Hardware support for Local Memory Transactions on GPU Architectures

Alejandro Villegas Ángeles Navarro
Rafael Asenjo Oscar Plata
Universidad de Málaga, Andalucía Tech.
Dept. Computer Architecture, 29071 Málaga, Spain
{avillegas, angeles, asenjo, oscar}@ac.uma.es

Rafael Ubal David Kaeli
Department of Electrical and Computer Engineering
Northeastern University, Boston, MA
{ubal, kaeli}@ece.neu.edu

Abstract

Graphics Processing Units (GPUs) are popular hardware accelerators for data-parallel applications, enabling the execution of thousands of threads in a Single Instruction - Multiple Thread (SIMT) fashion. However, the SIMT execution model is not efficient when code includes critical sections to protect the access to data shared by the running threads. In addition, GPUs offer two shared spaces to the threads, local memory and global memory. Typical solutions to thread synchronization include the use of atomics to implement locks, the serialization of the execution of the critical section, or delegating the execution of the critical section to the host CPU, leading to suboptimal performance.

In the multi-core CPU world, transactional memory (TM) was proposed as an alternative to locks to coordinate concurrent threads. Some solutions for GPUs started to appear in the literature. In contrast to these earlier proposals, our approach is to design hardware support for TM in two levels. The first level is a fast and lightweight solution for coordinating threads that share the local memory, while the second level coordinates threads through the global memory. In this paper we present GPU-LocalTM as a hardware TM (HTM) support for the first level. GPU-LocalTM offers simple conflict detection and version management mechanisms that minimize the hardware resources required for its implementation. For the workloads studied, GPU-LocalTM provides between 1.25-80X speedup over serialized critical sections, while the overhead introduced by transaction management is lower than 20%.

Categories and Subject Descriptors C.1.4 [*Computer System Organization*]: Processor Architectures - Parallel Architectures

Keywords GPU, Hardware Transactional Memory, Local Memory Transactions

1. Introduction

Graphics Processing Units (GPUs) have been adopted as hardware accelerators for massive data-parallel applications due to their support of Single Instruction-Multiple Thread (SIMT) execution, the availability of hundreds of cores and the inclusion of a highly-banked memory that offers high memory bandwidth. To support general purpose processing in GPUs, technologies such as CUDA [9] and OpenCL [8] were developed. In contrast to CPUs, GPUs are organized as a set of highly multi-threaded SIMT cores. Using OpenCL terminology, a SIMT core is called a *Compute Unit* (CU). A thread is a *work-item* and a *work-group* is group of work-items that must be scheduled to a single CU. Various work-groups can be mapped to the same CU, but only if the hardware resources (such as registers and memory) can be partitioned among them. Work-groups do not share such resources. Inside the work-group, work-items are grouped in different wavefronts, that are scheduled to use the different resources of the CU. An important component in GPU programming frameworks is the memory model. This model consists of two separate address spaces: (1) a *local memory*, shared by the work-items belonging to the same work-group, and (2) a *global memory*, visible by all the work-items, as well as the host CPU. The local memory is usually used as a scratchpad due to its low latency. This way, all work-items belonging to the same work-group can communicate via a fast local memory. Communication between work-items mapped to different work-groups must be done via a slower global memory. Each individual work-item also has a private memory space, which is typically mapped to registers.

In general, multithreaded data-parallel applications resort to explicit synchronization to avoid race conditions when accessing shared data. Supporting efficient mutual exclusion in a SIMT architecture poses a challenge. Several solutions have been proposed or implemented. A widely-adopted approach is to delegate the execution of critical sections to the host CPU. However, CPU-GPU communication and data transfer consume a large number of clock cycles. This issue impacts performance when the critical section protects shared data stored in the local memory, as the data has to be moved back to the global memory, and then transferred to the host CPU memory space. An alternative solution is to use barriers to control access to the critical section, allowing only one of the work-items to make progress. This solution causes a complete serialization of the execution of work-items, and can significantly impact performance. Finally, a third option is to use atomic operations to implement locking, assuring exclusive access to the critical section. We can define coarse-grained locks that produce the same serialization safety as barriers or fine-grained locks, while being less prone to programming errors, deadlocks and livelocks.

Transactional Memory (TM) [6] 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, but the results are the same as if they were executed serially. Many TM systems have been proposed in the last two decades for multicore CPU architectures [5]. Recently, TM solutions for GPUs have started to appear in the literature, both software [2, 7, 11] and hardware [4].

As modern multicore CPUs started to add hardware TM support, extending similar support for GPUs will be an important design decision. This is especially true given recent develops in programming models for heterogeneous architectures, as they will demand highly parallel applications to be uniformly deployed across all available computational engines (both CPU cores and GPU compute units). To date, TM proposals for GPUs only consider the GPU architecture and use global memory for synchronization. This approach offers scalable solutions for thousands of threads, but at the cost of significant hardware, performance and energy overheads [4] (in fact, Fung et al. [3] developed new techniques to improve the inefficiencies found in their earlier work [4]). In contrast, our approach is different and proceeds in two steps. The first step is to offer lightweight and fast hardware TM support within work-groups. More specifically, we will manage concurrent transactional work-items that belong to the same work-group. The idea is to take advantage of the fact that those work-items share the same local memory. We can leverage this fact and design a very efficient TM system, introducing little new hardware and reducing transaction handling overhead. The aim is to offer the programmer a method to write data-parallel applications that require fast local (restricted to a work-group) synchronization, comparable to fine-grain locking, but without many of lockings issues. This local TM support, however, can be combined with other standard techniques for synchronization across work-groups. Support for transactions belonging to different work-groups using global memory corresponds to a second step, and currently it is work-in-progress and beyond the scope of this paper.

This paper describes GPU-LocalTM. Our proposal for TM is a simple and effective hardware system for coordinating transactions within work-groups executing on a single SIMT core (CU). The main contributions of GPU-LocalTM are the following:

- It is designed to provide fast hardware-based TM support for local memory transactions, minimizing the amount of extra hardware.
- It offers an extension to the GPU ISA to define transactions, allowing the specification of new constructs in a high-level language (such as OpenCL), or at a library level, to use transactions.
- It implements version management and conflict detection techniques per memory bank, ensuring future scalability and exploitation of data locality and coalesced memory accesses.

The rest of the paper is organized as follows: Section 2, gives an introduction to GPU architectures and the Multi2sim simulation framework, which is used to implement GPU-LocalTM. Section 3 discusses the implementation of GPU-LocalTM and its main features. Section 4 discusses how GPU-LocalTM is modelled using the Multi2sim simulation framework. To assess GPU-LocalTM, we propose some benchmarks whose description and evaluation is done in Section 5. Sections 6 and 7 discuss the related work, conclusions and future work.

2. Background

GPUs, when used as general-purpose processors, consist of several SIMT cores or CUs. A CU has several functional blocks, such as multiple vector units, a scalar unit, branch units, a local data share (LDS) unit and a memory interface to global memory. Registers are allocated in the scalar and vector units. The LDS is a low-capacity, low-latency and highly-banked memory that is able to service multiple coalesced memory requests. Programmers define compute kernels which can be dispatched to run on the GPU. A kernel is divided into work-groups that are assigned to a single CU. However, a CU may process multiple work-groups. Each work-group preallocates its registers and the amount of local memory needed in the LDS. The work-group is composed of several work-items, that are arranged in wavefronts of a fixed size. The work-items of a wavefront execute in lockstep. Inside the CU, a wavefront is always assigned to the same vector unit. This preallocation of resources helps to reduce the overheads of a context switch among wavefronts.

As the baseline GPU to implement GPU-LocalTM, we have chosen the AMD's Southern Islands. This architecture supports a maximum of 256 work-items per work group, and 64 work-items per wavefront (i.e., 4 wavefronts per work-group). Each of the 32 CUs contains 4 SIMD units. The LDS unit, which contains the local memory, deserves special attention as it is heavily involved in the GPU-LocalTM design. AMD's GPU architecture includes 64KB of LDS (viewed by the programmers as local memory) distributed across 32 banks. Consecutive local memory words map to consecutive LDS banks. The LDS unit is in charge of managing this local memory. Whenever a wavefront requests to access the local memory, the LDS unit schedules the access of the wavefront, allowing up to 32 coalesced (i.e., without bank conflicts) accesses simultaneously. Uncoalesced memory accesses are serialized by the LDS unit. Coalesced/uncoalesced address detection by the LDS unit ensures that, at a given point of time, each of the 32 banks is accessed by only a single work-item at time.

The control flow of the SIMT programming model is implemented by using two masks managed by hardware and the compiler. The *execution mask* (EXEC) indicates, per wavefront, the work-items that are running or disabled. The *vector comparison mask* (VCC) stores the results of certain arithmetic operations, similar to the "vectorized" Z flag used in some CPUs. For the Southern Islands architecture, the size of these masks are of 64 bits, as the number of work-items per wavefront is 64. The VCC and EXEC masks are mapped into the general purpose scalar registers.

The implementation of GPU-LocalTM requires changes to the GPU architecture. We implemented these changes using the Multi2sim 4.2 hardware simulator. Multi2sim [10] is a simulation framework for CPU-GPU heterogeneous computing that includes models for superscalar, multithreaded, and multicore CPUs, as well as GPU architectures. It features a functional simulator which executes the instructions found in a given binary, and a detailed timing simulator which features the full hardware pipeline for multiple architectures providing a cycle-precise simulation. The current version of Multi2sim 4.2 provides the implementation of the AMD's Southern Islands architecture, as described above.

3. GPU-LocalTM

GPU-LocalTM is a hardware TM system designed to allow for explicit synchronization of work-items through local memory using transactions. With this proposal, we aim to reuse the existing memory resources present with a CU and minimize the amount of new hardware required.

GPU-LocalTM extends the GPU ISA with two new instructions, *TX_Begin* and *TX_Commit*. Even though individual work-groups could execute both instructions, they are designed to work at a wavefront granularity, since wavefronts are the schedulable unit of the CU resources. This way, while a wavefront is executing a transaction, other wavefronts can be executing different transactions or a non-transactional code region. All the local memory operations executed by a wavefront inside a transaction are considered transactional. When the wavefront reaches the TX_Commit instruction,

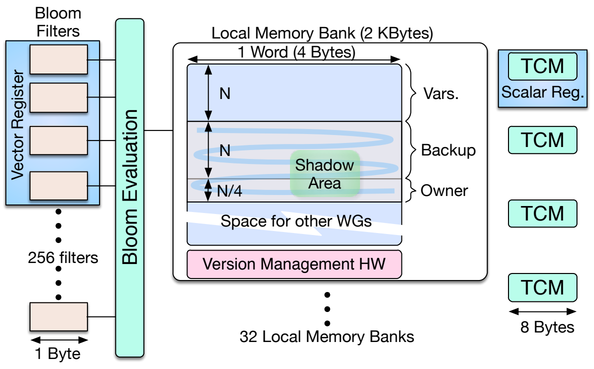


Figure 1: Hardware in the LDS unit to implement GPU-LocalTM.

```
//Before the transaction
s_tx_begin  //Begin transaction
v_cmp_gt_f32  v2, v1 //VCC=(v2>v1)
s_mov_b64  s0, exec //s0=EXEC
s_and_b64  exec, exec, vcc //EXEC=EXEC and VCC
s_cbranch_vccz  label0 //Jump to "else" if no VCC
//Code "if" with possible conflicts:
//{...}
label0:
s_andn2_b64  exec, s0, exec //EXEC=s0 and not(EXEC)
s_cbranch_execz  label1 //Jump to end if VCC
//Code "else" with possible conflicts:
//{...}
label1:
s_mov_b64  exec, s0 //EXEC=s0
s_tx_commit //End transaction
```

Figure 2: Example of an if-then-else statement inside a transaction implemented using the AMD's Southern Islands ISA.

for those work-items within the wavefront that were not able to complete their memory accesses due to conflicts, the transaction rolls-back and re-executes from the TX_Begin instruction. Otherwise, the wavefront continues execution of the next instruction after the TX_Commit. The reasons to consider all local memory accesses inside a transaction as transactional are: 1) to be able to reuse functions that can be called either inside and outside a transaction, 2) to minimize the number of new ISA instructions needed, easing the hardware and reducing compiler complexity, and 3) to simplify memory consistency, as there could be no transactional and non-transactional accesses to the same local memory address inside transactions. The main drawback of this proposal is that the programmer is not able to optimize the code by differentiating transactional and non-transactional memory accesses.

Barrier instructions are not allowed within a transaction, as the transaction itself causes wavefront divergence, and not all the work-items may reach the barrier. Furthermore, the TX_Begin and TX_Commit instructions do not impose an implicit barrier, delegating the use of a barrier to the programmer. The reason is that there might be transactions used in divergent paths of the code, and not all the work-items within the work-group will execute the transaction. In addition, we consider that nested transactions are *flattened*.

Figure 1 shows a local memory bank in the LDS unit, with the extra hardware added by GPU-LocalTM. The following subsections discuss the main characteristics of GPU-LocalTM: the transactional execution model, memory-bank based version management and conflict detection.

3.1 Transactional SIMT Execution Model

The GPU SIMT execution model relies on the existence of execution masks. The EXEC execution mask contains a bit for each work-item within the wavefront indicating if it is active (or inactive). The VCC mask stores the results of a vector comparison instruction. By combining the EXEC and VCC masks, compilers implement conditionals, loops and other features using the GPU ISA.

However, modifying these masks inside a transaction can create inconsistencies. If a work-item conflicts in a conditional statement, and the EXEC mask is modified by the TM system, it can be activated by another instruction later in the code, despite the fact it has to be quiesced until the transaction is re-started. For this reason, we implemented a *transaction conflict mask* (TCM) per wavefront, which stores a 1 for work-items that have a detected memory conflict, or 0 otherwise. This new TCM mask is mapped to a scalar register (see Figure 1), as well as the existing EXEC and VCC masks. The SIMT execution model must be modified to only allow the execution of a work-item whose EXEC bit is set to 1 and whose TCM bit is set to 0. The main advantage is that the use of this mask only requires hardware changes in the execution model, but the compiler and ISA remains unmodified (i.e., the compiler does not need the TCM mask to implement conditionals with the EXEC and VCC masks, or use new instructions to manage TCM). When a wavefront executes the TX_Begin instruction, the TCM mask is set to 0, and whenever a memory conflict is detected by a work-item, its corresponding bit in TCM is set to 1. During the execution of the TX_Commit instruction, if the TCM mask contains at least one bit set to 1, the transaction is re-started for those conflicted work-items, by copying TCM into EXEC, clearing TCM and returning to the TX_Begin instruction. If, otherwise, all the bits of TCM are 0 by the end of the transaction, this means that every work-item was able to complete without conflicts, and thus the wavefront can commit the transaction and continue normal execution. As a work-group consists of several wavefronts, each wavefront has its own TCM, EXEC and VCC masks and they can be in different stages of execution (i.e., while some wavefronts may execute a transaction, others can be executing non-transactional code). Figure 2 shows an assembly code level example of an if-then-else statement inside a transaction.

Progress is not guaranteed by this method, as some of the work-items may conflict an unlimited number of times, rolling back the transaction indefinitely. This situation can be detected at the end of the transaction if the TCM mask does not change in two consecutive transaction re-executions. In such cases, we propose an execution model with a two-level serialization process in order to provide forward progress guarantees. Whenever two consecutive re-executions of a transaction finish with the same TCM mask, the transaction is retried assuming that all of the conflicting work-items but one are going to conflict again during the next execution. To implement this policy, instead of clearing the TCM at the beginning of the next retry, only one of its active bits is reset. This results in the execution of only one of the work-items within the wavefront during the next transaction execution. We refer to this mechanism as *wavefront serialization mode*. Nevertheless, even executing only one work-item per wavefront, we can observe conflicts with work-items in other wavefronts, leading to the same situation. In case a transaction ends in the wavefront serialization mode and the TCM mask has not changed since the last execution, *work-group serialization mode* is enabled. In this mode, only the current wavefront executes a transaction, just as in wavefront serialization mode. Transactions executing in other wavefronts are aborted and stalled at the TX_Begin instruction until the current wavefront finishes its transaction. This procedure guarantees progress, as executing a single work-item within the work-group does not conflict with other work-items.

Table 1 shows an example of the execution with the different serialization modes and mask management for a particular trace of the sample code in Figure 2. For simplicity, we evaluated wave-

Instr	EXEC	TCM	TCM OLD	Mode	Conflicts
1	1111	-	-	NORMAL	-
2	1111	0000	-	TX	-
5	1100	0000	-	TX	-
10	0011	0000	-	TX	-
13	0011	0011	-	TX	WI2 and WI3
15	1111	0011	-	TX	-
16	0011	0011	-	TX	-
2	0011	0000	0011	TX	-
5	0000	0000	0011	TX	-
10	0011	0000	0011	TX	-
13	0011	0011	0011	TX	WI2 and WI3
15	0011	0011	0011	TX	-
16	0011	0011	0011	TX	-
2	0011	0001	0011	WF Serial.	-
5	0000	0001	0011	WF Serial.	-
10	0011	0001	0011	WF Serial.	-
13	0011	0001	0011	WF Serial.	-
15	0011	0001	0011	WF Serial.	-
16	0001	0001	0011	WF Serial.	-
2	0001	0000	0001	TX	-
5	0000	0000	0001	TX	-
10	0001	0000	0001	TX	-
13	0001	0000	0001	TX	-
15	0001	0000	0001	TX	-
16	0001	0000	0001	TX	-

Table 1: Example of the execution model applied to the code in Figure 2. The double lines separate transaction retries. It is assumed that register v1 contains the work-item ID (0, 1, 2 or 3) and register v2 contains the value 2. This example considers 4 work-items.

fronts consisting of 4 work-items (WI0, WI1, WI2 and WI3) and only the lines where the EXEC and TCM masks are modified are presented. Line 2 denotes the beginning of the transaction. The comparison instruction in Line 3 updates the VCC mask to set the bits corresponding to the work-items that evaluated the condition as one. In our example, this condition evaluates that the work-item ID is smaller than 2 (i.e., VCC sets the bits corresponding to WI0 and WI1 to 1, leaving the bits corresponding to WI2 and WI3 as 0). The EXEC mask is updated in Line 5 with the information stored in the VCC mask. The instructions in Lines 6 and 11 implement jumps to the "else" section of the code and the end of the conditional. It is assumed that execution of the code in the "if" leg of the conditional (Line 8) does not cause any conflicts, but the code in the "else" leg of the conditional (Line 13) presents a conflict between work-items WI2 and WI3. This conflict has to be resolved via wavefront serialization. The instruction in Line 15 restores the original EXEC mask after the conditional statement. The TX_Commit instruction in Line 16 denotes the end of the transaction. The TCM mask is updated whenever a conflict is detected in the execution of the code in Line 13. As two consecutive transaction re-executions were started with the same TCM mask, the execution is then serialized inside the wavefront, allowing the transaction to finally commit one of its work-items.

3.2 Version Management

Version management handles the memory and register updates during transactional accesses to local memory. In the specific case of the simulated architecture, local memory is distributed across 32 banks. In GPU-LocalTM, we propose a bank-level version management mechanism. The main advantages of this proposal are: i) it provides scalable performance as GPUs add more memory banks, and ii) version management and conflict detection can be executed in parallel in different banks.

GPU-LocalTM follows an eager schema: all the accesses to local memory or registers within a transaction have to save the old values and write the new values in their actual locations. In case of a transaction abort, the stored values must be restored.

To backup memory values, a local memory area called *shadow memory* is used. The shadow memory consists of a table of <owner, value> pairs, which is private to each memory bank. This table must have enough room to store backups for all the local memory variables declared within the kernel allocated in each bank. The advantage, as we will see, is that it requires simple management. The main drawback is that it allocates room for every variable in local memory, consuming a large amount of memory resources in local memory. Variables in local memory are statically allocated by the compiler in consecutive positions [1, 10]. In our proposal, shadow memory is also allocated by the compiler, adding space to backup all the variables in local memory and an extra byte to store the index of the 256 possible owners. The organization of this table is as follows: 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 additional bytes are reserved to store the owner. Given this layout, when we encounter a memory access to location k, we will have stored a backup value at position k+N, and its owner is stored k bytes after the backup. By adopting this model, the hardware required to backup a memory value and store its owner is minimal, as only the calculation of two offsets is required. In addition, capacity conflicts are avoided, as each memory location is ensured to have space for its backup. Lastly, the use of an offset does not force us to save <owner, address, value> triples, but only the pair <owner, value> (i.e., the address can be calculated using the offset). Figure 1 shows the organization of a local memory bank for version management.

Vector registers, which are used individually by each work-item, are backed up using other vector registers. This set of backup registers are called *shadow registers*. We propose the architectural implementation of the vector registers in pairs: two registers are connected and a signal can enable a copy of the contents of the first register to be written to the second whenever a backup is needed. Kernels that execute transactions must preallocate these registers for its work-items. Kernels that do not include transactions are not affected by this new layout of the registers.

Scalar registers are used to store common information for an entire wavefront, such as a loop index. When this occurs, the use of scalar registers creates inconsistencies when a transaction aborts inside a loop. For instance, the loop index of a while loop might be updated inside a transaction. As this index is shared by committed and aborted work-items, the value of this shared scalar register is inconsistent. For this reason, the compiler must promote the use of private vector registers instead of scalar registers to implement loops in kernels that execute transactions.

Register backup is performed at the beginning of the transaction. Memory backups are preformed in each individual memory access. Whenever a conflict is detected, a work-item must restore the backup values from the shadow registers and shadow memory.

3.3 Conflict Detection

Conflict detection implements a strategy that detects conflicting transactional accesses to the same memory location, and determines if a work-item must continue execution or abort its transaction. During an access to memory by a wavefront, the LDS unit serializes memory accesses so that at a given time, each bank is accessed by only one work-item. Parallel accesses to different memory banks do not present conflicts, as the memory banks have different address ranges. Signatures based on Bloom filters are used to detect conflicts. Signatures are private per work-item and per bank (i.e., there are 256 Bloom filters per bank). In our initial design, a single signature is used for both, reads and writes, and an 8-bit Bloom filter performs the hash for conflict detection. Since there is a large number of filters, 8-bit Bloom filters are used to consume lower memory resources to store them. For the hash function, we have found that a simple $address\%8$ operation work well on GPU local memory. The reason is that usually GPU programmers allocate variables in a coalesced way: in a given bank, the access of an address by a work-item is usually followed by the access to the following address by the same or a different work-item. The $address\%8$ hash minimizes the number of false conflicts caused when consecutive addresses are referenced. Bloom filters, stored in vector registers, are accessed by the memory banks when required. As an example, let us consider that the work-item 3 intends to access the memory location 35. This memory location is stored in the bank $35\%32 = 3$, and inside this bank it is stored in the position $35/32 = 1$. Then work-item 3 must check the bit $1\%8 = 1$ of the 256 Bloom filters assigned to bank 3. Bloom filters may result in false positives, which are treated as conflicts.

Conflict detection operates in 3 stages when a local memory access is issued by a work-item. This mechanism proceeds simultaneously in each one of the 32 memory banks, as follows:

1. *Fast conflict detection*. Bloom filters are used to detect conflicts during the first stage, since the query is fast. When issuing a local memory operation, a work-item checks its address against the 256 Bloom filters assigned to the current bank. The outcome of this stage can be: i) a conflict (the Bloom filter assigned to a different work-item returns a positive), ii) a new access (no Bloom filter returns a positive), and iii) no conflict (only the Bloom filter assigned to the current work-item returns a positive, but we cannot determine if it is a new access or not). In case i), the backed up values associated with the current work-item must be restored and the entries in the shadow memory table must be cleared. TCM is updated to mark that the current work-item had a conflict while performing a memory access. In this case, we can directly proceed to the third stage of the conflict detection mechanism (conflict broadcast). In case ii), the work-item performs a backup with the old value and sets its ownership in the shadow memory. Next, we proceed to the third stage of the conflict detection mechanism. In case iii), the current work-item claims ownership of the memory address. In this case we must proceed to the second stage of the conflict detection mechanism to distinguish a new access from a previous access to the same address.
2. *Ownership detection*. This stage is carried out only if the first stage returned a no-conflict outcome. In this case, the current work-item must examine the shadow entry for the current address. If the owner is the same, no action is required. Otherwise, the work-item must proceed as if it was a new access (i.e., the work-item performs a backup and sets its ownership in the shadow memory).
3. *Conflict broadcast*. Once a conflict is detected, it must be broadcast to all memory banks in order to clear the shadow memory entries that the current work-item has previously allocated in other memory banks. During this stage, each memory bank loads the TCM mask, which has been modified in parallel by all the memory banks during the fast conflict detection stage. For each work-item with its bit set to 1 in TCM, all the backups are restored and associated shadow memory is cleared.

This conflict detection system uses TCM both to update memory banks with information about a conflict without requiring an expensive broadcast bus, and to inform the execution model of a conflict. One drawback is that conflict detection using Bloom filters suffers from false conflicts, that can be avoided if we implement ownership detection. However, the hardware needed to detect a conflict using Bloom filters is much simpler than the one required to examine the shadow memory. It is a scalable and faster solution, because the evaluation of multiple Bloom filters can be performed in parallel.

Feature	Value
Compute Units (CU)	32
Vector Registers per CU	65536
Scalar Registers per CU	2048
SIMD Units per CU	4
SIMD Lanes	16
LDS Size per CU	65546 bytes
LDS Banks	32
LDS Latency	2 cycles

Table 2: Relevant features of the AMS's Southern Islands GPU implementation on Multi2sim 4.2 used by GPU-LocalTM.

4. GPU-LocalTM Modeling

The baseline architecture used to implement GPU-LocalTM is the AMD Southern Islands GPU provided with the Multi2sim 4.2 simulation framework. Table 2 shows key features of this architecture.

4.1 Functional Simulation

We add the TX_Begin and TX_Commit instructions to the AMD's Southern Islands ISA supported in Multi2sim, and we also update the wavefront execution model. The conflict detection and version management are implemented as a common part for all existing LDS instructions for the wavefronts running in transactional mode.

4.2 Timing Simulation

The detailed simulator has been used to model the clock cycles needed by each GPU-LocalTM feature. The TX_Begin and TX_Commit instructions are modeled as scalar instructions as they affect the whole wavefront. In addition to the standard cycles needed to execute a scalar instruction, we need to calculate the latency added by the version management process performed by these instructions.

The TX_Begin instruction has to evaluate the EXEC and TCM, and backup the vector registers. As explained previously, we assume that vector registers are implemented in pairs, and the backup can be done in one cycle. Evaluating EXEC, TCM and performing the backup of the registers can be done in parallel, which adds a cycle. Starting a transaction in the work-group serialization mode requires clearing the ownership records of the shadow memory. We model that the version management hardware added to support the GPU-LocalTM to be able to modify an entry in memory every clock cycle. This way, in each bank, we need to add an extra cycle per memory location accessed by the wavefront. As the banks are cleared in parallel, the number of cycles to add to the TX_Begin instruction includes the number of cycles that the slowest bank needs to clear all its entries. Summarizing, the TX_Begin instruction requires at least 1 extra cycle, and as many cycles as needed to clear the most occupied bank, in case of a work-group serialization.

Similar reasoning is used to model the TX_Commit instruction. This instruction needs one cycle to clear the Bloom filters of the committing work-items. In addition, the ownership records of the shadow memory must be cleared, adding one cycle per entry. As with the TX_Begin instruction, we evaluate the number of cycles needed in the different banks, and assume that the TX_Commit instructions need as many cycles as the slowest bank.

We need to consider the extra memory accesses required to query the Bloom filters and modify entries in the shadow memory. As with the TX_Begin and TX_Commit instructions, we assume that the Bloom filters can be evaluated in a clock cycle. Management of the shadow entries depends on the outcome provided by the Bloom filters, so these operations cannot be overlapped. During conflict detection, case i) in stage 1 (a conflict) requires clearing the ownership records for the conflicting work-item. This takes as many cycles as the needed to clear the slowest memory bank. If the conflict detection results in case ii) (a new access), the actions to be performed include a backup of the memory location and modification the ownership records. As only one access to memory per clock cycle can be done, we assume that these modifications add 2 cycles to the memory instruction latency. If the conflict detection results in case iii) (no conflict), we add the same number of cycles as we did in the new access case (in case it was a false positive), or only a single cycle to check the ownership records (in case of a second access to the same memory location). In addition, the last step of a local memory access requires the broadcast of the conflicts across the different banks to clear the ownership records. As in the previous stage, we account for this latency which is equal to the latency of the slowest bank.

4.3 Hardware Resources and Limitations

As previously discussed, the resources needed by a work-group on a given CU must be preallocated before starting execution. This preallocation limits the execution of work-groups whose requirements exceed the available resources. These limitations are: 1) the number of work-groups assigned to the same CU, 2) the number of vector and scalar registers needed, and 3) the amount of local memory allocated. For instance, using the data from Table 2, a work-group cannot run on this architecture if the number of vector registers needed by all its work-items exceeds 65,536.

GPU-LocalTM introduces new constraints to resource allocation. The use of vector registers to map Bloom filters reduces the number of available vector registers to 8192 (256 Bloom filters per bank, and 32 banks per CU) per work-group. Each wavefront requires its own TCM mask, which is mapped into scalar registers. Each work-group is composed by 4 wavefronts, so the number of vectors registers available is reduced by 4 per running work-group. The amount of local memory available for the work-group depends on the size of the shadow memory. In the simulated architecture, with 64KB of local memory, the use of GPU-LocalTM reduces the amount of local memory available for a single work-group to a maximum of 29,127 bytes. The same amount of memory is allocated for backups for the speculative accesses, and 7282 bytes are used for the ownership records.

GPU-LocalTM is designed to be fully configurable via the compiler and runtime. Memory resources required by the shadow memory, shadow registers and masks are allocated by the compiler. Hence, kernels that do not require transactional execution do not suffer from the resource limitations introduced by GPU-LocalTM.

5. Evaluation

We propose 3 benchmarks with different inputs in order to analyze the different features of GPU-LocalTM.

5.1 Benchmarks

5.1.1 Hash Table

In the Hash Table (HT) benchmark, each work-item inserts its own ID in a shared hash table consisting of N buckets. Each bucket is sized so all the inserts will fit. Once the work-item calculates the target bucket with a simple hash function, we try to find an empty location to store its ID. A transaction comprises reading a location, checking if it is empty and inserting the corresponding value (i.e., a read-modify-write set of operations on a single memory address). To test scenarios with a range of conflict probabilities, we conducted experiments doubling the number of buckets from 2, and increasing the number up to 256. The scenario HT2 contains 2 buckets and presents a high probability of conflict, while the scenario HT256 contains 256 buckets with no conflicts.

5.1.2 K-Means

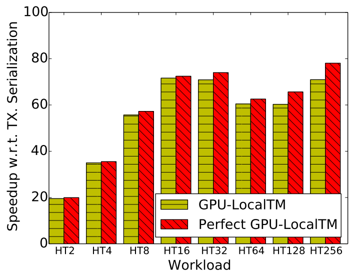
In the K-Means (KM) benchmark, for each point belonging to a set of N points, we calculate its closest center from a set of K centers. Then, each center is modified based on the geometrical center of its closest points. This is done for a set number of iterations, or until the system converges. In our implementation, each work-item processes a point. Calculating the closest center is executed in parallel, as there are no dependencies. The transactional region comprises updating the values of the centers, as several work-items will try to modify the same center speculatively. For KM, we considered having 256 3-dimensional random points (one per work-item within the work-group) and a single iteration. This is a memory bound application that features read-modify-write operations on multiple memory addresses within a transaction. For this benchmark, we executed a range of experiments, modifying the number of centers in order to test GPU-LocalTM under scenarios with different conflict probabilities and to change the number of memory locations accessed. The experiment KM2 considers 2 centers, and includes a high probability of conflicts. The number of centers is doubled in each experiment up to KM256. This last scenario presents a small probability of conflict, but the number of addresses shared among the different transactions is high (producing a high false positive rate in the Bloom filters).

5.1.3 Genetic Algorithm

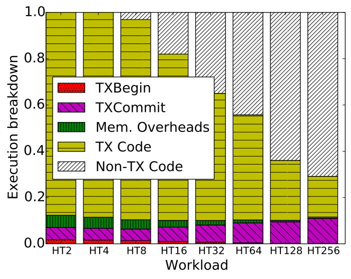
The Genetic Algorithm (GA) benchmark solves the knapsack problem. This problem aims to fit objects characterized by a weight and a price into a bag with a maximum capacity. The goal is to maximize the value of the bag. We simplified the problem, considering that the ratio weight/price is the same for all the objects. Thus, our goal is fit the bag as closest as possible to its maximum capacity. GA starts from a set of N random solutions to this problem. Each work-item randomly picks two solutions from this set and evaluates their fitness, as the solution closest to the bag capacity. A crossover operation is performed, transforming the solution with the lowest fitness into the solution with the highest fitness. Then, both solutions are added to the set, replacing the previous ones. After a certain number of iterations, most of the solutions will represent the objects that fit in the bag. In our experiments, we considered a set with a different number of initial solutions and 8 possible objects, in order to analyze transactions with a higher number of operations than the other benchmarks. The set of solutions is composed by integers, whose 8 least significant bits represent a mask. If an object is included in the solution, the mask will have its corresponding bit set to 1; otherwise the bit will be 0. A transaction comprises reading two random solutions from the set, running the previous algorithm, and replacing the solutions.

5.1.4 Performance Evaluation

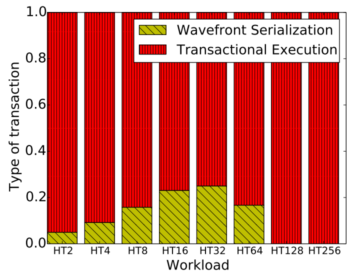
To assess the potential for our proposal, we implemented 2 versions of GPU-LocalTM. The first version is GPU-LocalTM as described in this paper. The second version, *Perfect GPU-LocalTM*, has the same behavior as GPU-LocalTM, but avoids the extra clock cycles needed by conflict detection and version management. This version allows us to measure the overheads introduced by GPU-LocalTM.



(a) Speedup w.r.t. TX. Serialization

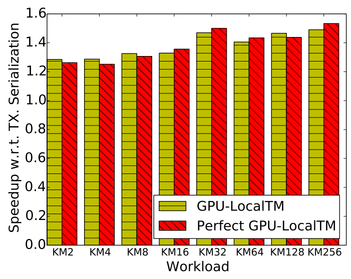


(b) Normalized Exec. Breakdown

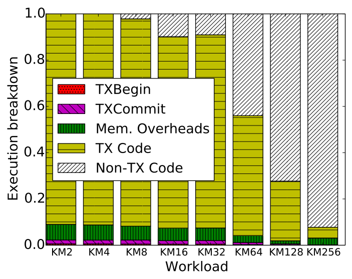


(c) Transaction Type

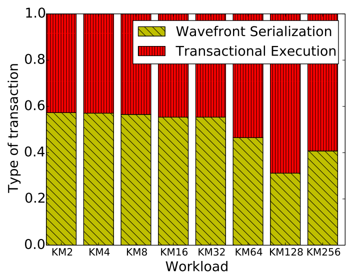
Figure 3: Hash Table benchmark



(a) Speedup w.r.t. TX. Serialization

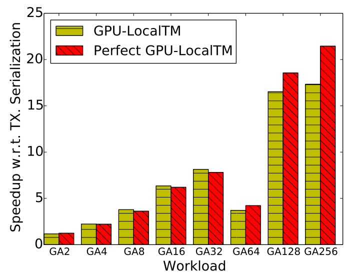


(b) Normalized Exec. Breakdown

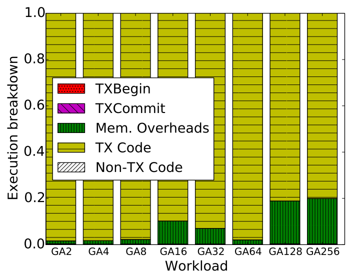


(c) Transaction Type

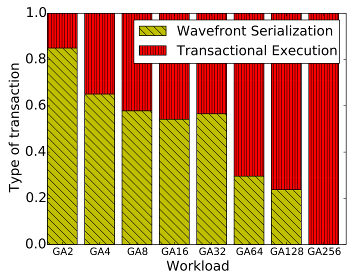
Figure 4: K-Means benchmark



(a) Speedup w.r.t. TX. Serialization



(b) Normalized Exec. Breakdown



(c) Transaction Type

Figure 5: Genetic Algorithm benchmark

For each benchmark, we analyzed the following metrics:

- *Speedup*. We compare the performance of the two versions of GPU-LocalTM as compared against a version of the benchmarks where each transaction is serialized and executed by a single work item (*TX. Serialization*). This version of the code requires a programming effort similar to the use of GPU-LocalTM and presents similar performance as the use of coarse-grained locks.
- *Execution Breakdown*. The execution breakdown measures the portion of the total number of clock cycles spent in the following stages: native code outside the transactional region (*Non-TX Code*), native code inside the transactional region (*TX Code*), memory access overheads caused by conflict detection and version management (*Mem. Overheads*), and the overheads caused by the TX_Begin and TX_Commit instructions.
- *Transaction Type*. A transaction that conflicts many times enters either the wavefront or work-group serialization mode. This metric measures the percentage of the transactions that needed to be serialized over the total amount of transactions. We only represent wavefront serialization since work-group serialization never occurs in these benchmarks.
- *Commit Ratio*. This metric computes the quotient *commits/aborts*. We do not plot this since we found uniform values for all the benchmarks, but is important metric and we will discuss it with the other results.

5.2 Results

5.2.1 Hash Table

In all the scenarios, the speedup obtained by GPU-LocalTM (Figure 3a) significant as compare to *TX Serialization* execution mode, and close to the the ideal speedup obtained by the optimized GPU-LocalTM scenario. This implies that GPU-LocalTM introduces low overhead in the presence of small read-modify-write transactions.

The execution breakdown (Figure 3b) shows that the overhead introduced by GPU-LocalTM does not exceed 16% of the execution time of the native GPU code. The overhead due to the TX_Commit instructions appears in all the scenarios, as some extra cycles are required in order to clear the shadow memory entries. In the HT2 scenarios, this is a scenario with a high probability of conflict, so we observe that most of the cycles are wasted executing inside a transaction due to an increasing number of retries. In addition, the overhead resulting from accesses to memory starts to become noticeable due to the cycles spent in conflict detection and version management.

Figure 3c shows that many transactions serialize as the probability of conflict increases. This is also related to the commit ratio, which ranges from 4.9% in HT2 up to 100% in HT256. The relative number of transactions that have to be serialized is smaller in experiments with a higher probability of conflict (e.g., HT2), as compared to experiments with an medium probability of conflict (e.g., HT32). However, the absolute number of transactions that have to be serialized is 189 out of 3801 in HT2 (4.9%) and 9 of 36 in HT32 (25%), which is a result consistent with the expected probability of conflict.

5.2.2 K-Means

Figure 4a shows the speedup obtained for K-Means. In scenarios with a higher number of centers (e.g., from KM64 to KM256) the Bloom filters are too small to store all the memory accesses performed by all the work-items, incurring a high number of false positives. The speedup obtained is relatively small (1.5X), but we should consider that the portion of code that benefits from the transactional execution consumes about 5% of the total cycle count and represents less than 10% of the complete code. We can also observe that, in some cases, GPU-LocalTM outperforms its perfect version. The reason is that with the extra cycles associated with the TX_Begin and TX_Commit instructions add some divergence that avoids future conflicts. This scenario also occurs in the GA benchmark.

In Figure 4b we also observe that the overhead introduced during conflict detection and version management by the TM system is lower than 10% of the total cycles in the benchmark.

In addition, as each one of the K 3-dimensional centers is stored in a single bank, the probability that different dimensions from different centers produce a false positive is high. As a result, many of the transactions serialized their execution (Figure 4c). Our analysis shows that about 39% of the memory accesses result in false conflicts when checking the Bloom filters. This also causes the commit ratio to never grow higher than 4%, except in the scenario KM256, where it rises to 15%.

5.2.3 Genetic Algorithm

Figure 5a shows the speedup obtained over serial execution for the GA benchmark. As we can observe, we see an improvement in the speedup as the conflict probability decreases. However, we should note that in some cases, GPU-LocalTM outperforms the ideal version, and these cases deserve special attention. In these cases, the cycles added by GPU-LocalTM introduced divergence in the wavefront due to the conflict detection mechanism during the LDS operations and the TX_Begin and TX_Commit instructions. These divergences are the explanation of why interactions between work-items that occurred in the perfect GPU-LocalTM version are absent in our non-ideal version. As a result, the number of conflicts decreases with the number of wavefront serializations. In the particular case of GA8, the GPU-LocalTM version has to serialize 134 wavefronts, while the perfect GPU-LocalTM version had to serialize 177 times. In this case, we also observe differences in the number of conflicts: while GPU-LocalTM suffers a total of 2578 conflicts, the perfect version experiences 3126 conflicts.

In Figure 5b we observe that most of the execution time is spent inside a transaction and most of the overhead is due to conflict detection and version management. This benchmark also presents a low commit ratio: the best scenario, GA256, has a commit ratio of 15%. The ratio decreases, reaching about 1% in GA2. The commit ratio is related to the breakdown. As the commit ratio is low, the overhead of the conflict detection and version management is significant as compared to the TX_Begin and TX_Commit instruction overhead. This trend is also related to the type of transaction (Figure 5c). In scenarios with a higher probability of conflict and a lower commit ratio, a higher percentage of transactions are serialized.

6. Related Work

Cederman *et al.* propose two STMs for graphics processors [2], but focused on conflicts produced between different thread-blocks (work-groups) and not considering possible interactions of the single threads (work-items). Their two proposals are a blocking STM that prevents another transaction to commit if a committing one is updating its values to memory, and a non-blocking STM that allows transactions to use values that committing transactions have to write to memory.

Another STM is GPU-STM [11] proposed by Xu *et al.*. GPU-STM works at the granularity of a work-item, includes a hierarchical validation based on a timestamp, followed by a value-based validation. GPU-STM is aimed at transactions that make use of global memory, while GPU-LocalTM focuses on the use of local memory. In addition, GPU-LocalTM is designed to minimize the required hardware resources, as well as to keep transaction overhead down as low as possible.

Holey *et al.* also propose the use of STM at a work-item granularity and focus on the global memory [7]. They propose an Eager STM (ESTM), which detects conflicts eagerly, a Pessimistic STM (PSTM), that simplifies conflict detection by not treating reads and writes differently, and an Invisible Read STM (ISTM) that validates reads during the commit of a transaction. Our proposal is similar to their PSTM, which in many cases was the best performing of the STMs that they proposed.

KILO TM [3, 4] is a hardware TM for GPU architectures that operates at a thread-level and considers global memory. KILO TM describes commit units that perform a value-based conflict detection, in contrast to GPU-LocalTM that uses Bloom filters, located within the local memory banks to speedup conflict detection. In addition, GPU-LocalTM proposes the automatic serialization of conflicting transactions, which is not present in KILO TM. While KILO TM aims to applications that require synchronization at global memory, GPU-LocalTM aims to those synchronizing at local memory. Both memory spaces serve at different purposes and, thus, a comparison is not possible. However, in the future, GPU-LocalTM will be extended to provide support for transactions at global memory. In that moment, a comparison between both approaches is unavoidable.

7. Conclusions and Future Work

In this paper we present GPU-LocalTM as a hardware TM for GPU architectures focusing on the use of local memory. GPU-LocalTM is intended to add minimal additional hardware to an existing GPU and minimizes transaction overheads. Our experiments show that GPU-LocalTM outperforms the execution of kernels that rely on serialization to solve memory conflicts. In addition, GPU-LocalTM introduces a serialization mechanism to ensure progress within a transaction.

Future research will focus study alternative organizations of the Bloom filters and explore alternatives that use local memory as a shadow memory, in order to decrease the amount of memory resources needed by GPU-LocalTM. In addition, future work will consider the global memory space in our transactional memory mechanism.

Acknowledgments

Thanks to the student Oliver Fishstein from Northeastern University for providing a base implementation of the genetic algorithm benchmark. In addition, the authors would like to thank the anonymous reviewers for their useful comments.

References

[1] AMD. *Southern Islands Series Instruction Set Architecture*. 2012.

[2] D. Cederman, P. Tsigas, and M. T. Chaudhry. Towards a software transactional memory for graphics processors. In *10th Eurographics Conf. on Parallel Graphics and Visualization (EG PGV'10)*, pages 121–129, 2010.

[3] W. W. L. Fung and T. M. Aamodt. Energy efficient GPU transactional memory via space-time optimizations. In *46th Ann. IEEE/ACM Int'l. Symp. on Microarchitecture (MICRO'13)*, pages 408–420, 2013.

[4] W. W. L. Fung, I. Singh, A. Brownsword, and T. M. Aamodt. Hardware transactional memory for GPU architectures. In *44th Ann. IEEE/ACM Int'l. Symp. on Microarchitecture (MICRO'11)*, pages 296–307, 2011.

[5] T. Harris, J. Larus, and R. Rajwar. *Transactional Memory, 2nd*. Morgan & Claypool Publishers, USA, 2010.

[6] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *20th Ann. Int'l. Symp. on Computer Architecture (ISCA'93)*, pages 289–300, 1993.

[7] A. Holey and A. Zhai. Lightweight software transactions on GPUs. In *43rd Int'l Conf. on Parallel Processing (ICPP'14)*, pages 461–470, 2014.

[8] Khronos. *The OpenCL Specification. Version 2.0*.

[9] NVIDIA. *NVIDIA CUDA Programming Guide*.

[10] R. Ubal, B. Jang, P. Mistry, D. Schaa, and D. Kaeli. Multi2Sim: A Simulation Framework for CPU-GPU Computing. In *21st Int'l. Conf. on Parallel Architectures and Compilation Techniques (PACT'12)*, 2012.

[11] Y. Xu, R. Wang, N. Goswami, T. Li, L. Gao, and D. Qian. Software transactional memory for GPU architectures. In *Ann. IEEE/ACM Int'l. Symp. on Code Generation and Optimization (CGO'14)*, pages 1:1–1:10, 2014.