Hardware Support for Scratchpad Memory Transactions on GPU Architectures

Alejandro Villegas[1(✉)], Rafael Asenjo[1], Angeles Navarro[1], Oscar Plata[1], Rafael Ubal[2], and David Kaeli[2]

[1] Department of Computer Architecture, University of Málaga, Andalucía Tech, 29071 Málaga, Spain
{avillegas,asenjo,magonzalez,oplata}@uma.es

[2] Department of Electrical and Computer Engineering, Northeastern University, Boston, MA, USA
{ubal,kaeli}@ece.neu.edu

Abstract. Graphics Processing Units (GPUs) have become the accelerator of choice for data-parallel applications, enabling the execution of thousands of threads in a Single Instruction - Multiple Thread (SIMT) fashion. Using OpenCL terminology, GPUs offer a global memory space shared by all the threads in the GPU, as well as a low-latency local memory space shared by a subset of the threads. The latter is used as a scratchpad to improve the performance of the applications.

We propose GPU-LocalTM, a hardware transactional memory (TM), as an alternative to data locking mechanisms in local memory. GPU-LocalTM allocates transactional metadata in the existing memory resources, minimizing the storage requirements for TM support. In addition, it ensures forward progress through an automatic serialization mechanism. In our experiments, GPU-LocalTM provides up to 100X speedup over serialized execution.

Keywords: Transactional memory · Scratchpad memory · GPGPU

1 Introduction

Graphics Processing Units (GPUs) have been adopted as hardware accelerators given their ability to significantly improve the performance of data-parallel applications. Using OpenCL terminology, GPUs are organized as a set of highly multi-threaded Single Instruction - Multiple Thread (SIMT) cores called compute units (CUs) and feature two different memory spaces. The *global memory* space provides high capacity with high latency. In contrast, the *local memory* space (named *shared memory* in CUDA terminology) features a smaller capacity with lower latency. Programmers are encouraged to use local memory as a scratchpad to speedup their applications (in fact, 27 out of the 52 sample applications in the AMD APP SDK for OpenCL prove the benefit from using local memory).

F.F. Rivera et al. (Eds.): Euro-Par 2017, LNCS 10417, pp. 273–286, 2017.
DOI: 10.1007/978-3-319-64203-1_20

Transactional Memory (TM) [6,7] has emerged as a promising alternative to locking mechanisms to coordinate concurrent threads. TM provides the concept of a transaction to determine the bounds of a critical section (usually providing TX_Begin and TX_Commit functions) enforcing atomicity and isolation. In contrast to traditional lock-based mechanisms, transactions are allowed to run in parallel. A conflict occurs if two transactions have to access to the same memory location and, at least, one of the accesses is a write. In such situations, one of the transactions aborts, discarding its updates to memory and restarting its execution. This is achieved by implementing appropriate conflict detection and version management mechanisms. Recently, TM solutions have been proposed for GPU global memory, both software [2,8,10,12] and hardware [3–5].

Motivating Example. The left side of Fig. 1 shows the traditional implementation of a spinlock. In the SIMT programming model, as threads execute in lockstep, only one of them is able to get the lock and leave the while-loop (line 1). As there is a divergence in the execution of the program, the SIMT programming model sets a convergence point at the end of the while-loop (line 2), creating an implicit barrier. This implicit barrier forces the thread who acquired the lock to wait for the rest to finish the execution of the while-loop. This will never happen, as the lock is held by the waiting thread and the remaining threads will not leave the while-loop until the lock is released. Thus, the classic spinlock creates a deadlock in the SIMT programming model. The central part of Fig. 1 shows the required transformation for this spinlock to work. In this case, all the active threads enter the while-loop (line 2). The convergence point (i.e., implicit barrier) for this loop is set in line 8, which will be eventually reached by all the threads. Then, only one of the threads acquires the lock (line 3) inside an if-statement. In this case, the convergence point is set at line 7. This way, the threads that did not acquire the lock perform the implicit barrier at line 7, while the thread that acquired the lock executes its critical section (line 4), sets itself to go to the convergence point of the while-loop (line 5), and releases the lock (line 6). With this code transformation we can safely implement coarse-grained locks in the SIMT programming model.

```
1 while (!getLock()){;}
2
3 //Critical Section
4 releaseLock();
```

```
1 bool done = false;
2 while (!done){
3   if (getLock()){
4     //Critical Section
5     done = true;
6     releaseLock();
7   }
8 }
```

```
1 TX_Begin();
2   //Critical Section
3 TX_Commit();
```

Fig. 1. Coarse-grained lock implementation in CPUs (left), its required transformation to avoid deadlocks in the SIMT programming model (center), and the TM-based solution (right).

Using a coarse-grained lock creates an inefficient serialization of the execution of the critical sections. Fine-grained locks can help improving parallelism, but its use is complicated and error-prone. Furthermore, its implementation can be harder in the SIMT programming model, as transformations similar to the ones shown in Fig. 1 are required in order to avoid deadlocks and livelocks. In addition, the use fine-grained locks is application-specific and a generic template can not be provided. Thus, it is hard to implement automatic code transformations similar to the one explained above. Given these problems, TM has been proposed to both improve parallelism of the applications and ease programming. The right side of Fig. 1 shows how simple is the implementation of mutual exclusion using the TM interface. In order to discuss the performance of the TM implementation, the applications Hash Table (HT) and Genetic Algorithm (GA) (see Sect. 5 for a full description) were implemented on a GPU using fine-grained locks (FGL) as well as with TM to coordinate the execution of 256 threads (*work-items* in OpenCL terminology). The implementation of these applications were done taking advantage of the low-latency provided by the local memory. HT is a simple application and the 3 implementations require a similar programming effort. Both the FGL and TM versions outperform the serial execution (90X and 60X, respectively). To implement GA using a FGL approach, lock acquisition has to be serialized to avoid deadlocks, requiring more programming effort. Also, execution time increases by 30% due to lock management overhead. However, a TM-based solution halves the execution time and requires a similar programming effort as a serial implementation.

This paper introduces GPU-LocalTM, a lightweight hardware TM for local memory. The goal is to use TM as an efficient alternative to existing methods (i.e., locks). GPU-LocalTM is designed in a way that reuses existing memory resources (if active), and can be disabled (if not needed). The conflict detection and version management mechanisms are distributed per local memory bank, improving concurrency. Lastly, GPU-LocalTM implements an automatic serialization mechanism that ensures forward progress of the transactions without the need of any programmer action.

The rest of the paper is organized as follows. Section 2 provides the background and discusses the related work. Section 3 presents the design of GPU-LocalTM. Section 4 presents the simulation framework used for evaluation, and Sect. 5 discusses the experimental evaluation. Finally, Sect. 6 draws the conclusions.

2 Background and Related Work

Baseline GPU Architecture. We use OpenCL terminology to describe our baseline GPU architecture, which is the AMD's Southern Islands [1] (see Fig. 2). An *ultra-threaded dispatcher* assigns *work-groups* (work-groups are a set of computing threads called *work-items*) to the *Compute Units* (CUs). A work-group is assigned to a single CU, but a CU may contain several work-groups. This architecture supports a maximum of 256 work-items per work-group. This set of work-items are grouped in 4 *wavefronts* of 64 work-items executing in lockstep. Wavefronts are the schedulable unit within the CU. All work-groups share data through the physical *global memory* available on the GPU. Work-items within a work-group have access to the *local memory*, a low-latency memory used to speedup applications. Each CU contains a wavefront scheduler, 4 SIMD units (consisting of vector ALUs and general purpose vector registers), a scalar unit (with a scalar ALU and general purpose scalar registers), a local data share (LDS) unit and a L1 data cache.

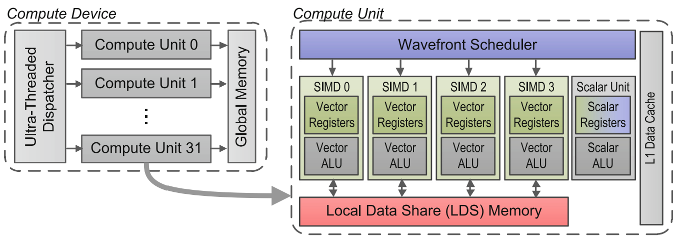

Fig. 2. Baseline GPU architecture: AMD's Southern Islands.

The LDS unit, which contains the *local memory*, deserves special attention as it is a key component in the GPU-LocalTM design. The LDS included in each CU features 64 KB distributed across 32 banks with interlaced addressing (consecutive memory addresses map to consecutive banks). Each work-group is allowed to use only 32 KB, leaving the other 32 KB reserved for concurrent work-group execution. The LDS unit is in charge of managing this local memory. The accesses to local memory issued by a wavefront are scheduled by the LDS unit, supporting up to 32 coalesced (i.e., without bank conflicts) accesses simultaneously. Uncoalesced memory accesses are serialized by the LDS unit.

Related Work. To the best of our knowledge, Kilo TM [3–5] is the only existing hardware TM for GPUs. Kilo TM [5] operates on global memory and implements conflict detection and version management at commit time (lazy) using a specific Commit Unit. Kilo TM was improved by considering read-only transactions and reducing bus communication [4], and to detect conflicts before sending transactions to the Commit Units and to stall transactions that are likely to conflict [3].

In our proposal, we target different applications: Kilo TM addresses applications that synchronize at global memory and our proposal supports synchronization at local memory. Both memory spaces have different purposes and very different characteristics. For instance, the difference in latency of both memory spaces affects application performance, even if they do not use any TM support. Once both TM approaches are integrated, applications can be developed taking full advantage of the complete GPU memory model. In addition, we explore eager (at memory access time) conflict detection and version management, in contrast to the lazy Kilo TM approach, by adding logic to the LDS unit instead of having a new dedicated unit for TM.

There are a number of software TM proposals for GPUs which only use the global memory space. Cederman *et al.* proposes two STM systems for graphics processors [2], operating at work-group granularity. Xu *et al.*, Holey *et al.*, and Shen *et al.* propose different STM approaches working at work-item granularity [8,10,12].

3 GPU-LocalTM Design

GPU-LocalTM is a hardware TM for GPU local memory. Transactional execution, conflict detection, and, version management are implemented with minor logic modifications in the wavefront scheduler, SIMD and LDS units. Required space is taken from the scalar register file and the local memory banks.

Transactional SIMT Execution. In our baseline architecture, control flow of the SIMT programming model is implemented with predication, using two 64-bit masks managed by the hardware and the compiler. The *execution mask* (EXEC) indicates, per wavefront, the work-items that are running or disabled (one bit per work-item). The *vector comparison mask* (VCC) stores, for each work-item within the wavefront, the resulting Z flag of arithmetic/logic operation. By combining EXEC and VCC, compilers implement loops and conditionals. The VCC and EXEC masks are mapped into two consecutive 32-bit scalar registers each one (four 32-bit registers in total) [1].

In order to define the bounds of a transactional block of code, we add two instructions to the ISA: TX_Begin and TX_Commit (see Fig. 1). These instructions work at a wavefront granularity as work-items within the wavefront execute in lockstep. Local memory operations performed between these instructions are *transactional* and are instrumented via hardware for conflict detection and version management. The TX_Begin sets the beginning of the transaction. When the TX_Commit instruction is reached, the transactional SIMT execution is responsible for restarting the execution of the conflicting work-items (if any).

To implement this, we introduce a new 64-bit *transaction conflict mask* (TCM) per wavefront (one bit per work-item). Similarly to EXEC and VCC masks, TCM is mapped to two consecutive scalar registers. TCM is used to mark conflicting work-items. The reason for not reusing EXEC is that it is explicitly managed by the compiler [1] and allowing implicit hardware modifications can lead to inconsistent situations. When the work-items within a wavefront execute the TX_Begin instruction, the TCM mask is reset (all bits to 0, meaning that no conflict occurred). If a work-item detects a conflict the corresponding bit in TCM is set to 1. When this bit is 1, it indicates the such work-item is disabled (i.e., the enabled work-items are those whose EXEC bit is 1 and whose TCM bit is 0). If all the TCM bits are 0 when the TX_Commit instruction is reached, then all transactions have finished with no conflicts. In other case, conflicting transactions must retry the execution by copying TCM to EXEC and returning to the TX_Begin instruction (and, again, TCM is reset).

Instruction	EXEC	TCM	Comments
TX_Begin	111...1	000...0	Wavefront starts transactions.
Mem Access	111...1	010...0	Conflict detected by WI 1, which is disabled.
TX_Commit	111...1	010...0	Transactions of all WIs but 1 end. Wavefront rollbacks and restarts WI 1 (TCM is copied to EXEC).
TX_Begin	010...0	000...0	Only WI 1 retries transaction
...	010...0	000...0	No new conflicts detected.
TX_Commit	010...0	000...0	Transaction of WI 1 ends. All transactions complete.

Fig. 3. Example of transactional SIMT execution. A work-item (WI) is enabled if EXEC[WI] & !TCM[WI]. Single lines separate transaction executions.

Figure 3 shows an example of transactional execution using TCM. In this example, during the execution of the transactions, work-item 1 detects a conflict, is marked by setting its bit in TCM, and is disabled immediately. At commit time, the rest of work-items successfully complete their transactions and wait while work-item 1 is restarted. This second time, work-item 1 is able to complete.

Forward Progress. A livelock situation can be detected if the TCM remains the same after two consecutive transaction re-executions. This means that two or more work-items were not able to progress, creating an infinite loop. To resolve this without requiring programmer action, GPU-LocalTM includes a two-level serialization mechanism: *wavefront serialization (WfS) mode* and *work-group serialization (WgS) mode.*

The basic WfS mode is enabled when a livelock situation is first detected. In this mode, the transaction is retried a third time but, instead of clearing TCM at the beginning of the transaction execution, only one of the active bits is reset. This action results in the execution of the only selected work-item within the wavefront during the next transaction retry (i.e., the rest are already marked as conflicting). If the execution ends with no new conflicts, the transaction is again retried but in normal mode (i.e., not using WfS). Otherwise, the conflict may come from a work-item that belongs to a different wavefront. In such situations, the basic WfS mode transfers to the basic WgS mode. In this mode, only the current wavefront re-executes transactionally. Transactions executing in other wavefronts are aborted, rolled back and stalled at the *TX_Begin* instruction until the selected work-item ends execution. Now that a single work-item within the work-group is accessing local memory, no conflicts can occur and forward progress is assured. After this execution, the transaction returns to normal mode and the stalled wavefronts are allowed to continue execution.

Version Management. GPU-LocalTM uses eager version management, where new local memory values are stored in place while old values are saved on the side. Specifically, old values are stored in a memory area called *shadow memory*, allocated in local memory. These values are used to restore the original state of the local memory in case of a transaction abort. As the local memory is multi-banked (32 banks in the case of our baseline GPU architecture), version

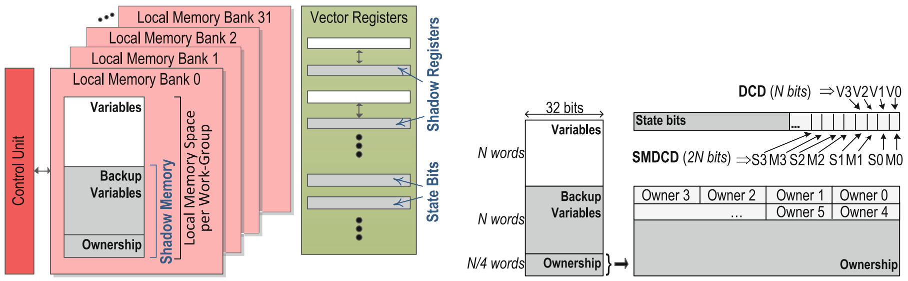

Fig. 4. Version management (LDS unit) and register checkpointing (SIMD units) (a) and shadow memory organization (b).

management and conflict detection can be carried out concurrently in different banks (i.e., there is a shadow memory per bank). The shadow memory area is organized in two spaces (see Fig. 4(a)): a backup space with enough room to store backups for all of the local memory variables declared within the kernel allocated in each bank, and an ownership space.

The shadow memory is organized as in Fig. 4(b): if there is a set of N words in local memory, a contiguous section of N words is allocated to backup the values, and after this section, $N/4$ additional words are reserved to store the owners. Each word in the ownership region stores 4 owners (1 byte each). Given this layout, when a memory access is issued to a location k, a backup value is stored at word position $N + k$, and the work-item ID (owner) is stored at word position $2N + k/4$, byte $k\%4$. By adopting this scheme, the hardware required to backup a memory value and store its owner is minimal, as it only performs integer addition and bit manipulation. In addition, capacity conflicts are avoided, as each memory location is ensured to have space for its backup. The shadow memory area is statically allocated by the compiler using the same mechanism used for regular local memory variables [1,11].

Register Checkpointing. When starting a transaction (TX_Begin instruction), the user-visible non-memory work-item state must be saved (and restored on transaction abort). This includes vector and scalar registers. Vector registers are checkpointed to a *shadow register* file. This is implemented by splitting the vector register file in each SIMD unit into two equally sized parts. Every two registers, one for each part, are paired together so as one of them acts as the backup (shadow) register of the other (see Fig. 4(a)). Scalar registers, on the other hand, are used to store scalar shared data for an entire wavefront, such as a for-loop index. As this information is shared by 64 work-items, if some of them commit their transactions while others abort, the value held by scalar registers become inconsistent. For this reason, scalar registers are not checkpointed at the beginning of a transaction. To allow for loops within a transaction, the compiler must promote the use of work-item-private vector registers.

Conflict Detection. GPU-LocalTM performs eager conflict detection at a work-item level. During the transactional execution of a wavefront, the LDS unit serializes all local memory accesses so that, at a given time, a memory bank is accessed by only one work-item. Parallel accesses to different banks do not present conflicts, as the banks have different address ranges. Assuming a multi-bank arrangement, conflict detection proceeds in two steps:

(1) *Intra-bank conflict detection:* conflicts are detected for memory accesses within a bank. The conflict detection mechanism works in parallel for all memory banks. This step is responsible of updating TCM, setting to 1 the bits for those work-items involved in a conflict.
(2) *Inter-bank conflict communication:* once a conflict is detected in a memory bank, it is communicated to the rest of banks in order to remove the shadow memory entries allocated for the conflicting work-item. This is accomplished through the TCM, avoiding the need of an expensive broadcast communication. TCM informs to each memory bank which work-items detected conflicts (bits set to 1). For each one of these work-items, all the backups are restored and the associated shadow memory is cleared.

We have designed two strategies for intra-bank conflict detection (inter-bank conflict communication is common for both approaches).

Directory-Based Conflict Detection (DCD). In order to detect conflicts, the DCD mechanism checks the ownership information associated to the memory location being accessed. Valid bit V, required to differentiate empty and non-empty entries, is stored in vector registers. The number of V bits required is equal to the number of words allocated in each bank (see Fig. 4(b)). Provided that N words are allocated, $N/32$ vector registers are needed to store the V bits. Depending on the result of the check, three actions may occur (see Table 1(a)):

(1) *First (new) access:* the shadow memory entry has no owner associated (valid bit V is 0). A copy of the current value of the memory location is stored in the corresponding shadow memory entry and its owner is set to the work-item that made the access (now V is set to 1).
(2) *Repeated access:* the owner of the shadow memory entry is the accessing work-item. If the access is a read, the value in memory is returned. If it is a write, the memory is updated.
(3) *Conflict:* the owner of the shadow memory entry is a different work-item than the one that made the access. TCM is updated to mark this conflict, setting to 1 the bit of the work-item accessing to memory. In addition, the backup values of the accessing work-item are restored and all ownership entries in the shadow memory for WI are deleted.

DCD is a simple and precise approach for detecting conflicts, but at the cost of an additional local memory access to check the ownership records. Note that this mechanism cannot filter out read-read conflicts.

Table 1. Conflict detection using DCD (a) and SMDCD (b). WI is the accessing work-item, o-WI is other work-item, "0/1" means 0 or 1. "Abort" means the following actions: restore backup for WI, delete WI ownership entries and set TCM[WI] = 1.

Current State		**Mem.**	**Next State**		
Owner	**V**	**Operat.**	**Owner**	**V**	**Action**
Not set	0	Read or Write	WI	1	back up value; read or write mem.
WI	1	Read or Write	WI	1	read or write memory
o-WI	1	Read or Write	o-WI	1	conflict; abort

(a)

Current State			**Mem.**	**Next State**			
Owner	**S**	**M**	**Operat.**	**Owner**	**S**	**M**	**Action**
Not set	1	1	Read	WI	0	0	read memory
			Write	WI	0	1	back up value; write memory
WI	0	0/1	Read	WI	0	0/1	read memory
			Write	WI	0	1	write memory
WI	1	0	Read	WI	1	0	read memory
			Write	WI	1	0	Conflict (R→W); abort
o-WI	0/1	0	Read	o-WI	1	0	read memory
			Write	o-WI	0/1	0	Conflict (R→W); abort
o-WI	0	1	Read	o-WI	0	1	Conflict (W→R); abort
			Write	o-WI	0	1	Conflict (W→W); abort

(b)

Shared-Modified DCD (SMDCD). The DCD mechanism can be improved by adding two state bits per memory location to the ownership records: the S bit, set to 1 if multiple work-items accessed the location, and the M bit, set to 1 if the location has been written. These bits replace the valid bit (V) (see Fig. 4(b)) and permit to filter out read-read conflicts. In this case, provided that N words are allocated per memory bank, $N/16$ 32-bit vector registers are used to store this information. The new mechanism is called *Shared-Modified Directory-based Conflict Detection* (SMDCD).

The case of both state bits set to 1 at the same time is used to encode the "not set" (i.e., $V = 0$) owner state. This way, when starting a transaction (TX_Begin), both S and M are set to 1. For each transactional access to local memory, the SMDCD mechanism carries out the actions specified in Table 1(b). Accessing a memory location for the first time sets the owner in shadow memory and performs a backup of the current memory value if the access is a write (bit M permits to distinguish between reads and writes). A read access to memory location owned by a different work-item is allowed as long as M is 0. These accesses set the S bit to 1. If M is, however, 1, a conflict is detected (read after write). A write access is allowed only if the owner is the accessing work-item and the memory location was not accessed by another work-item (bit S set to 0). These accesses set the M bit to 1. Otherwise, they are considered conflict (write after read, or write after write).

4 GPU-LocalTM Modeling

The implementation of GPU-LocalTM requires changes to the GPU microarchitecture. We have implemented these changes using the Multi2sim 4.2 simulation framework [11] which supports the AMD Southern Islands family of GPUs. These changes introduce memory and latency overheads in the microarchitecture.

Latency Overhead. The TX_Begin and TX_Commit instructions are modeled as scalar instructions with an extra cycle of latency to manage the EXEC and TCM masks. Accesses to shadow memory are modeled as local memory accesses, plus an extra cycle used to manage the state bits.

Storage Overhead. Storage resources required in GPU-LocalTM are taken from those available in the CU. The amount of local memory available per work-group depends on the size of the shadow memory. If the user requests N words to store local variables, the shadow memory allocates another N words for backups and $N/4$ words for the ownership records (see Fig. 4). As the physical amount of local memory is 64KB, the maximum value of N is 29126 bytes and $N/4$ is 7282 bytes. This represents and overhead of ∼56% of the total local memory space. Vector registers are used to store the state bits. In the case of DCD, we need to store a V bit per word, requiring a maximum of 228 registers This supposes ∼0.3% of 65536 available. In the case of SMDCD, the number of registers needed doubles. The 4 TCMs required for a work-group (one per wavefront) use 8 scalar registers (two 32-bit registers for a 64-bit TCM, ∼0.4% of 2048 available). GPU-LocalTM may require to use the full amount of physical memory (64KB) for memory-demanding workloads, reducing potential concurrency. GPU-LocalTM is designed with the principle of not adding extra memory resources and to be fully configurable: no TM-dedicated memory needs to be added and the amount memory available is not affected if no TM support is needed. Furthermore, the programmer (or compiler) can opt for a lock implementation if no resources are available for TM support, and the runtime can assign new work-groups to a different CU to improve concurrency.

5 Evaluation

We have designed eight TM benchmarks to evaluate GPU-LocalTM in specific scenarios. All the experiments execute a single work-group with the maximum number of work-items allowed (i.e., 4 wavefronts of 64 work-items each, for a total of 256 work-items). The benchmarks are implemented in 3 different versions: a TM version, a fine-grained locks (FGL) or atomics version, and a third version serializing the critical section. In addition, each application uses two different inputs to test different levels of contention: high contention (HC) and low contention (LC). Table 2 summarizes the descriptions of these workloads.

Note that the HT, IT, DB and QU implementations using atomics are simple and the programming effort is comparable to the use of TM. However, GA, KM, GC and VA require extra lock management for FGL (17%, 10%, 42%, and 22% of the total code, respectively). The DB and QU applications are prone to conflicts and are designed to stress the TM to understand the possible sources of overhead (i.e., they test the TM beyond its expected capabilities). The serialization of the critical section is implemented by delegating the work of the whole work-group to a single work-item.

Table 2. Characteristics of the applications used for evaluation

Bench.	Description	Bench.	Description
HT	Inserts elements in a hash table, searching for the desired position. Features short and read-only transactions	IT	Similar to HT, but uses an index to point to the desired position. Features short and read-modify-write transactions
VA	The Vacation workload from the STAMP [9] suite, adapted and evaluated for inputs that modify from 2–4 elements. Features long transactions with a low probability of conflict	GA	Genetic algorithm used to solve an optimization problem searching for the best solution by combining a set of possible solutions. Features long and read-modify-write transactions
GC	Decentralized Graph Coloring algorithm. Features read-only transactions	KM	Implementation of the K-Means clustering algorithm. Features long transactions with multiple memory accesses
DB	Simulates an in-memory database composed of multiple IT tables. Features multiple memory accesses	QU	Simulates the queue and dequeue operations on a concurrent queue. Features short transactions with a high probability of conflict

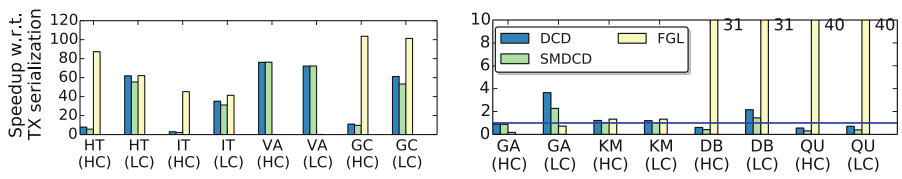

Fig. 5. Speedup w.r.t. TX. serialization (higher is better).

Speedup. Figure 5 presents the speedup achieved by the two different conflict detection strategies (DCD and SMDCD), fine-grained locks (FGL), and when serializing the critical sections. Performance is relative to serialized execution. In general, both DCD and SMDCD have similar performance. The exception are these applications with read-modify-write and read-only transactions that do not benefit from the SMDCD features. For the first set of applications (HT, IT, VA, and GC) both TM solutions and FGL outperform serial execution. The exception is VA when using FGL: the overhead of lock management is too high and such algorithm is not suitable for the use of fine-grained locks in a SIMT architecture. In the case of low contention scenarios, the performance of GPU-LocalTM is similar to FGL for applications such as HT and IT, and is in the same order of magnitude for GC. The second set of applications (GA, KM, DB, and QU) present a different scenario. As in VA, the extra lock management required for GA results in low performance when using FGL. For KM, GPU-LocalTM and FGL perform similar and close to the serial execution. The reason is that only 10% of the code of KM is able to take advantage of TM or FGL execution. DB and QU are challenging scenarios for GPU-LocalTM. The following metrics help to explain the reasons of their low performance.

Execution Breakdown. Figure 6 shows the execution breakdown using the two implementations of GPU-LocalTM. In all the scenarios, most of the overhead is introduced during the memory operations due to conflict detection and version management. As DCD aborts transactions on read-read conflicts while SMDCD waits until one of the memory operations is a write. Thus, in some cases, the overhead of SMDCD is larger as these conflicts are detected later. The overhead of TX_Begin and TX_Commit instructions is low and almost unnoticeable.

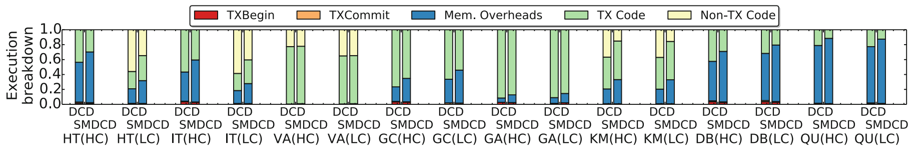

Fig. 6. Normalized execution breakdown.

Commit Ratio. Figure 7 shows the ratio of transactions committed over transactions started. In general, DCD and SMDCD conflict detection algorithms offer similar commit ratio. In the case of GA(LC), as transactions perform read-modify-write operations on the same memory location, DCD has some advantage over SMDCD as conflicts are detected during the read operation. GA, KM, DB, and QU suffer of a low commit ratio, harming performance (see Fig. 5). Future research will reduce the overhead of conflict detection in order to minimize the impact in performance of applications with a high probability of conflict.

Serialization Mechanism Evaluation. Figure 8 shows the percentage of transactions that proceed in transactional, WfS and WgS modes. Both DCD and SMDCD have similar results. We observe that many transactions (up to 90% in

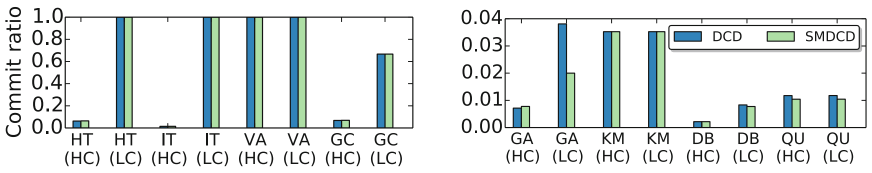

Fig. 7. Commit ratio (higher is better).

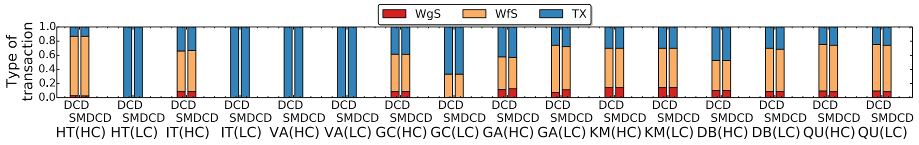

Fig. 8. Normalized transaction execution mode.

HT with high contention) need to make use of the serialization mechanism. The reason is that most of the conflicts continue to appear after a transaction retry due to lockstep execution.

6 Conclusions

In this paper we present GPU-LocalTM as a hardware TM for GPU architectures that focuses on the use of local memory. GPU-LocalTM is intended to limit the amount of additional GPU hardware needed to support TM. We propose two alternative conflict detection mechanisms targeting different types of applications. Conflict detection is performed per-bank, ensuring scalability of the solution. We find that for some applications the use of TM is not optimal and discuss how to improve our implementation for better performance. Furthermore, GPU-LocalTM introduces a serialization mechanism to ensure forward progress.

Acknowledgements. This work has been supported by projects TIN2013-42253-P and TIN2016-80920-R, from the Spanish Government, and P11-TIC8144 and P12-TIC1470, from Junta de Andalucia.

References

1. AMD: Southern Islands series instruction set architecture (2012)
2. Cederman, D., Tsigas, P., Chaudhry, M.T.: Towards a software transactional memory for graphics processors. In: 10th Eurographics Conference on Parallel Graphics and Visualization (EG PGV 2010), pp. 121–129 (2010)
3. Chen, S., Peng, L.: Efficient GPU hardware transactional memory through early conflict resolution. In: 22nd International Symposium on High Performance Computer Architecture (HPCA 2016) (2016)

4. Fung, W.W.L., Aamodt, T.M.: Energy efficient GPU transactional memory via space-time optimizations. In: 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2013), pp. 408–420 (2013)
5. Fung, W.W.L., Singh, I., Brownsword, A., Aamodt, T.M.: Hardware transactional memory for GPU architectures. In: 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2011), pp. 296–307 (2011)
6. Harris, T., Larus, J., Rajwar, R.: Transactional Memory, 2nd edn. Morgan & Claypool Publishers, San Rafael (2010)
7. Herlihy, M., Moss, J.E.B.: Transactional memory: architectural support for lock-free data structures. In: 20th Annual International Symposium on Computer Architecture (ISCA 1993), pp. 289–300 (1993)
8. Holey, A., Zhai, A.: Lightweight software transactions on GPUs. In: 43rd International Conference on Parallel Processing (ICPP 2014), pp. 461–470 (2014)
9. Minh, C.C., Chung, J., Kozyrakis, C., Olukotun, K.: STAMP: Stanford transactional applications for multi-processing. In: IEEE International Symposium on Workload Characterization (IISWC 2008), pp. 35–46 (Sept 2008)
10. Shen, Q., Sharp, C., Blewitt, W., Ushaw, G., Morgan, G.: PR-STM: priority rule based software transactions for the GPU. In: Träff, J.L., Hunold, S., Versaci, F. (eds.) Euro-Par 2015. LNCS, vol. 9233, pp. 361–372. Springer, Heidelberg (2015). doi:10.1007/978-3-662-48096-0_28
11. Ubal, R., Jang, B., Mistry, P., Schaa, D., Kaeli, D.: Multi2Sim: a simulation framework for CPU-GPU computing. In: 21st International Conference on Parallel Architectures and Compilation Techniques (PACT 2012) (2012)
12. Xu, Y., Wang, R., Goswami, N., Li, T., Gao, L., Qian, D.: Software transactional memory for GPU architectures. In: Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO 2014), pp. 1:1–1:10 (2014)