management plays an important role in obstruction-free STM algorithms [14], and we believe that OFG-STM could be improved with CM policies specially designed for it. Furthermore, we would like to implement other applications using OFG-STM, as for example, Memcached [23] and also we would like to compare OFG-STM with other STM systems for GPU.

Acknowledgments

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil (CAPES) – Finance Code 001.

References

[1] Ehsan Atoofian and Amir Ghanbari Bavarsad. 2012. AGC: adaptive global clock in software transactional memory. In *Proceedings of the 2012 International Workshop on Programming Models and Applications for Multicores and Manycores* (New Orleans, Louisiana) *(PMAM '12)*. Association for Computing Machinery, New York, NY, USA, 11–16. https://doi.org/10.1145/2141702.2141704

[2] Trevor Alexander Brown. 2015. Reclaiming Memory for Lock-Free Data Structures: There has to be a Better Way. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing* (Donostia-San Sebastián, Spain) *(PODC '15)*. Association for Computing Machinery, New York, NY, USA, 261–270. https://doi.org/10.1145/2767386.2767436

[3] Daniel Cederman, Philippas Tsigas, and Muhammad Tayyab Chaudhry. 2010. Towards a Software Transactional Memory for Graphics Processors . In *Eurographics Symposium on Parallel Graphics and Visualization*, James Ahrens, Kurt Debattista, and Renato Pajarola (Eds.). The Eurographics Association. https://doi.org/10.2312/EGPGV/EGPGV10/121-129

[4] Sui Chen and Lu Peng. 2016. Efficient GPU hardware transactional memory through early conflict resolution. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 274–284. https://doi.org/10.1109/HPCA.2016.7446071

[5] Sui Chen, Lu Peng, and Samuel Irving. 2017. Accelerating GPU hardware transactional memory with snapshot isolation. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. 282–294. https://doi.org/10.1145/3079856.3080204

[6] Dave Dice, Ori Shalev, and Nir Shavit. 2006. Transactional Locking II. In *Distributed Computing*, Shlomi Dolev (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 194–208.

[7] Aleksandar Dragojević, Rachid Guerraoui, and Michal Kapalka. 2009. Stretching transactional memory. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Dublin, Ireland) *(PLDI '09)*. Association for Computing Machinery, New York, NY, USA, 155–165. https://doi.org/10.1145/1542476.1542494

[8] Pascal Felber, Christof Fetzer, and Torvald Riegel. 2008. Dynamic Performance Tuning of Word-Based Software Transactional Memory. *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP* (02 2008). https://doi.org/10.1145/1345206.1345241

[9] Pascal Felber, Christof Fetzer, and Torvald Riegel. 2008. Dynamic Performance Tuning of Word-Based Software Transactional Memory. *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP* (02 2008). https://doi.org/10.1145/1345206.1345241

[10] Sérgio Miguel Fernandes and João Cachopo. 2011. Lock-free and scalable multi-version software transactional memory. *SIGPLAN Not.* 46, 8 (feb 2011), 179–188. https://doi.org/10.1145/2038037.1941579

[11] Wilson W. L. Fung and Tor M. Aamodt. 2013. Energy efficient GPU transactional memory via space-time optimizations. In *2013 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 408–420.

[12] Wilson W. L. Fung, Inderpreet Singh, Andrew Brownsword, and Tor M. Aamodt. 2011. Hardware transactional memory for GPU architectures. In *2011 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 296–307.

[13] Rachid Guerraoui and Paolo Romano (Eds.). 2015. *Transactional Memory. Foundations, Algorithms, Tools, and Applications*. LNCS, Vol. 8913. Springer.

[14] Tim Harris, James Larus, and Ravi Rajwar. 2010. *Transactional Memory, 2nd edition.* Morgan and Claypool Publishers.

[15] Maurice Herlihy, Victor Luchangco, and Mark Moir. 2006. A Flexible Framework for Implementing Software Transactional Memory. In *21st OOPSLA*. ACM, 253–262.

[16] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer. 2003. Software Transactional Memory for Dynamic-Sized Data Structures *(PODC '03)*. ACM, 92–101.

[17] Maurice Herlihy and J. Eliot B. Moss. 1993. Transactional memory: architectural support for lock-free data structures. *SIGARCH Comput. Archit. News* 21, 2 (may 1993), 289–300. https://doi.org/10.1145/173682.165164

[18] Maurice Herlihy, Nir Shavit, Victor Luchangco, and Michael Spear. 2020. *The Art of Multiprocessor Programming.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[19] Anup Holey and Antonia Zhai. 2014. Lightweight Software Transactions on GPUs. In *2014 43rd International Conference on Parallel Processing*. 461–470. https://doi.org/10.1109/ICPP.2014.55

[20] Richard Jones, Antony Hosking, and Eliot Moss. 2023. *The garbage collection handbook: the art of automatic memory management.* CRC Press.

[21] Virendra J. Marathe, William N. Scherer, and Michael L. Scott. 2005. Adaptive Software Transactional Memory. In *DISC'05*.

[22] Maged M Michael. 2004. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Transactions on Parallel and Distributed Systems* 15, 6 (2004), 491–504.

[23] Diogo Nunes, Daniel Castro, and Paolo Romano. 2022. CSMV: A Highly Scalable Multi-Versioned Software Transactional Memory for GPUs. In *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 526–536. https://doi.org/10.1109/IPDPS53621.2022.00057

[24] Jerônimo Ramos, Andre Rauber Du Bois, and Gerson Cavalheiro. 2023. Obstruction-Free Distributed Transactional Memory. In *Proceedings of the XXVII Brazilian Symposium on Programming Languages* (, Campo Grande, MS, Brazil,) *(SBLP '23)*. Association for Computing Machinery, New York, NY, USA, 33–40. https://doi.org/10.1145/3624309.3624316

[25] Torvald Riegel, Christof Fetzer, and Pascal Felber. 2007. Time-based transactional memory with scalable time bases. In *Proceedings of the Nineteenth Annual ACM Symposium on Parallel Algorithms and Architectures* (San Diego, California, USA) *(SPAA '07)*. Association for Computing Machinery, New York, NY, USA, 221–228. https://doi.org/10.1145/1248377.1248415

[26] Mohamed M. Saad and Binoy Ravindran. 2012. Transactional Forwarding: Supporting Highly-Concurrent STM in Asynchronous Distributed Systems. In *IEEE 24th SBAC-PAD*. 219–226. https://doi.org/10.1109/SBAC-PAD.2012.36

[27] William Scherer and Michael Scott. 2005. Contention management in dynamic software transactional memory. In *Proceedings of PODC'05*.

[28] William N. Scherer and Michael L. Scott. 2005. Advanced contention management for dynamic software transactional memory. In *Proceedings of the Twenty-Fourth Annual ACM Symposium on Principles of Distributed Computing* (Las Vegas, NV, USA) *(PODC '05)*. Association for Computing Machinery, New York, NY, USA, 240–248. https://doi.org/10.1145/1073814.1073861

[29] Qi Shen, Craig Sharp, William Blewitt, Gary Ushaw, and Graham Morgan. 2015. PR-STM: Priority Rule Based Software Transactions for the GPU, Vol. 9233. 361–372. https://doi.org/10.1007/978-3-662-48096-0_28

[30] Qi Shen, Craig Sharp, Richard Davison, Gary Ushaw, Rajiv Ranjan, Albert Y. Zomaya, and Graham Morgan. 2020. A general purpose contention manager for software transactions on the GPU. *J. Parallel Distrib. Comput.* 139, C (may 2020), 1–17. https://doi.org/10.1016/j.jpdc.2019.12.018

[31] Konrad Siek and Paweł T. Wojciechowski. 2016. Atomic RMI: A Distributed Transactional Memory Framework. *Int. Journal of Parallel Programming* 44, 3 (01 Jun 2016), 598–619.

[32] Michael F. Spear, Virendra J. Marathe, William N. Scherer, and Michael L. Scott. 2006. Conflict Detection and Validation Strategies for Software Transactional Memory. In *DISC*.

[33] Kai Wang, Don Fussell, and Calvin Lin. 2019. Fast Fine-Grained Global Synchronization on GPUs. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 793–806. https://doi.org/10.1145/3297858.3304055

[34] Yunlong Xu, Rui Wang, Nilanjan Goswami, Tao Li, Lan Gao, and Depei Qian. 2014. Software Transactional Memory for GPU Architectures. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization* (Orlando, FL, USA) *(CGO '14)*. Association for Computing Machinery, New York, NY, USA, 1–10. https://doi.org/10.1145/2581122.2544139