Improvements in Hardware Transactional Memory for GPU Architectures

Alejandro Villegas, Angeles Navarro, Rafael Asenjo, and Oscar Plata

Dept. Computer Architecture
University of Malaga, Andalucia Tech, 29071 Malaga, Spain
{avillegas,angeles,asenjo,oscar}@ac.uma.es

Abstract. In the multi-core CPU world, transactional memory (TM) has emerged as an alternative to lock-based programming for thread synchronization. Recent research proposes the use of TM in GPU architectures, where a high number of computing threads, organized in SIMT fashion, requires an effective synchronization method. In contrast to CPUs, GPUs offer two memory spaces: global memory and local memory. The local memory space serves as a shared scratch-pad for a subset of the computing threads, and it is used by programmers to speed-up their applications thanks to its low latency. Prior work from the authors proposed a lightweight hardware TM (HTM) support based in the local memory, modifying the SIMT execution model and adding a conflict detection mechanism. An efficient implementation of these features is key in order to provide an effective synchronization mechanism at the local memory level.

After a quick description of the main features of our HTM design for GPU local memory, in this work we gather together a number of proposals designed with the aim of improving those mechanisms with high impact on performance. Firstly, the SIMT execution model is modified to increase the parallelism of the application when transactions must be serialized in order to make forward progress. Secondly, the conflict detection mechanism is optimized depending on application characteristics, such us the read/write sets, the probability of conflict between transactions and the existence of read-only transactions. As these features can be present in hardware simultaneously, it is a task of the compiler and runtime to determine which ones are more important for a given application. This work includes a discussion on the analysis to be done in order to choose the best configuration solution.

Keywords: Transactional Memory, GPU Architecture

1 Introduction

Transactional Memory (TM) [10] has emerged as a promising alternative to locking mechanisms to coordinate concurrent threads. TM provides the concept of a transaction to *wrap* a critical section. A transaction enforces atomicity and isolation during the execution of a critical section. Transactions are allowed to run concurrently, tracking all memory accesses into a read-set and a write-set. If two concurrent transactions conflict (i.e., they access the same memory position and, at least, one of the accesses is a write), one of them must abort, restoring its initial state and retrying its execution. When a transaction concludes its execution without conflicts, it commits, making definitive its changes in memory. Many TM systems have been proposed in the last two decades for multi-core CPU architectures [9].

Graphics Processing Units (GPUs) have been widely adopted as hardware accelerators for data-parallel applications. In contrast to CPUs, GPUs are organized as a set of highly multi-threaded SIMT cores. Using OpenCL [12] terminology, a SIMT core is referred to as a *Compute Unit* (CU). Computing threads are called *work-items*, and are grouped into *work-groups*, that are scheduled to run on the CUs. In addition, work-items have access to two different memory spaces. The *global memory* space, located off-chip, is addressable by all the work-items running on the GPU, and provides high capacity, though with high latency. The *local memory* space is located on-chip inside each CU, featuring small capacity, but also lower latency (as compared to global memory). Local memory is shared by work-items belonging to the same work-group and is used by programmers as scratch-pad to speedup their applications.

Applications resort to explicit synchronization to avoid races when accessing shared data. Supporting efficient mutual exclusion using traditional lock-based implementations in a SIMT architecture poses a challenge. The left side of Figure 1 shows an implementation of a coarse-grained lock in the SIMT programming model, which can become much more complicated when adopting fine-grained locks. To alleviate the burden of lock-based programming and to improve program performance, the use of TM has been proposed for GPU architectures both in software [2, 15, 11] and hardware [8, 7, 14]. In order to implement TM on the scratch-pad memory, previous work [14] proposed modifications to the SIMT execution model and a novel conflict detection mechanism based in signatures. Firstly, the SIMT execution model allows for running transactions concurrently and to restart the conflicting ones. According to this model, transactions run in lockstep and retrying conflicting transactions may result in a deadlock. To avoid this situation and ensure forward progress, a serialization mechanism is

```
bool done = false;
while (!done){
  if (getLock()){
    //Critical Section
    done = true;
    releaseLock();}
}
```

```
TX_Begin();
  //Critical Section
TX_Commit();
```

Fig. 1. Coarse-grained lock implementation in the SIMT programming model (left) and the required transformation for the use of TM (right).

proposed. If this mechanism is required very often it may harm performance. Secondly, the conflict detection mechanism is based on unified read-write signatures (i.e., a single signature to represent both the read-set and write-set). This design decision, despite its simplicity, has some disadvantages. For instance, signatures may introduce false positives which, in some cases, results in the unnecessary re-execution of transactions. In addition, for applications where the read-set differs from the write-set we may find better performance using dedicated signatures for each set instead of using a unified signature.

In this paper, we analyze in detail the aforementioned SIMT execution mode and conflict detection mechanism from previous work on TM for GPU local memory. In addition, we compile different proposals for improving both implementations. During the evaluation of these proposals, we highlight the characteristics of the applications that allows for achieving better performance. These proposals are built in incrementally, which means that they can coexist implemented in hardware, but those that are not efficient for a given application can be deactivated via software or the compiler. In order to help the compiler and runtime design, we provide a discussion on the TM features that should be enabled for a given application with the objective of improving performance.

2 Background

2.1 GPU-LocalTM

GPU-LocalTM [14] is a hardware TM for GPU architectures that allows for explicit synchronization of work-items through local memory using transactions. GPU-LocalTM is designed on top of an AMD's Southern Islands GPU and evaluated using the Multi2Sim 4.2 [13] simulation framework. The Southern Islands' ISA is extended with two new instructions that mark the boundaries of the transaction: TX_Begin and TX_Commit. Local memory operations performed within both instructions are considered transactional. Conflicts are detected as soon as the memory access is performed (eager conflict detection). A successful write access performs a back-up of the old value, which has to be restored in case a conflict is detected and the transaction aborts (eager version management). GPU-LocalTM follows a *requester loses* policy (i.e., the work-item who detects the conflict aborts its own execution). In order to store the old and speculative values, GPU-LocalTM uses existing memory resources. Specifically, in each local memory bank, memory is allocated to store the backed-up value and the work-item identifier for each local memory variable. The memory allocated for this purpose is called *shadow memory.* We discuss the transactional SIMT execution model and the conflict detection mechanism proposed in GPU-LocalTM, as they are relevant for the purposes of this paper.

Transactional SIMT execution model. The Southern Islands' SIMT execution model is based the existence of execution masks managed by hardware and compiler. These masks, available per wavefront, are 64-bit width as there are 64 work-items per wavefront in the current architecture. The *execution mask* (EXEC) indicates, the work-items that are running (those whose bits are set to 1) or disabled (those whose bits are set to 0). The *vector comparison mask* (VCC), stores the results of comparison operations, with a behavior similar to a "vectorized" Z flag. GPU-LocalTM adds the *Transaction Conflict Mask* (TCM), completely managed by hardware, to determine which work-items have a conflict when running a transaction. If a work-item detects a conflict, its corresponding bit in TCM is set to 1. The work-item remains inactive for the rest of the transaction. Once the transaction reaches the TX_Commit instruction, work-items whose bit in TCM is set to 1 (i.e., have detected a conflict during the transaction) must restart the execution from the TX_Begin instruction. Work-items that successfully completed the transaction must wait until the whole wavefront has finished the execution.

As they execute in lockstep, two work-items within the same wavefront may be conflicting after each retry. This deadlock situation can be detected if, after two consecutive retries, the same TCM bits are active. To solve this situation and ensure forward progress, GPU-LocalTM triggers the *wavefront serialization mode*. In this case, a single work-item within the wavefront is chosen to re-execute the transaction while the remaining ones wait for the next retry. This is done by changing only one of the bits in TCM from 1 to 0 when the transaction retries. The policy implemented by GPU-LocalTM is to choose the work-item with lower identifier among the conflicting ones. This mechanism ensures that the work-item selected for execution does not conflict with any other work-items within the wavefront. Table 1 shows an example of the execution of a transaction that requires the use of the wavefront serialization mode.

If, after executing in wavefront serialization mode, the TCM mask still remains the same at the end of the transaction it means that the deadlock is produced by actions of another wavefront. In this case, the *work-group serialization mode* is triggered: the wavefront that detected the deadlock restarts in the wavefront serialization mode and the remaining wavefronts within the work-group must self-abort the ongoing transactions and quiesce in the TX_Begin instruction until the running one finishes its execution.

Conflict detection mechanism. The conflict detection mechanism proposed in GPU-LocalTM is distributed per memory bank and makes use of signatures [1]. A shared signature is used to represent both read-set and write-set for each work-item. The Southern Islands architecture features 32 memory banks for local memory. Non-coalesced memory accesses are serialized by a coalescing unit, which ensures that a bank is accessed by a single work-item at a given time. When accessing local memory within the boundaries of a transaction, GPU-LocalTM triggers the conflict detection mechanism in parallel in each memory bank, which works in 3 phases:

1. Fast conflict detection: in each bank, we determine in parallel if the access to memory causes a conflict with a previous access to the same memory bank. We use signatures, which can cause false positives.

Instruction	EXEC	TCM	TCM OLD	Mode	Comments
TX_Begin	1111	0000	-	TX	Transaction starts
Memory Access	1111	1100	-	TX	Conflict detected between work-items 0 and 1
TX_Commit	1111	1100	-	TX	Transaction finished. Work-items 2 and 3 successfully commit.
TX_Begin	1100	0000	1100	TX	Transaction retries for work-items 0 and 1
Memory Access	1100	1100	1100	TX	Conflict detected between work-items 0 and 1
TX_Commit	1100	1100	1100	TX	Transaction finished. Deadlock detected.
TX_Begin	1100	0100	1100	WF Serial	Transaction retries for work-items 0. Work-item 1 remains inactive.
Memory Access	1100	0100	1100	WF Serial	Only work-item 0 accesses to memory.
TX_Commit	1100	0100	1100	WF Serial	Transaction finished. Return to TX mode.
TX_Begin	0100	0000	0100	TX	Transaction retries for work-item 1
Memory Access	0100	0000	0100	TX	No conflicts detected
TX_Commit	0100	0000	0100	TX	Transaction finished for every work-item.

Table 1. Example of the execution model considering 4 work-items. Double lines separate different transaction retries.

2. Ownership detection: as signatures can return false positives, we use this phase to differentiate between a second access to the same memory location (true positive) from a new access to memory (false positive). This phase is required when the work-item currently accessing to the memory bank is the only one returning a positive in the signatures.
3. Conflict broadcast: this phase serves to broadcast conflict information from different memory banks and update transactional metadata.

Note that, as the fast conflict detection is performed in parallel, two (or more) work-items might abort each other unnecessarily. Suppose addresses A and B are located in different memory banks. If work-item 0 accesses A and work-item 1 accesses B, they update the corresponding signatures to register that access. If, later in the program, work-item 0 accesses B and work-item 1 accesses A, both conflicts are detected in parallel, aborting the execution of both A and B. In addition, having an unified shared signature to represent both read-set and write-set presents some issues. Applications that perform read-modify-write operations on a set of memory positions may benefit from an unified signature and detecting conflict during read operations. For instance, suppose a work-item reads A, performs some operations, and then write the results back to A. Detecting the conflict during the write operation results in a waste of time and computational resources. Detecting conflicts during the read operations results in an early conflict detection, reducing the amount of wasted work. On the contrary, if the application reads and writes different memory positions, detecting a conflict during the read operation may result in an unnecessary abort of the application. Suppose work-item 0 reads A and writes B, while work-item 1 reads A and writes C. As both transactions write different memory positions, no real conflict happens. Detecting conflicts during the read operation using a shared signature may result in a conflict and the re-execution of one of the transaction. If conflicts are detected only during write operations, then no conflict happens and both transactions are allowed to finish in parallel.

Version Management For version management, GPU-LocalTM allocates a region of local memory called *shadow memory*. Write accesses update this region of memory. A non-conflicting write access performs a backup of the original value and sets the current work-item as owner for this memory position. New (speculative) values are written directly to their final memory locations. Conflicting work-items use this information to discard the speculative values and restore the old ones.

3 Improving the serialization mechanism

The wavefront serialization mode implemented in GPU-LocalTM selects a single work-item within the wavefront for execution. Specifically, whenever two consecutive retries of the transaction finish with the same TCM mask, the work-item with lower identifier is the one selected to retry in wavefront serialization mode. This choice has two important consequences.

The first consequence is that by choosing only one work-item for re-execution may not be optimal and multiple work-items could be selected at the same time. We can think of an execution with 4 work-items: A, B, C, and D. These work-items can conflict in pairs (A and B conflict mutually, while C and D also conflict mutually), but with no other work-item within the wavefront. This situation can create a deadlock and trigger the wavefront serialization mode, where only one work-item is selected for execution. However, as work-items conflict in pairs, we can select both A and C (or B and D) for execution during the serialization mode. We refer to choosing only one work-item as *single* work-item selection, while selecting two or more work-items is called *multiple* selection.

The second consequence of choosing for re-execution the work-item with lower identifier can affect performance in some cases. Depending on the memory access pattern, and due to the requester loses policy, a work-item may cause many others to conflict. In some scenarios, the first work-item within the wavefront causes many others to conflict. In this case, in order to avoid future conflicts, it makes sense to select it to execute in the first place if the wavefront requires the use of the serialization mode. This way, it will finish its transaction in serialization mode and will no cause other work-items to conflict when the wavefront returns to transactional mode. However, if the last work-item within the wavefront is the one causing most of the conflicts, starting by choosing work-items with lower identifiers for re-execution will keep this work-item conflicting with others for a long period of time. In this case, the serialization mode is more effective if it chooses the work-item with higher identifier. In order to explore the different options, we create the *ascending* and *descending* mechanisms which select the work-item with lower identifier and higher identifier when entering the serialization mode, respectively.

The descending mechanism can be implemented as described in GPU-LocalTM, but processing the TCM mask backwards. The mechanism selecting multiple work-items per wavefront for re-execution after detecting a deadlock requires changes in the conflict detection mechanism and the SIMT execution model proposed by GPU-LocalTM. In addition to the EXEC and TCM masks used by GPU-LocalTM we propose the use of a new mask to keep track of the conflict pattern. We call this mask Multiple Conflict Mask (MCM). In the evaluated architecture, MCM keeps one bit per work-item within the wavefront (this is, a 64-bit mask). If the bit corresponding to a given work-item is set to 1, it means that is a candidate for re-execution in the next retry when running the serialization mechanism. If, otherwise, the value of its bit is 0 it means that the work-item is not chosen for re-execution. Initially, when a transaction starts (or restarts its execution), every bit in MCM is set to 1 (i.e., initially, every work-item is a candidate to re-execute if it finds any conflict). The update of this mask is done during the conflict detection mechanism. At the end of the transaction, the SIMT execution model must use the information stored in MCM to select which work-items re-execute. Combining the order (ascending or descending) and the choice of work-items (single or multiple) we obtain 4 configurations of the serialization mechanism (ascending-single, descending-single, ascending-multiple, and descending-multiple)

3.1 Conflict Detection Mechanism

GPU-LocalTM implements a distributed per-bank conflict detection mechanism which executes in three stages. The first stage, fast conflict detection, detects conflicts and updates TCM. The second stage, ownership detection, discriminates self-conflicting memory accesses from new ones. The third stage, conflict broadcast, propagates conflict information among the banks in order to clear transactional metadata.

Implementing a conflict detection mechanism that allows the execution of multiple work-items when entering the serialization mechanism requires modifications on the fast conflict detection stage. When a work-item detects a conflict, it compares its own identifier against the identifier of the owner of the memory position that is being accessed. We can get this information from the ownership records stored in the shadow area allocated by GPU-LocalTM. The MCM is updated with this information depending on the election of the ascending or descending mechanism. Conflicts with work-items in different wavefronts always set the current work-item as candidate for re-execution, leaving its bit in MCM with the default value of 1. When the ascending algorithm is enabled, work-items with lower identifier have priority over those with higher identifier. Thus, if the conflict is detected with a work-item with higher identifier, the work-item will set itself as candidate for for re-execution and will leave its bit in MCM with the default value of 1. Otherwise, if the identifier of the conflicting work-item is lower, the current work-item will write a value of 0 in its corresponding bit in MCM. The opposite happens when the descending algorithm is enabled. As work-items with higher identifier have higher priority, a work-item updates its corresponding MCM bit to 0 if the conflict is detected with a work-item with higher identifier. Otherwise, its bit in MCM will remain set to the default value of 1.

In summary, MCM contains 1 for work-items that potentially can execute in the next iteration or 0 otherwise.

3.2 SIMT execution mode

GPU-LocalTM modifies the SIMT execution model of the Southern Islands architecture to allow for the correct execution of transactions. The TX_Begin instruction starts the transaction and decides whether the serialization mode is required or not. The first time TX_Begin instruction executes, the transactional mode starts. If there is a conflict, TMC is updated with information on the work-items that conflicted. In that case, the TX_Commit instruction decides to restart the transaction for those work-items. If two consecutive transactions retries finish with the same value in TMC it means that the transaction is stuck in a deadlock. In this case, the TX_Begin starts the transaction in wavefront serialization mode.

When the wavefront serialization mode is required, we use MCM to choose a subset of the conflicting work-items for re-execution. As the wavefront serialization mode is active after, at least, one transaction retry, MCM is already updated with information on the conflicts. In the descending algorithm, a bit is set to 0 in MCM if its corresponding work-item had a conflict with a work-item that belongs to the same wavefront but with lower (or higher, in case of the descending algorithm) identifier. Those work-items with its bit set to 1 in MCM are candidates to execute the transaction in the next retry. In order to activate these work-items we perform the operation TCM = TCM & $\sim$ MCM. Note that a 0 in TCM means that the work-item has no conflict and, thus, is active for execution in the next retry.

During the wavefront serialization mode, new conflicts may appear due to aliases in the signatures or changes in memory by other wavefronts. If a conflict is detected when executing multiple work-items during serialization, then the work-group serialization mode is active and only one work-item is selected for re-execution.

4 Improving conflict detection

In our initial design [14], an unified signature is used to record both reads and writes accesses. Unified signatures are private per work-item and per bank (i.e., there are 256 Bloom filters per bank). Since there are a large number of signatures, 8-bit Bloom filters are used to limit the amount of memory resources needed. We name this conflict detection mechanism as Private Read/Write Signature Conflict Detection (pRW-sig).

By using private unified read-write signatures in our pRW-sig implementation, we provide GPU-LocalTM with an efficient and low-cost mechanism for conflict detection. However, as the same signature is used to register reads and writes, it results in false read-read conflicts. In order to avoid these conflicts, Choi et al. [5] propose the use of smaller, per-transaction, helper signatures to represent the write set. Memory reads are registered only in the unified signature, while memory writes are registered in both the unified and the helper signatures. A conflict is detected during a memory read operation if both the unified and helper signatures return a positive. A false read-read conflict would return a positive in the unified signature, but not in the helper signature. This allows us to filter out read-read conflicts, which provides a more precise conflict detection mechanism. A property of helper signatures is that they are meant to be smaller than the unified signatures (otherwise, using separate read and write signatures would be more efficient). As 8-bit unified signatures are small enough, we propose the use of a shared per-wavefront 32-bit helper signatures to record writes. We name this solution Shared Write-Only Signature Conflict Detection (sWO-sig).

These two mechanisms use signatures to detect conflicts, considering read-read interactions as conflicts (pRW-sig) or not (sWO-sig). Signatures are used as an efficient method to detect conflicts, as they are evaluated quickly. However, its use may introduce false conflicts as their evaluation may produce false positives. To complement these two conflict detection mechanisms we propose the use of a directory, which does not return any false positive. This directory is mapped in the local memory portion allocated by GPU-LocalTM to back-up speculative memory values. This results in a more precise but slower conflict detection mechanism, as the access latency to local memory is higher than the latency of evaluating signatures but returns no false positives. The Directory-based Conflict Detection (DCD) mechanism uses these speculative values to detect conflicts. This method is equivalent to pRW-sig in the sense that it consider read-read accesses as conflicts. By adding shared and modified bits to each directory entry (S and M) we can differentiate read-read conflicts and, thus, allow concurrent read accesses to memory without conflicts. Initially, both S and M are 0. When accessing a memory location for the first time, the version management mechanism sets the accessing work-item as owner in shadow memory. A read access for different work-items are allowed as long as M is disabled, and they enable the S bit. A write access is allowed only if the owner is the current work-item and the S is disabled; otherwise is considered as conflict. A success-

	read-read conflicts	no read-read conflicts
False positives (signatures)	pRW-sig	sWO-sig
No false positives (directory)	DCD	SMDCD

Table 2. Classification of the conflict detection mechanisms.

ful write will set the bit M as enabled. This solution is called Shared-Modified Directory-based Conflict Detection (SMDCD).

Table 2 classifies the conflict detection mechanism proposed for GPU-LocalTM. Note that the conflict detection mechanism is designed to be coupled with the modifications made to the SIMT execution model to ensure forward progress. The SIMT execution model is modified to allow execution of a single or multiple work-items within the whole work-group in the case a deadlock is detected. In addition, these conflict detection mechanisms can coexist implemented in hardware and enabled or disabled depending on the characteristics of the applications to execute.

5 Modeling

The designs described in sections 3 and 4 can coexist in hardware and occupy a different amount of hardware resources. Table 3 contains the hardware resources available per CU in the AMD's Southern Islands GPU, and those required by the original design of GPU-LocalTM when running a work-group (composed of four wavefronts).

In order to implement the multiple serialization mechanism (i.e., ascending-multiple and descending-multiple) we require storage for the novel MCM mask. As these mask are located per wavefront we can map them in scalar registers which can be accessed by each wavefront. For work-groups with four wavefronts we need to store four 64-bit masks, resulting in the use of eight scalar registers. No other memory resources are required to implement the proposed SIMT execution mode mechanisms.

The most resource-consuming conflict detection mechanism is sWO-sig, as it requires to store both privates and shared signatures. The storage of per-bank and per-work-item private signatures is done using private vector registers and it is already modeled by GPU-LocalTM. This results in a usage of 8192 vector registers (assuming 8-bit signatures, 256 work-items per work-group, 32 memory banks, and 32-bit vector registers). Adding shared signatures require an extra 32-bit signature per wavefront and per memory bank. These signatures can be mapped into scalar registers, resulting in $4 \times 32 = 128$ scalar registers to store the shared signatures. The directory-based algorithms require no use of registers to store signatures, resulting in no pressure on both vector and scalar registers. The DCD algorithm uses the existing memory allocated by GPU-LocalTM to store speculative values. The SMDCD algorithm requires 2 extra bits (S and M) per memory location and per memory bank. As the maximum number of

Feature	Value	GPU-LocalTM
Compute Units (CUs)	32	-
Vector Registers per CU	65536	2048
Scalar Registers per CU	2048	8
SIMD Units per CU	4	-
SIMD Lanes	16	-
LDS Size per CU	65546 bytes	36405 bytes (55%)
LDS Banks	32	-
Minimum LDS Latency	2 cycles	5 cycles

Table 3. Relevant features of the AMD's Southern Islands GPU implementation on Multi2sim 4.2 and the maximum amount of resources required by 1 work-group using GPU-LocalTM.

memory words accessed by GPU-LocalTM is 7281, we need 14562 bits to store S and M. We can carve this space from the vector registers, reducing the number of vector registers available in 456 ($< 1\%$ of 65536 available).

6 Evaluation

In this section we evaluate the behavior of GPU-LocalTM when using the four mechanisms for serialization described in the previous sections: ascending-single (GPU-LocalTM default), descending-single, ascending-multiple, and descending-multiple. Furthermore, we evaluate the following mechanisms for conflict detection: pRW-sig (GPU-LocalTM default), sWO-sig, DCD, and SMDCD. For a better understanding of the proposal, we choose pRW-sig as default conflict detection mechanism when evaluating the different serialization modes. When evaluating the different conflict detection mechanisms, we choose ascending-single as the serialization mechanism.

We evaluate our proposal using four benchmarks, using for each one a high contention input (HC) and a low contention input (LC). The application **HT** is the implementation of hash table where, after selecting the bucket to insert data, we go through every position until we find an empty space. This application performs many read-only transactions when trying to locate the empty space. The indexed table (**IT**) is a different implementation of hash table. In this case the index of the next empty place is stored in an array. Transactions are simple read-modify-write operations on a single memory position. The graph coloring (**GC**) application is adapted from previous work [6]. It features transactions that read multiple memory positions, but a single write operation. In addition, it presents a number of read-only transactions. K-Means (**KM**) is a clustering algorithm adapted from Rodinia [3, 4] to work in local memory. Transactions are used to modify the position of the clusters, allowing read-modify-write operations on multiple memory locations.

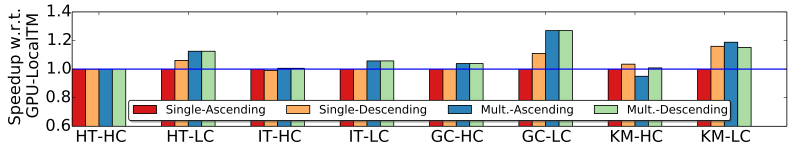

Fig. 2. Speedup evaluation with respect to GPU-LocalTM.

6.1 SIMT execution mode evaluation

Speedup. Figure 2 shows the speedup obtained when using the four methods proposed for serialization with respect to GPU-LocalTM. GPU-LocalTM implements the ascending-single algorithm. In most of the applications we observe that selecting multiple work-items for re-execution benefits performance by up to 30%. The use of the descending algorithms show better or similar performance that their ascending counterparts, depending on the memory access pattern. It is in K-Means with high contention inputs (KM-HC) where performance is affected by using the ascending-multiple algorithm. The reason is that, when accessing multiple memory locations, it create aliases in the signatures and causes false conflicts to appear even executing the serialization mode.

Breakdown. Figure 3 shows the normalized execution breakdown for the proposed serialization algorithms. Usually, the multiple algorithm improves performance and has a positive impact in the time required to execute transactions. In addition, the memory overheads of managing shadow memory in order to update MCM have no noticeable impact in performance. The reason is that choosing multiple work-items for execution pays off the memory overheads.

Transaction Type. Figure 4 shows the percentage of transactions that require the use of serialization for each algorithm. In some cases, there is no difference in the number of transactions of each type to be executed. For HT-LC, GC-HC and GC-LC we observe that the number of transactions requiring work-group serialization is sightly higher for the ascending-multiple and descending-multiple

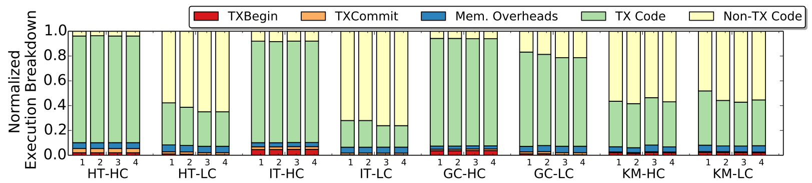

Fig. 3. Normalized execution breakdown. 1, 2, 3, and 4 stand for ascending-single, descending-single, ascending-multiple, and descending-multiple, respectively.

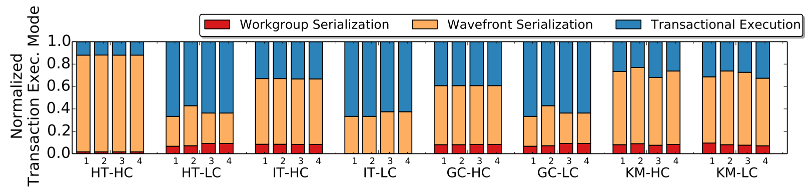

Fig. 4. Normalized type of transaction. 1, 2, 3, and 4 stand for ascending-single, descending-single, ascending-multiple, and descending-multiple, respectively.

algorithms. The reason is that false conflicts during the wavefront serialization mode require the use of work-group serialization to solve conflicts.

6.2 Conflict detection mechanism evaluation

Speedup. Figure 5 shows the speedup obtained when using the four conflict detection mechanisms proposed, normalized to the execution time of pRW-sig (GPU-LocalTM default implementation). Applications with read-only transactions such as HT and GC benefit from the shared signatures provided by sWO-sig, as this mechanism is able to filter out false read-read conflicts. For applications such as IT and KM, where the read-set equals to the write-set, the performance difference is minimal. Using the slower but precise directory-based mechanisms (i.e., DCD and SMDCD) only results effective in GC with low contention input. For the rest of the applications, these algorithms do not perform better that their signature-based counterparts. The reason is that usually the fast evaluation of the signatures pays off the false positives caused by address aliases. In the case of the application IT with high contention input, the original implementation of GPU-LocalTM results optimal. The reasons are that the application performs read-modify-write operations on the same memory positions and that the high number of conflicts are efficiently detected by the signatures.

Breakdown. Figure 6 shows the normalized execution breakdown for the conflict detection mechanism. On one hand, as the directory is located in memory, the use of directory-based algorithms result in a larger memory overhead as compared to the signature-based algorithms. As we see in Figure 5, this increment in

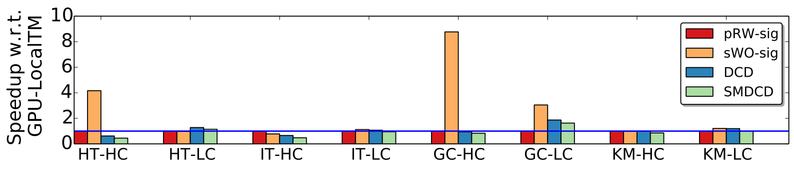

Fig. 5. Speedup evaluation with respect to GPU-LocalTM.

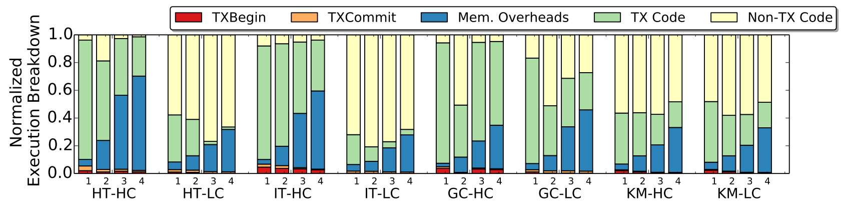

Fig. 6. Normalized execution breakdown. 1, 2, 3, and 4 stand for pRW-sig, sWO-sig, DCD, and SMDCD, respectively.

the memory overhead does not harm performance as the conflict detection mechanism is more precise. On the other hand, normally the signature-based solutions spend more time executing transactions as compared to their directory-based counterparts. The reason is that the less precise conflict detection mechanism requires more transactions to be retried due to false conflicts. In cases such as HT and GC we observe a reduction in the time inside transactions when using sWO-sig. This is explained by a reduction in the number of transactions that need to retry, as these benchmarks feature read-only transactions that do not cause read-read conflicts when using the shared signatures.

Transaction Type. Figure 7 shows the percentage of transactions that require the use of serialization when using the pRW-sig, sWO-sig, DCD, and SMDCD algorithms. We observe that the use of shared signatures (sRW-sig) discriminate the false conflicts of the read-only transactions, reducing the amount of transactions that require serialization in HT-HC, GC-HC and GC-LC. For the HT and IT algorithms (that feature short transactions) with low contention inputs the directory-based algorithms are able to finish with no requirement of serialization.

7 Compiler design considerations

In section 5 we model the hardware resources needed by GPU-LocalTM and the proposed improvements. These resources are carved from the existing memory

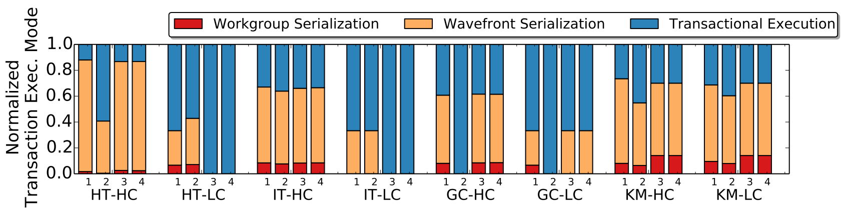

Fig. 7. Normalized type of transaction. 1, 2, 3, and 4 stand for pRW-sig, sWO-sig, DCD, and SMDCD, respectively.

in the GPU architecture. Specifically, memory storage for speculative values and transactional metadata occupy up to 55% of the 64Kb available per CU, and a number of vector and scalar registers are needed in order to store execution masks and signatures, when required. For instance, the less demanding implementation would use an ascending-single serialization mode as well as a DCD conflict detection mechanism. This implementation requires no execution mask different from those defined by GPU-LocalTM and no storage for signatures. On the opposite side, the ascending-multiple algorithm combined with the sWO-sig requires of scalar registers to store the novel MCM mask and shared signatures, as well as vector registers to store private signatures.

In addition, in section 6 we evaluate the performance of the different algorithms. In some cases the detection of conflict during read operations using the pRW-sig and DCD results beneficial, while for applications with read-only transactions the algorithms able to filter out read-read conflicts such as sWO-sig and SMDCD are more efficient.

7.1 High level constructs

High-level languages, such as OpenCL, should provide language constructs to allow programmers to implement TM applications. At least, the TX_Begin and TX_Commit operations have to be present in order to define the transaction boundaries (an alternative is to define a code block as *atomic*). Despite is not required by hardware to label memory operations as transactional (i.e., TX_Read and TX_Write) it could be of help to the compiler in order to determine which memory positions (or variables) require of storage for speculative values. An initial analysis of the code can determine the amount of memory, scalar registers and vector register required by the program and the TM implementation.

7.2 TM feasibility

If the hardware requirements of a GPU program (with no TM support) exceed those available in hardware, the compiler (or runtime, for pre-compiled programs) is not able to compile (or execute) the program. TM programs introduce new restrictions for these requirements. If, according to the values defined in Table 3, the program and TM requirements overtake those available in the GPU, the compilation of the TM code is not feasible. However, if the hardware requirements of the program (with no TM support) are lower than those available in the GPU, compilers can perform a code transformation of similar to the represented in Figure 1 to replace the TM implementation for a coarse-grained lock implementation. This would affect the performance of the application, but will relieve programmers from thinking on a TM and non-TM solution.

The same analysis apply to the different implementations of the SIMT execution mode and the conflict detection mechanism. Initially, the compiler can consider an ascending-multiple and sWO-sig implementations, which in many cases result in a good performance. However, the register requirements of these algorithms might be too high for a given program. For instance, many loop indices and other per-wavefront information are mapped into scalar registers. If the number of scalar registers required by the program is too high, the sWO-sig conflict detection mechanism might not be feasible as it maps the shared signatures into scalar registers. In that case, the compiler should inform the hardware through the Application Binary Interface (ABI) that the sWO-sig can not be used, relying the TM support to the pRW-sig or a directory-based algorithm. In the case of selecting the pRW-sig, the same analysis is to be performed for the vector registers.

7.3 Read-only transactions

Analysis of the high-level code can be used to determine if any of the transactions (or any code path inside a transaction) is read-only. During evaluation, we observe that sWO-sig performs better in read-only transactions thanks to its ability to differentiate conflicts in read-read accesses and the effectiveness of the signatures to easily detect conflicts. For this reason, in case of availability of resources, compilers should choose the sWO-sig when detecting read-only transactions. Additionally, having read-only transactions is a good hint for selecting the execution of multiple work-items as it is unlikely that read-only transactions cause a false positive if they are chosen for re-execution.

7.4 Large read-set and write-set

Applications with a large memory imprint, such as KM, may cause false positives in the signatures due to the large number of aliases (i.e., memory addresses that map on the same bits in the signatures). For these applications, a directory-based conflict detection mechanism is more efficient. Despite, during our evaluation, it did not outperform the signature-based solutions, it requires no hardware resources to store signatures. The application breakdown shows that, for this type of application, the amount of time executing (and retrying) transactions is reduced with respect to the signature-based implementations.

8 Conclusions

In this paper we analyze previous work on the SIMT execution model for transactional memory on GPU scratch-pad memory and we gather different proposals for improving its performance. Firstly, when a deadlock is detected, we allow to choose more than one work-item for re-execution. Secondly, we allow to change the order of execution of conflicting work-items. Results show that we can obtain up to 30% of performance improvement with minimal overhead. Lastly, we consider different conflict detection techniques that speed up the performance of the applications up to 9X.

These features can coexist implemented in hardware. The hardware can be configured via ABI to select the most appropriate technique. In this paper we provide considerations for compiler design in order to choose which features are convenient for a given application. Future research directions consider implementing these compiler features as well as analyzing other compiler and runtime decisions in order to improve TM efficiency on GPUs.

References

1. Bloom, B.H.: Space/time trade-offs in hash coding with allowable errors. Communications of the ACM 13(7), 422–426 (Jul 1970)
2. Cederman, D., Tsigas, P., Chaudhry, M.T.: Towards a software transactional memory for graphics processors. In: 10th Eurographics Conf. on Parallel Graphics and Visualization (EG PGV'10). pp. 121–129 (2010)
3. Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J., Lee, S.H., Skadron, K.: Rodinia: A benchmark suite for heterogeneous computing. In: IEEE Int'l. Symp. on Workload Characterization (IISWC'09). pp. 44–54 (Oct 2009)
4. Che, S., Sheaffer, J., Boyer, M., Szafaryn, L., Wang, L., Skadron, K.: A characterization of the rodinia benchmark suite with comparison to contemporary cmp workloads. In: IEEE Int'l. Symp. on Workload Characterization (IISWC'10). pp. 1–11 (Dec 2010)
5. Choi, W., Draper, J.: Improving utilization of hardware signatures in transactional memory. IEEE Trans. on Parallel and Distributed Systems 24(11), 2230–2239 (Nov 2013)
6. Duffy, K., O'Connell, N., Sapozhnikov, A.: Complexity analysis of a decentralised graph colouring algorithm. Information Processing Letters 107(2), 60 – 63 (2008)
7. Fung, W.W.L., Aamodt, T.M.: Energy efficient GPU transactional memory via space-time optimizations. In: 46th Ann. IEEE/ACM Int'l. Symp. on Microarchitecture (MICRO'13). pp. 408–420 (2013)
8. Fung, W.W.L., Singh, I., Brownsword, A., Aamodt, T.M.: Hardware transactional memory for GPU architectures. In: 44th Ann. IEEE/ACM Int'l. Symp. on Microarchitecture (MICRO'11). pp. 296–307 (2011)
9. Harris, T., Larus, J., Rajwar, R.: Transactional Memory, 2nd. Morgan & Claypool Publishers, USA (2010)
10. Herlihy, M., Moss, J.E.B.: Transactional memory: Architectural support for lock-free data structures. In: 20th Ann. Int'l. Symp. on Computer Architecture (ISCA'93). pp. 289–300 (1993)
11. Holey, A., Zhai, A.: Lightweight software transactions on GPUs. In: 43rd Int'l Conf. on Parallel Processing (ICPP'14). pp. 461–470 (2014)
12. Khronos: The OpenCL specification. version 2.0
13. Ubal, R., Jang, B., Mistry, P., Schaa, D., Kaeli, D.: Multi2Sim: A Simulation Framework for CPU-GPU Computing. In: 21st Int'l. Conf. on Parallel Architectures and Compilation Techniques (PACT'12) (2012)
14. Villegas, A., Navarro, A., Asenjo, R., Plata, O., Ubal, R., Kaeli, D.: Hardware support for local memory transactions on gpu architectures. In: 10th ACM SIGPLAN Workshop on Transactional Computing (TRANSACT 2015, part of FCRC 2015). (Jun 2015)
15. Xu, Y., Wang, R., Goswami, N., Li, T., Gao, L., Qian, D.: Software transactional memory for GPU architectures. In: Ann. IEEE/ACM Int'l. Symp. on Code Generation and Optimization (CGO'14). pp. 1:1–1:10 (2014)