A Semantically Aware Transactional Concurrency Control for GPGPU Computing

Qi Shen

School of Computing Science

University of Newcastle

A thesis submitted for the degree of

Doctor of Philosophy

October 2017

Acknowledgements

I would like to thank my supervisor, Dr Graham Morgan for his invaluable guidance and support throughout all my time at Newcastle. This research and thesis would not be possible without his kind encouragement and motivation that Graham provided.

I also would like to thank my colleagues and friends in Newcastle University, who have helped me in various ways during my PhD life. Especially Dr. Craig Sharp, who helped me a lot at the beginning of my study and gave me a lot of advices.

Certainly, I would like to thank my family, my father and mother. I would not have a chance to carry on the research without their unconditional and continuous support. Finally, I would like to express my utmost thanks to my loving wife, for her support and patience during my research.

Abstract

The increased parallel nature of the GPU affords an opportunity for the exploration of multi-thread computing. With the availability of GPU has recently expanded into the area of general purpose programming, a concurrency control is required to exploit parallelism as well as maintaining the correctness of program. Transactional memory, which is a generalised solution for concurrent conflict, meanwhile allow application programmers to develop concurrent code in a more intuitive manner, is superior to pessimistic concurrency control for general use. The most important component in any transactional memory technique is the policy to solve conflicts on shared data, namely the contention management policy.

The work presented in this thesis aims to develop a software transactional memory model which can solve both concurrent conflict and semantic conflict at the same time for the GPU. The technique differs from existing CPU approaches on account of the different essential execution paths and hardware basis, plus much more parallel resources are available on the GPU. We demonstrate that both concurrent conflicts and semantic conflicts can be resolved in a particular contention management policy for the GPU, with a different application of locks and priorities from the CPU.

The basic problem and a software transactional memory solution idea is proposed first. An implementation is then presented based on the execution mode of this model. After that, we extend this system to resolve semantic conflict at the same time. Results are provided finally, which compare the performance of our solution with an established GPU software transactional memory and a famous CPU transactional memory, at varying levels of concurrent and semantic conflicts.

Contents

Chapter 1

Introduction

1.1 Parallel Computing

As sequential programs can be executed more efficiently when processor frequency increases, processing frequency scaling can provide an easier option to enhance sequential programming for more difficult tasks. However, from the beginning of the 21st century, parallel computing has become more important as it is the only way to achieve the best performance from multi-processor platforms when limitations have been placed on frequency scaling. Until the time of writing, computing platforms with increasing numbers of processor cores are the mainstream on the market and in design.

1.1.1 Benefit of Parallelism

In recent decades, the study of parallel computing has become ever more important. The main reason for this is parallel computing offers numerous advantages over sequential computing in the following ways:

1. *Efficiency*: As many processors are available for modern computers, in theory, using multiple processing elements for a task can shorten the time taken to compute a solution compared to a single processor. Parallel computing is the only way to utilise all computation resources for solving problems, because otherwise additional processors would be suspended and would give no advantage.

2. *Performance*: Many problems can be distributed and solved with multiple processing elements in ways faster than with sequential computing. Furthermore, some large projects are too complex to be solved on a single computation unit, as memory is limited. Plus, some problems are parallel in nature and would too much time to be solved using a sequential algorithm. Both types of problem can be more easily solved and performance improved using parallel computing.

3. *Tolerance*: As opposed to sequential computing, which is halted when an error occurs since there is only one processor operating, parallel computation can be designed to operate in fault-tolerant systems, especially by using lockstep systems which operate the same instructions in parallel. Replication methods are also applied in hard real-time systems using parallel computing.

1.1.2 Classifications of Parallelism

In identifying the different patterns of sequential characteristics and parallelism which exist in computing science. Flynn's taxonomy [13] classifies any system according to two independent dimensions: the instruction stream and data stream. Each dimension can be divided into a single or multiple case. Therefore, the four possible categories are as follows:

1. *SISD*: a single instruction stream and single data stream is commonly used in serial computing. Only one each instruction and data stream is being used during one clock cycle.

2. *SIMD*: single instruction stream and multiple data stream systems are used in parallel computing. This is the most suitable solution for problems characterized by a high degree of regularity, such as graphics processing.

3. *MISD*: multiple instruction stream and single data stream systems are also used in parallel computing. This type of parallel computing is rarely used in practise.

4. *MIMD*:multiple instruction stream and multiple data stream systems are again used in parallel computing. Most current multi-core computing falls into in this category. Execution can be deterministic or non-deterministic in this pattern.

1.2 Multi-Threading

Multi-threading is an execution model in which programmers can create multiple independent processes within a single operating system process. At the time of writing, multi-threading with shared memory is arguably the most popular model of concurrent programming.

Multi-threading is not only a model of execution, but also has a tremendous impact on program design [36]. When multi-threading is introduced, as threads can access shared data in different permutations from one to another, determinism is lost. This loss of determinism complicates the prediction of the results of concurrent programs. Moreover, non-deterministic execution also introduce inconsistencies for shared data, and leads to greater complexity in program debugging.

Compared to multi-thread programming, sequential programming can benefit from determinism, which means that multiple executions of the same program will obtain the same result. This mechanism is lost in concurrent programming, and concurrency control is demanded to reintroduce determinism for multi-thread programming.

1.2.1 Concurrent Conflict and Concurrency Control

Race conditions are a well-known type of concurrency error that may result in inconsistent data in multi-threaded programs. Figure 1.1 provides an example with two threads (T1 and T2) which read the same location (x) in shared memory and increase the value of that shared memory by 1. The possibility of a race condition means that there could be two different final values for memory location x. In the top scenario, the memory value holds the value 2 when there is no interference between T1 and T2. However, in the bottom scenario, interleaved

thread execution produces a final memory value of 1.

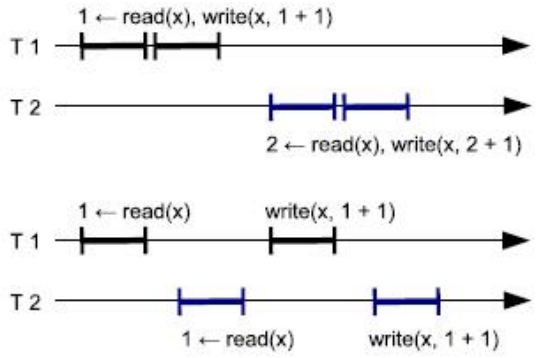


Figure 1.1: A race condition

In this example the possibility of a race condition has removed the element of determinism from the program; it is no longer possible to say what the value of memory x will hold. Concurrency control aims to implement mechanisms which prevent this loss of determinism. The two most prominent approaches to implementing concurrency control are pessimistic and optimistic concurrency control. Pessimistic concurrency control (PCC) protocols aim to prevent non-determinism conflict occurring in advance, normally by using blocking synchronization such as locks or semaphores. Optimistic concurrency control (OCC) detects conflicts after they have occurred and then implements steps to correct such conflicts [35], for example using abortable transactions.

Blocking synchronisation by mutual exclusion is a common approach to pessimistic concurrency control and transactional memory is a popular optimistic method for concurrency control. As there exist many kinds of applications in which concurrency control is needed, no single approach is likely to be the best solution for all scenarios. However, whether a method is pessimistic or optimistic, the main aims of concurrency control mechanism are:

1. *Correctness*: This means that the concurrency control technique should prevent the logical inaccuracy of any program it is applied to.

2. *Efficiency*: This means that the concurrency control technique should not

impose a large burden on the execution platform. Resources for concurrency control should be used efficiently.

1.2.2 Mutual Exclusion

Mutual exclusion is a foundational and frequently used technique in pessimistic concurrency control [10]. Mutual exclusion is normally used when memory access is restricted, but becomes extremely difficult to implement when software becomes large and the data interacts a lot. The ultimate principle of mutual exclusion is that only one thread at a time can write-access data in a critical section. The determinism can be reintroduced, as only a single thread can access a critical region at one time. Programming with mutexes also provides user-defined conditions that can deal with the more complex coordination of threads. There are some disadvantages to this approach as well, such as starvation in which situation some threads repeatedly locking critical sections, blocking access to them for others. Besides this, the programmer must be aware of the risk of deadlock, when two or more threads wait for each others' resources indefinitely. In addition, livelock can be introduced if threads continually respond to each other's action instead of progressing their own execution.

1.2.3 Transactional Memory

Transactional memory is a modern and well-known example of optimistic concurrency control. In the transactional memory paradigm, threads can access shared data within the execution of a transaction. However, modifications to shared data are not made permanent until the transaction has completed successfully. These changes are made into shared data only if no concurrency conflict is detected, or otherwise the potential changes must be aborted and the transaction will restart. The main benefit of transactional memory compared to mutual exclusion is that dead lock can be avoided [22]. This is because all threads can operate without interference from other threads. Transactional memory implementations, however, typically require an overhead in excess of that required by a more simple blocking approach, which may lead to the degradation of performance when contention is high.

To achieve data consistency, transactions must be

1. *Atomic*: this means that all operations within a transaction must be treated in their entirety. All the operations can be committed or none at all.

2. *Consistent*: this means that all data have to meet the validation rules declared by applications.

3. *Isolated*: this means that each thread has its own mirror during a transaction and all changes to shared data cannot be observed by other threads until the transaction is committed.

However, the *Durability* is removed from ACID rules for databases because it cannot be maintained in physical violate event such as power loss. In Transactional Memory for concurrency, all transactions can only have their effects remain after commits if the task could be done without any physical damage.

1.2.4 Semantic Conflict

Semantic conflict is another circumstance that prevents a transaction from executing but which is not caused by data interference. It occurs due to the presence of certain application conditions. When transactions were originally introduced for distributed database applications, the order of transactions was an inconsequential matter since operations tended to be independently. However, when implementing software transactional memory for multi-thread programming, transactions turn into tightly coupled and coordination becomes a significant issue. For example, a transaction may process a withdrawal from a bank account with insufficient funds, while another transaction tries to deposit money into the same bank account. In these circumstances, the deposit transaction must precede the withdrawal transaction in order for progression to occur. In these kinds of conditions, resolving semantic conflicts without overloading a huge amount of programming work (for example, programmers need to schedule all transactions by themselves.) is necessary.

1.3 GPGPU Computing

1.3.1 The demand of Transactional Memory on the GPU

Graphics processing units (hereafter GPU) were first designed for rendering graphics in the early 1980s. At that time, GPU could only be used for accelerating the creation of images in a frame buffer and then output to a display. When dealing with computer graphics and image processing, as they are highly parallel structures in nature, the GPU processes blocks of visual data in parallel and never encounters data interference. So there is no demand for concurrency control.

However, the availability of GPU has recently expanded into the area of general purpose programming, giving rise to a new genre of applications known as General Purpose GPU (hereafter GPGPU). The principle benefit of using the GPU is the relatively high degree of parallel computation available compared to the CPU. Furthermore, programming APIs such as CUDA have grown in sophistication with every new advancement in GPU design. As such, GPGPU programmers now have at their disposal tools to enable them to write complex and expressive applications which can leverage the power of modern GPUs [55]. As with multi-threaded applications on the CPU, GPGPU applications require synchronization techniques to prevent the corruption of shared data. As has long been experienced in the domain of CPU computing, correctly synchronizing multiple threads is a difficult task to implement without introducing errors such as deadlock and livelock. To compound matters, the high number of threads available on modern GPUs means that contention for shared data is an issue of greater potential significance than on the CPU where the number of threads is typically much lower. To have a tool which can free the application programmer from implementing complex concurrency control, a transactional memory is a notable technique to use. Although transactional memory may have a overhead than customized concurrency control solution, achieving a generalized transactional approach to the GPU is yet important.

1.3.2 GPU Architecture and Memory Hierarchy

A GPU has more transistors devoted to processing to data caching and flow control. With this architecture, the GPU is optimized for data parallel computations (single instruction multiple data). In addition to the high degree of threads available, groups of GPU threads execute as part of a 'warp'. Threads belonging to the same warp share the same instruction counter and thus execute the same instruction in a 'lock-step' fashion [42]. In addition to the risk of high contention given the high number of threads, deadlock and livelock are possible because threads of the same warp cannot coordinate their accesses to locks as they can on the CPU.

To use the GPU properly, a general idea about memory hierarchy of the GPU is necessary. There are three access levels in the hierarchy of memory, as follows:

1. *Global Memory*: Global memory is off-chip memory, which means that access speed is slow, but all threads in a GPU kernel can access it.

2. *Shared Memory*: Every block has a piece of shared memory which is on-chip, and threads within one block are able to access it.

3. *Thread Local Memory*: Two kinds of thread local memory are involved in the GPU: registers and local memory. Registers are its fastest memory in the GPU with a restricted capacity, which can only be accessed by one thread. Local memory is an abstract memory which is allocated to global memory but one thread can only access its own local memory.

Sometimes using global memory for thread interaction between blocks is demanded. The most efficient way to use global memory is to obtain data from global memory and replicate it to local memory.

1.4 Thesis Contributions

This thesis addresses the scientific and engineering problems related to concurrency control in general purpose computational systems. A particular focus is to derive software solutions for GPU hardware architectures. The work presented

in the thesis is suitable for current commercial GPU hardware and future GPU hardware where core numbers may increase significantly.

The main contributions the Thesis provides are:

- A transactional memory implementation on the GPU that can take, as its batch input, arbitrarily ordered transactions and successfully execute them all without deadlock occurring, using a priority system for determining the contention resolution of locks. With the inherent priorities to sort threads order when manage conflicts during transaction time is optimal on GPU compare to previous work. Theoretically, this order of execution is a scalable and lock-free solution for real-time concurrent problems.

- A contention management system for the GPU that can handle semantic conflict at the application layer. The ordering of transactions can increase throughput and semantic correctness. For example, if the two operations of deposit and withdrawal on a shared empty bank account are ordered as withdraw first, then a semantic failure would occur. However, if the deposit was ordered before withdraw then this would be a correct semantic. Besides this, as GPU threads are executing in a lock-step fashion, it may introduce the possibility of live-lock. A re-ordering of a semantically aborted transaction can evade this. The proposed search is general purpose in nature, as the re-order is independent of execution or data structure.

- A comprehensive evaluative benchmarking of the above two contributions demonstrates the improvements they provide in terms of throughput and efficiency compared to similar works in this area. Experiments were devised to show the effectiveness and generalizability of the proposed approach.

In order to obtain a practical realisation of the mechanisms described, theory was implemented in a programmer-defined manner popular within transactional systems (begin/commit/abort). As the proposed approach is the first to use contention management in software transactional memory (STM) coupled with rescheduling semantically aborted transactions on the GPU, the proposed solution is compared with other STMs (without handling semantic conflict) on both CPUs and GPUs, which are the closest approaches to ours.

1.5 List of Publications

Portions of the work within this thesis have been documented in the following publications:

1. Q. Shen, C. Sharp, W. Blewitt, G. Ushaw, and G. Morgan *PR-STM: Priority Rule Based Software Transactions for the GPU* In: 21th International Conference on Parallel and Distributed Computing (Euro-Par), 2015

2. Q. Shen, Y. Gu, C. Sharp, G. Ushaw, G. Morgan *A Parallelised Contention Management Policy for the GPU* In: ACM Transactions on Parallel Processing, Under Review

1.6 Thesis Outline

The rest of this thesis is organised as follows.

Chapter 2 describes the background knowledge about the work carried out, and highlights the main contributions of the research.

Chapter 3 describes a model of the proposed system and discusses its advantages and disadvantages. Then the algorithm of the contention manager is presented.

Chapter 4 addresses semantic conflict in this system and an extra contention manager for it is introduced. After that, an implementation is described.

Chapter 5 provides a description of experiments which conducted to evaluate the performance of both systems. The PR-STM is compares to previous work and the PR-STM2 shows a unique ability for general purpose tasks.

Chapter 6 summarises and concludes the thesis and suggests some possibilities for further work.

Chapter 2

Background and Related Work

This chapter introduces concurrency control, and a number of frequently-used contemporary concurrency control models which are relevant to the research reported in this thesis. The problem of concurrency control and some basic solutions are introduced at the beginning of this chapter. After that, a summary is presented the different features of frequently used parallel computation platforms. At the end, some related work are presented such as research on software transactional memories on the CPU and recent concurrency control solutions on the GPU.

2.1 Concurrency Control

As discussed in Chapter 1, the race condition is one of the most important types of error which has to be resolved in parallel computing. For the sake of getting the correct results for concurrent programming, and dealing with the race condition as quickly as possible, concurrency control is introduced. This section discusses some basic concepts of concurrency control and then some general solutions to it are presented.

2.1.1 Correctness Criteria

To describe concurrency control, a definition of concurrent objects is first required. A concurrent object is any data structure or entity which provides some equivalent

behaviours to a sequential object. When discussing concurrent object, two factors which need to be considered are correctness and progress.

Correctness or memory consistency means that any interaction by threads parallel to the concurrent object should not introduce any inconsistency with the object. There are two levels of consistency which are important to address:

Linearizability – Linearizability means that all operations on a single concurrent object are atomic, provide a real-time guarantee and are composable [30]. So linearizability is about a single operation, a single object and operating in real-time.

Serializability – serializability means that the execution of a set of transactions over multiple data should be able to reorder in such a way that is equivalent to some sequential execution of the transactions [18]. So this concerns multiple operations, multiple objects and not operating in real-time.

Then when the property of progress is being discussed, it means that level of liveliness about the interactions on a concurrent object. Three frequently used levels are:

Obstruction-Free – this is the weakest natural non-blocking progress guarantee. It requires that any thread executes in isolation from obstructing threads, and it should complete its operations within a finite number of steps [29].

Lock-Free – this means that no thread is blocked indefinitely from an execution of another thread so that deadlock is avoided in this model.

Wait-Free – this is a technique where any operation by a thread should finish in a limited number of steps, and all algorithms could be implemented as wait-free [28]. It provides a stronger guarantee than lock-free progress but is more complicated to implement.

Although wait-free progress seems to be superior to lock-free solutions, the later is more efficient in many situations as extra problems need to be solved in the wait-free mechanism.

2.1.2 Categories of Concurrency Control

Pessimistic

Whenever blocking is the main way to achieve data consistency, the technique

can be classified as pessimistic concurrency control, where when a contention for shared data access arises, one thread may impede activities from other threads. It is called pessimistic because the approach assumes that the worst case will always occur with concurrency conflicts over shared data. So a pessimistic concurrency control technique would take whatever steps are necessary to prevent data access interference from taking place [12, 19].

Optimistic

Due to the major problem with pessimistic concurrency control which are reductions in performance and the danger of deadlock, another solution allows threads do modify data first, and then detect any concurrent interference which arises to interrupt consistency afterwards [6]. And as so this is on the opposite to a pessimistic approach, and it is called optimistic approach. In an optimistic approach, if no interference takes pace, then a modification made by the thread will carry on and be revealed to other threads [35]. However, if such interference occurs, it must be detected and all modifications should be rolled back. After aborting the modifications, the thread may attempt to repeat these modifications. Deadlock is impossible here as no blocking is used.

2.1.3 Basic Methods for Concurrency Control

Locking

Locking constructs were proposed by the early concurrent programmers. It was designed to prevent race conditions and still remains widely used. Basically, locking data means that only one thread would be allowed to access a piece of shared data at a time. Access to shared data may be sacrificed at times, but it can be guaranteed that locked data can be accessed or modified in a deterministic manner. The essential trade-off is that the reliability of data offsets the reduced parallel speed, as threads need longer average time to access shared data.

There are lots of approaches used to implement a lock strategy, however, when a thread confronts locked data, two primary approaches exist:

1. *Spinning*: a thread can repeatedly check the lock until it becomes available if the lock is expected to be held for only a short duration of time. This is also referred to *spinlock* or *busy waiting*;

2. *Blocking*: the waiting thread can also be suspended and the operating system can use a context switch to allow another thread to access the scheduler if the lock is expected to be held for a long duration of time. However, context switching is very expensive, and the decision should be made very carefully.

Most operating systems have specified operations to help in building locks (for example, non-interruptible critical sections) and locking has often been used for many years in diverse programming domains such as operating system programming [50]. As a result, lots of locking constructs have been implemented and are widely available for programmers to use, and they are supported by full documentations.

However, programmers using locks may encounter difficulties when dealing with sophisticated multi-threaded programs for which locks cannot be composed. For example, although the occurrence of race conditions on a single data structure can be prevented by mutual exclusion, threads may introduce deadlock when they require multiple locks. Developing an algorithm to avoid deadlock tends to be only suitable for particular software and cannot be transferred to other applications. Furthermore, proving that complex locking applications do not introduce deadlock into programs may take a lot of time.

In addition to the difficulty of implementing and proving sophisticated locking protocols, locking also encounters the problem of efficiency. Because of the pessimistic nature of the lock in blocking thread execution, programs with locking are often suffocate even when concurrent access to shared data would not cause inconsistency. As the fundamental principle of pessimistic approaches is the assumption that inconsistency will always arise if threads access the same shared data, locking applications can result in execution bottlenecks, especially when the number of threads increases.

Read-Write Locks

Different types of locks can be used when dealing with different levels of access rights to shared data. In this way, the potential bottlenecks can be reduced and the level of concurrency can be increased. Locks can be simply distinguished between *read* locks and *write* locks. A typical lock only allows single access to

a single shared piece of data, making the application inefficient if there are a lot of threads which only read from the shared data and they are all prevented, as read-only threads guarantee that there will not be any modification and there is no need to prohibit access [38].

Read-write locks allow the programmer to specify whether a thread intends to access a shared resource for reading or writing. As in its literal meaning, a thread can acquire a write lock which behaves like a typical lock when it wishes to write or modify the shared data, and it can grant exclusive access to the owner. A thread can also acquire a read lock which allows more threads to gain access to the shared data in parallel when it is only wished to read the shared data. This may improve the level of concurrency available to threads. However, as locks have an error-prone nature, this simple mechanism also introduces some possible errors:

1. As a write thread can only lock shared data when there is no read thread, it is possible that the write thread could be 'starved' when read threads are abundant. The write thread cannot gain access to the shared resource due to everlasting read requirements for the resource.

2. Using a queue for access to a shared resource or prioritizing a queue of write threads can solve the problem of starvation, but it may reduce the level of parallelism that the read-write lock was supposed to offer. The reason for this is that now all read or write threads have to perform coordinated queuing operations to assign priorities to the write threads.

Two-Phase Locking

In an execution schedule, when there are multiple locks used by multiple threads, non-serializable schedules are not guaranteed to be prevented. A serializable schedule means that when two or more threads contain read or write instructions, they can be reordered into an new schedule which is equivalent to a serial schedule [44](for example, where one thread's instructions all precede the others'). However, some instructions are in a causal order and cannot be reordered without infringing the results of previous instructions if one thread reads or writes to shared data which another thread has written to. Figure 2.1 shows an execution schedule which is not serializable. In this figure:

1. Thread 1 cannot precede Thread 2 as item A is written to by Thread 2 and then read by Thread 1.

2. Thread 2 cannot precede Thread 1 as item B is written to by Thread 1 and then read by Thread 2.

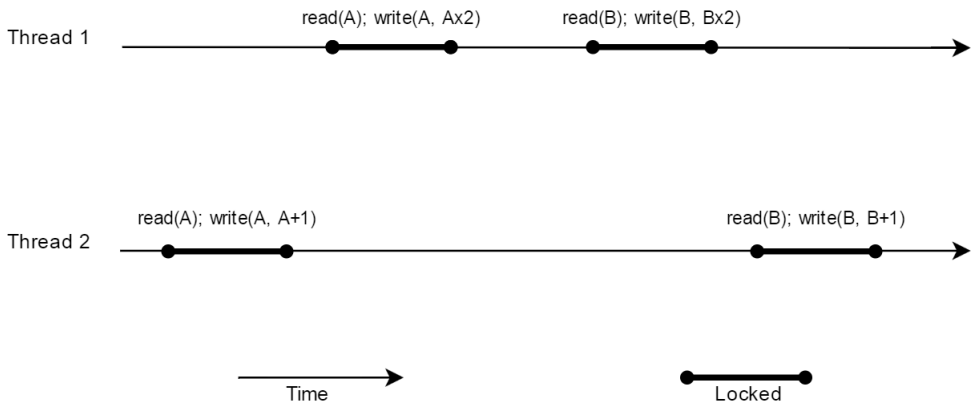


Figure 2.1: An Example of a Non-Serializable Schedule.

Reordering these instructions in a new order so that either thread executes strictly in serial is impossible. It is assumed that there is a constraint that shared data A and B must always hold the same values and there would be no solution for these threads to execute without violating the constraint because of the non-serializable schedule.

However, two-phase locking (hereafter 2PL) can resolve this problem without resorting to locking shared data unnecessarily, although it reduces the potential concurrency [16]. To utilize a 2PL, threads have to process an acquisition phase and a release phase (Figure 2.2 shows). A thread has to acquires all the locks it needs to guarantee exclusive access in the acquisition phase. If one thread has locked all necessary locks, then it can be able to access and modify shared data and subsequently release all acquired locks in the release phase. In such a way, the constraint on shared data A and B will not be violated, because Threads 1 and 2 have to initially lock both shared data and then process access and modification operations.

Time Stamps

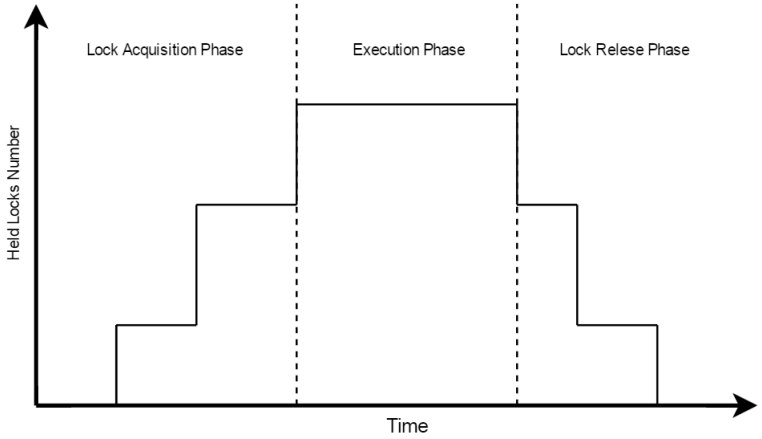


Figure 2.2: Two Phase Locking : *This graph shows a single processor gradually acquiring locks, executing after obtain all lock and then releasing locks gradually.*

Besides locking, the timestamp is another method used to enforce concurrency control, especially in some database applications [51]. A timestamp is basically a value conferred on every thread whenever there is concurrent access to shared data, following these rules:

1. Each participated thread should have a unique timestamp;
2. The orderings of multiple concurrent actions must be able to be identified by timestamps;

The timestamps should also deal with out-of-order accesses. For example, there is a database of multiple shared data, and a timestamp would be generated whenever read or write by a thread happens to indicate the order of those reads and writes to keep a sequential order. But as thread communication is asynchronous in nature, some accesses can be received out of timestamps order, and this kind of accesses should be detected and aborted.

Obviously this kind of aborting can reduce the performance of parallel execution, especially when there are hug numbers of requests should be aborted. However, buffer is proposed to improve the reduced performance by this. Not like always requiring the server to deal with requests whenever they arrive, all requests can join a buffer for a specified duration instead. And after some extended

behaviour of the application, an order of buffered access requests can be selected to reduce the aborted accesses number. The difficulty in applying this technique into application is the duration of requests buffered time and the amount of stored requests.

There are three frequently implemented methods of timestamp [1]:

1. *System clocks*. Timestamps can be generated from the system clock on the host platform;

2. *logical clocks*. Timestamps can also be generated from simply increasing integer values, which is called a logical clock;

3. Some others methods use a combination of system clock and logical clock. For example, when applying timestamps to distributed systems, as each site has a unique ID, a logical lock can be appended to the ID to form a timestamp which is unique to a specific thread on a specific site.

Timestamps are better for hosts over geographically distributed systems than locking because of the high latency of inter-host communication. However, the following issues should be processed appropriately when implementing concurrency control with timstamps:

Resolution. When using clocks as timestamps, the granularity of timing must be sufficiently accurate to make sure that timestamps are unique values even when generated very close in time.

Locking. Irrespective of whether system clocks or logical clocks are used, some concurrency control mechanism is also required to ensure the uniqueness of timestamps for each thread, for example, and atomic increase to a counter.

Bounds. As memory is not infinite in any site, a maximum number of values that can be represented needs to be clear. And further care should be taken when the value of a timestamp is going to exceed the capacity of memory, in order to ensure that no error will occur.

2.1.4 Transactions

The term transactions used for concurrency control on general purpose multiprocessor platforms has involves similar principles to those applied in database

applications. In a database system, many concurrent clients apply changes to the database in the form of transactions which contains all of the modifications a client wishes to commit. And modifications in a transaction can be either all successfully committed or all failed [24]. A database manager with some programmer defined rules will resolve conflicts between requested transactions so that the client can be liberated from the complexity of concurrent programming. Figure 2.3 shows an example of a time-line with three threads trying to commit transactions. As there is a contention (updates to the same memory location) between the transactions committed by thread 1 and thread 2, thread 2 must abort its transaction and retry it. However, although thread 1 and thread 3 try to commit transactions at the same time, as there is no interference between them, they can both commit their transactions.

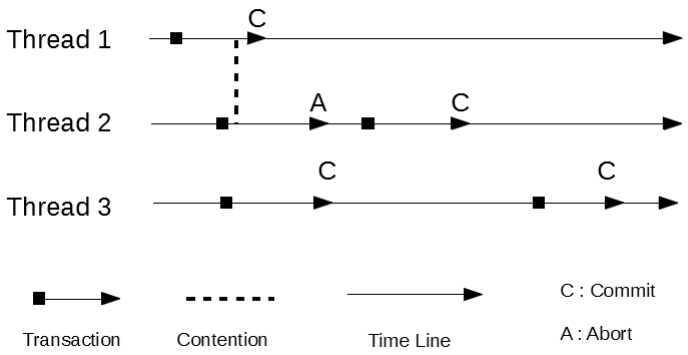


Figure 2.3: An Example of Transaction Contention.

The principles of how database systems manage concurrency are based on the 'ACID' regulations [22]:

1. *Atomicity*: all actions made by a transaction must take place entirely or not at all;

2. *Consistency*: the behaviours of transactions must be validated to meet all rules set by the database;

3. *Isolation*: all behaviours of a transaction must be isolated from other threads until it has been committed successfully;

4. *Durability*: all transactions should have their effects remain after commit, even in an event such as power loss.

If all transaction managers observe these ACID rules then a generalised transactional system can be provided, which is lacking in pessimistic approaches. Furthermore, atomicity and isolation are especially useful for multi-processor platform applications as they treat multiple operations on shared data within a single atomic section. In this way operations on shared data can be composed optionally to reduce the difficulties of concurrent programming with intricate interactions.

However, there are still two major problems that transactional memory needs to solve, namely:

1. High Contention. As threads must abort and retry all executions that encounter interference, a lot of the work accomplished by thread may be wasted, especially when the contention rate for shared data is high.

2. Starving. Another main drawback is that some transactions may always be in the position of being aborted and rolled-back [52]. For instance, if there is a long transaction and many short transactions continuously come in, if the long transaction always conflicts with the short transactions, then it might never be able to commit.

To solve these problems, a contention management policy(CMP) is required, which is discussed in Section 2.3.4.

In parallel programming practice, transactional memory fall into two main areas:

1. *Hardware Transactional Memory (HTM)*: this is a solution for modifying cache protocols and architectural design to provide some functionality to transactional memory semantics. The execution speed usually exceeds the software transactional memory.

2. *Software Transactional Memory (STM)*: this is a generalised programme implementation of transactions that apply ACI rules for threads accessing shared data. The flexibility and dispensability of modifying hardware are the main advantages compared to hardware transactional memory;

Hardware Transactional Memory

It is not only restricted to software applications when implementing transactional memory, but hardware transactional memory (HTM) is also an area where transactional memory can be implemented. The main idea is to modify the hardware architecture to provide some features for supporting transactions in HTM, specifically:

1. a mechanism to provide isolation when threads execute instructions;
2. a mechanism to detect conflicts in the consistency of data;
3. a mechanism to undo or roll-back committed results.

The techniques used by HTM have been extended from those used to deal with the consistency of data held within processor and memory caches such as cache coherence protocols, speculative execution techniques and memory consistency models [5]. Cache coherence protocols can detect inconsistent data held in multiple versions by different caches. In this way, threads are allowed to effectively execute many instructions in isolation from other threads as a significant aspect of the transactional memory. And speculative execution techniques enable processors to execute instructions in a different order from the program's order and then roll-back to a previous state if any inconsistencies occur. With its help, threads are able to try different of instructions orders and roll-back to a consistent state if necessary. Finally, memory consistency models allow processors to detect and forbid errors introduced by executing instructions out of the program's order. This can support a thread in executing atomic instructions and detecting conflicts.

Software Transactional Memory

There are a lot of techniques which aim to provide transactions implemented in software. With the ACI rules, concurrency control can be achieved in an application level. As it is more flexible and easier to experiment compared to the use of hardware transactional memory, STMs can be easily integrated with programming language and provide a light prototype solution. However, the performance and bottleneck of improvement tends to be worse than with HTM, and numerous studies have been carried on to reduce those defects.

Many STM techniques have been devised at the time of writing, and they can often be categorised in three types of methods: the granularity of shared data, overheads of updating shared data, and the synchronization mechanism in accessing shared data.

STMs can fall into two approaches if the granularity is differentiated:

1. Object based STMs – here data is represented as objects in this kind of solution. An object is a composed concurrent data structure. As it is similar to Object Orientated language in nature, it can easily be integrated into that kind of programming languages(e.g. DSTM2 [26]).

2. Word based STMs – the granularity of shared data is memory words in this kind of solution, so it involves a lower-level of concurrency than object based STMs(e.g. TinySTM [45]).

Also, when considering the overheads associated with accessing shared data, two modes are available:

1. Deferred update – the updating of shared data is not really accomplished during the transaction, but is enacted a cope-in thread local cache. After the whole transaction finishes, all updating then takes place from private caches to the shared data;

2. Direct update – the updating of shared data is directly made on the original version, which can reduce the overheads of creating local thread copies. The consistency of shared data can be achieved by restricting shared data to one thread at a time. And when a conflict over shared data acquisition occurs, a contention management scheme should be consulted to decide which one must abort.

Besides this, STM approaches can be divided into two kinds with regard to the technique of synchronization employed of accessing data:

1. Obstruction-free – In an obstruction-free STM model, if a thread executes in isolation at any point, it will make progress with its transaction(whether commit or abort). For instance, a numerical marker can be set for shared

data, and when threads read the shared data, they read the marker as well. Threads will compare the previous read marker and the current marker when they try to update and if they match, then committal can be accomplished and consistency is guaranteed; otherwise, they need to abort.

2. Lock-based – In a lock-based STM model, the short critical sections guarded by locks allow ownership of shared data. A deadlock needs to be settled cautiously by methods such as maximum attempt limitation or prioritising.

Every different approach to shared data access has its own benefits and overheads. Most STMs combine several approaches for different purposes. Additionally, a significant feature of all STMs is the contention management policy(CMP). This is consulted to decide which thread can progress when threads confront conflicts in trying to access shared data. When the number of threads and the conflict rate are low, the CMP seems to be less important. However, if the concurrent resource is numerous and the conflict rate is high, it becomes a non-negligible task to ensure that all threads can execute as expected.

2.1.5 Contention Manager

As mentioned above, the two main issues in Transactional Memory are high contention and starving. As more transactions execute concurrently, there are more likely to be contentions for access to shared data, which may lead to a greater frequency of aborting or rolling-back transactions and the overall performance of the programme could be reduced. To mitigate the time wasted caused by this, contention management is required for STMs to reduce the frequency of occurrence of aborted transactions. A variety of studies of contention management have been carried out and various of approaches have been implemented which can be divide into the following categories:

1. Exception-based. The way to deal with aborting transactions is to leave it to the programmers to decide. Exception handling only provides a mechanism in programming languages to support throw and catch semantics for exceptions;

2. Wait-based. This kind of contention management typically solves interference between threads by just allowing one thread to proceed while aborting other threads and forcing them to wait for a period of time before retrying.

3. Serializing. Rescheduling the execution of aborted transactions is the main method for resolving conflicts between threads. This is different from wait-based approaches, in the serializing solutions often need to allocate transactions to threads so that aborted transactions can be executed by one single thread(sequentially) in order to avoid further conflict.

With the exception-based approach, the programmer can benefit from customized solutions to contention which are more flexible and can be more suitable for a single application. However, the main burden of conflict-solving still falls on application programmers.

Plenty of approaches to solve contention exist in wait-based CMPs. For instance, Polite, Karma and Polka [21, 46] are well-known approaches which are relatively straightforward to implement, offering versatility and good performance. The CMP in Polite will abort one transaction if two transactions conflict, and will then increase the back-off time before retrying to avoid another conflict before the winning one finishes. Karma uses priorities for each transaction, and aborts the one with lower priority when conflict occurs. The main idea is to abort the transaction with the least cost associated with rolling it back. Polka is a combination of Polite and Karma, and uses an exponential back-off time period for aborted transactions. An inefficiency of wait-based approaches has been identified [25], however, as the dynamic nature of the execution of STM leads to difficulty in finding an adequate back-off period.

Serializing CMPs typically reschedule or serialize aborted transactions. An example has been described [3] whereby transactions are distributed among the threads of the application, with transactions that are likely to conflict being assigned to the same thread, thus assuring serialization. In Ansari's study a Stealon-Abort approach is described in which various techniques are considered for rescheduling transactions amongst threads, with additional work-queues created when the number of transactions passes a threshold [2]. A collision avoidance and resolution approach to schedule-based CMP was also been described [11]

which also reassigns conflicting transactions to the same work thread.

2.1.6 Semantic resolving Contention Manager

Several approaches to implementations of STM have been developed to solve semantic conflict by employing a universal construction (UC) [31]. This concept enables multiple threads to access a shared data structure in a wait-free manner. The UC technique was subsequently applied to transactions to handle particular failure conditions [8, 53]. The approach was further extended to remove the abort semantics of STM [9]. Thread level speculation is introduced to transactional memory [4]. In this approach, platform parallelism is exploited to explore different permutations of transactional elements. This is achieved by reordering the internal execution elements of a particular transaction to better reflect concurrent schedules of execution. This technique reorders execution at a lower level than that presented the present study which seeks to reschedule whole transactions to better accommodate the semantics of execution.

More recently, a CMP described has used a UC technique to resolve conflicts (labelled Hugh2) [47, 48]. The threads which are considered are those which contain transactions that have been aborted due to semantic conflicts within the context of a session. The technique was first presented for object-based STM [48] and then extended to resolve semantic conflicts in word-based software transactional memory [47]. Universal Construction allows any sequential data structure to be transformed into a linearisable representation that can be accessed and updated by a number of threads [31]. The UC consists of three phases. Firstly, a thread proposes an input to be added to the UC. Secondly, each thread which has proposed an input reaches a consensus to decide which input will be added. Finally, the winning thread updates a log of inputs to reflect the decision. Hugh2 accepts as input a permutation of one or more sequentially executed transactions and decides which permutation will be added to the log. This approach to CMP for STM was the first to use additional threads to provide multiple serialized executions of aborted transactions in parallel. This was achieved without the overhead of a thread-pool.

2.1.7 Summary

This section has explained the demand for concurrency control in parallel computing, and some basic methods. Most focus is currently on concurrency control as transactional memory as this is a general solution that can avoid the need for programmer of data correctness and preventing deadlock. However, different ideas of dealing with contention can lead to huge differences in performance, which makes a contention manager one of the most important parts of transactional memory. The next section discusses different parallel architectures and how these can provide concurrency control mechanisms for programmers.

2.2 Parallel Architectures

Traditionally, a computer is built in a single core architecture, and most computer software is written for serial computation. Moore predicted that chip performance would double every 18 months so that sequential programming can be easily enhanced by hardware [39]. However, from the beginning of the 21st century, multi-core processor platforms have appeared in CPU architecture and have become the main trend since integrating more cores is cheaper than improving frequency for a single core to provide the same speed-up for the CPU. Since then, the CPU has turned into a commonly used platform for parallel computing.

Developments in modern graphics processing units have attracted significant interest partly due to the fact that they can now accelerate applications not only in traditional computer graphics domains, but also in general purpose domains. General-Purpose GPU (GPGPU) applications provide a different architecture from Central Processing Units (CPUs). For example, a modern CPU architecture is typically optimized for low-latency access to cached data, and offers sophisticated control logic for out-of-order and speculative executions. For this purpose, the CPU has a larger cache and stronger control units than a GPU. The GPU, conversely, has more parallel processing elements than a CPU. As such, a GPU application can access and exploit a far greater number of threads.

2.2.1 Hardware

There are two different types of main memory in a parallel computer, which are known as shared memory and distributed memory. In a shared memory parallel computer, memory is shared between all processing elements in a single address space, and in a distributed memory parallel computer, each processing element has its own logical address space (normally in physical as well). If all elements of main memory can be accessed in the same bandwidth, it is called a uniform memory access (UMA) system, otherwise it is known as a non-uniform memory access (NUMA) system [37].

Depending on the different levels of hardware, parallel computers can be classified as multi-core computing, symmetrical multiprocessing, distributed computing, cluster computing, massively parallel computing, and grid computing. Multi-core computing has multiple processing units (know as cores) integrated in the same chip, whereas symmetrical multiprocessing means multiple identical processors that share memory and connect via a bus. Meanwhile, distributed computing is obviously a distributed memory computer system in which all processors are connected via a network, and cluster computing is a close group of loosely coupled computers. Massively parallel computing indicates a single computer but with many network processors, and finally grid computing means computers communicating via the internet to solve a given problem. As this thesis focuses on single chip computers, the next sections introduce the single multi-core processor for the CPU and the single graphic card for the GPU, both of which are UMA systems.

Although both architectures are suitable for recent parallel computing, there are still significant differences in their nature. As Figure 2.4 shows, the CPU usually has several Arithmetic Logical Units which can be treated as processors since they provide most of the computational power. However, as the CPU is designed to be more suitable for an operating system, it has to employ most of its transistors on features for that purpose. For example, a large on-chip cache is applied in the CPU together with further control units (such as instruction reorder buffers and reservation stations). In these ways it can optimize the execution speed of a single thread and is more suitable for most operating systems. As a

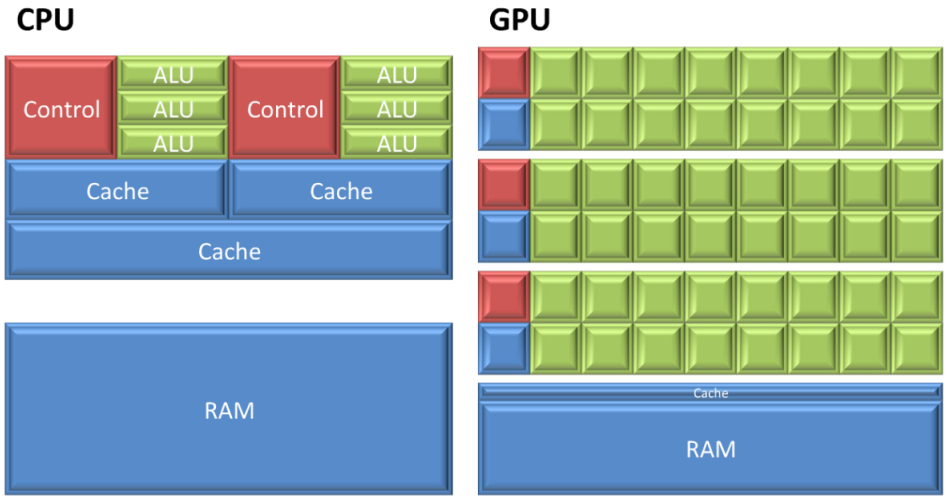


Figure 2.4: Differences between CPU and GPU Architectures.

result, the CPU can perform better for complex operations on a single or a few streams of data, but it is not that good for handling many streams simultaneously.

On the other hand, as the GPU is designed to execute parallel streams of instructions as quickly as possible, as Figure 2.4 shows, a GPU employs more transistors in processor arrays, multi-threading hardware, and multiple memory controllers which do not focus on a particular thread but support tens of thousands of threads to run concurrently with a high memory bandwidth. In such a way, the GPU excels over the CPU in areas which require more parallelism such as video processing and physics simulations which execute simple operations on large amounts of data [42]. However, GPU memory is very limited compared to the CPU as the latter can use virtual memory while the GPU cannot, and GPU execution units do not support hardware interrupts in the same way CPUs do.

As a result of different design patterns, the CPU can only support one or two threads per core, while a CUDA capable GPU can support up to 1024 threads per streaming multiprocessor. CPU takes hundreds of cycles to switch threads but the GPU has little cost in switching threads. To reduce the latency of memory access, the CPU can use its large caches and branch prediction hardware while the GPU switches to another thread when one thread needs to wait for a memory reading.

2.2.2 Execution

To design a proper algorithm for GPU computing, it is necessary to take into account the differences in execution between the CPU and the GPU. The execution for a multi-core CPU is quite similar to that of a single thread, except that more threads can work together on the same task. Tasks are allocated to the CPU all the time and the results are calculated continuously.

In this thesis, the NVIDIA introduced CUDA (Compute Undified Device Architecture) is used as the GPGPU computing API for the graphics card hardware, and this section explains how CUDA executes.

Figure 2.5 shows an abstract architecture of a CDUA device. CUDA applications can run on any card which supports this architecture, but as each GPU device has different specifications, they may have slightly different sets of supported features and different numbers of available computational resources. However, any CUDA supported device should have some Streaming Multiprocessors (SMs), and each of which consists of some processor cores (ALUs), one multi-thread instruction unit, a shared memory and a set of registers [43].

Whenever a CUDA application is launched, and if it needs to use GPU resources for computation, a 'kernel' has to be invoked to match the work-flow from the CPU to the GPU, and a specified number of primitive threads will simultaneously execute that work-flow. Batches of these primitive threads are organized into 'thread blocks'. The amount of primitive threads within a thread block is specified by the programmer, but is also limited by the amount of available shared memory as well as the desired memory access latency hiding characteristics. At the same time, a total of 512 threads for one 'thread block' is the upper limit set by the architecture. One or more 'thread blocks' are assigned to an SM during the execution of a kernel, and CUDA run time handles their dynamic scheduling to a group of SMs. The scheduler will only assign a thread block to a SM when it has enough resources to support that thread block. Then after a thread block is assigned to a SM, it will be further divided into 'warps' which each contain 32 threads and share the same instruction counter. If flow-control statements (e.g if-then-else) are introduced to the program, a possibility of thread divergence is also introduced. Thread divergence occurs when some threads in the same warp

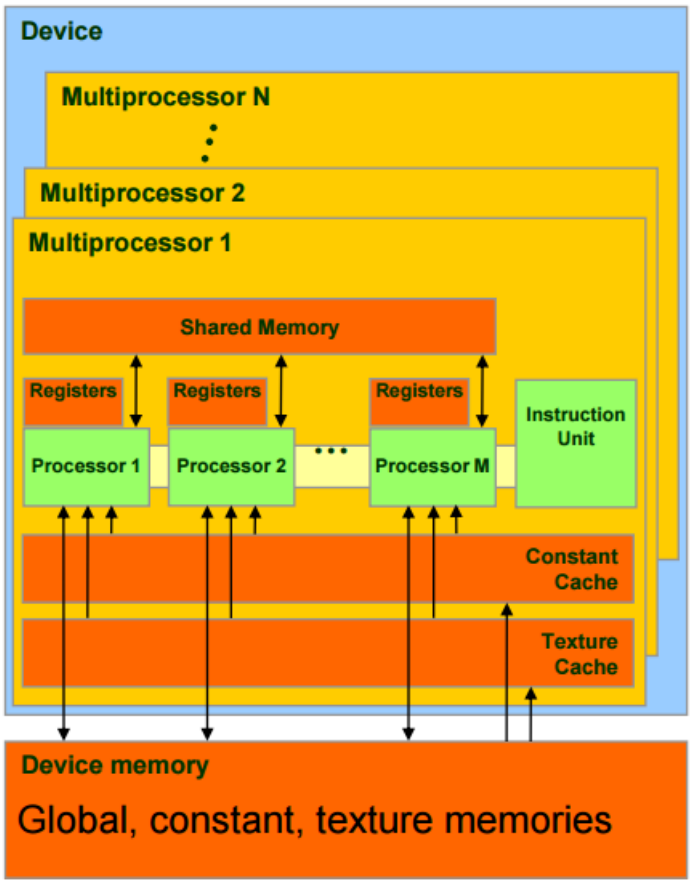


Figure 2.5: CUDA Device Architecture.

follow a different execution path from others. In this circumstance, threads following one path execute first, while other threads following different paths are suspended, then after this path is completed, other threads will execute while the current executing threads are suspended. This is called a 'lock-step' process and is one of the aspects most different from CPU parallel computing. After all threads in a kernel have finish their assigned tasks, the results data would be stored in GPU global memory and the programmer can copy them from there to the CPU memory for further use.

There are several levels of memory in a GPU device as Figure 2.6 shows, and each of them has its own read and write characteristics. Every primitive thread has access to private 'local' memory and registers. This 'local' memory merely means that the memory is private to the thread, but does not mean that it is

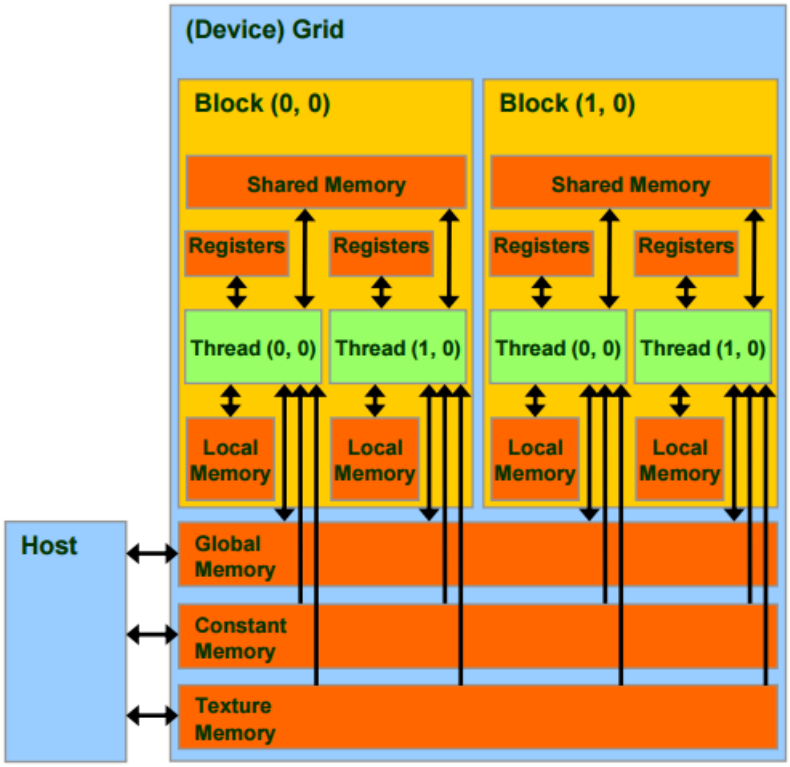


Figure 2.6: CUDA Memory Architecture.

stored locally on the thread registers. Instead, it is actually an off-chip memory in the global GDDR memory available on the graphics card. A unified 'shared memory' is located in an SM, which is shared among all threads of the same thread block. Finally, all threads have read and write access to 'global memory', which is located off-chip on the main GDDR memory, and is not cached. Therefore, it has the largest capacity but also costs the most to access. There also exists read-only 'constant' and 'texture' memory, which are in the same location as global memory but can be accessed more quickly as sometimes they may be cached. Those three types of memories located in the main GDDR memory and can be accessed by the CPU. While global memory can be read from and written to by both the CPU and GPU, read-only memories can only be initialized by the CPU.

2.2.3 Concurrency Control Operations

To produce a general purpose parallel programme, concurrency control has to carefully be implemented. And both the CPU and GPU provides basic operations to achieve this.

CPU

As the CPU has more control units on its chip, and it has been upgraded many times for control function, it has much more flexible basic concurrency control operations than the GPU. Normally, atomic operations, and thread synchronization and communication are all available for the CPU.

Atomic operations are operations guaranteed to be isolated from concurrent processes, and for all processes processing the same atomic operation at the same time, only one can succeed while others should receive a fail use notice. Atomic operations usually affect performance, but are the basic element in implementing a lock [33]. Normally atomic operations consist of atomic read-write, atomic test-and-set, atomic fetch-and-add and atomic compare-and-swap.

Thread synchronization is also named process synchronization, and it means that two or more threads will be guaranteed to not execute simultaneously in some parts of the program (called a critical section) [17]. It can be implemented by using spin-locks, barriers or semaphores.

Thread communication can be divided into broadcasting and all-to-all communication. Broadcasting means that one thread can send transmit messages to all receiver threads while all-to-all communication means that threads can send transmit messages to each other. Both types are important when modifications have to be made to shared data which has been copy held by more than one thread.

With all these concurrency control primitives, the CPU can use various concurrency control methods. Transactional memory is the most advanced kind as it can peel off the most difficult tasks such as deadlock and race conditions from parallel programming, with only cost being some overhead of performance.

GPU

Although GPU applications offer more parallelism than a typical CPU application, restrictions on data access mean that the semantics of GPU applications

tend to differ significantly from those of their CPU counterparts. Applications on the GPU tend to feature large numbers of threads which are wholly independent of one another with regard to their accessing of shared data (such applications are termed embarrassingly parallel). The coordination of thread activity is minimal in such applications, whereas this is rarely the case in the domain of parallel programming on the CPU.

To make GPU applications more generalized, concurrency control is needed. It is also achievable on the GPU because it has basic operations which are required by concurrency control such as atomic functions and thread synchronization.

However, approaches to concurrency control on the GPU are different from those with the CPU because the former has different features, as discussed in previous subsections. For example, as mentioned before, group of GPU threads execute as a 'warp' and all threads in the same warp share a same instruction counter [42]. This makes the GPU threads in the same 'warp' can only execute the same instruction at the same time, so that strategies to operate different threads with different instructions as is suitable for CPU applications is not achievable on the GPU. For example, a frequently used method of dealing with concurrent conflict is to make conflict threads back off and to ask them to wait for a different periods of time before retrying. In such a way, conflicting threads can restart at different time points to avoid the conflict happening again. However, all GPU threads in the same 'warp' share the same instruction counter and always execute the same instructions [55]. If this method was applied to the GPU, threads encountering conflict would always retry the aborted instruction at the same time and conflict again. Therefore, all technologies based on this principle which work well on the CPU cannot be applied for the GPU.

Another common technology used on the CPU is broadcasting. A lot of concurrency control methods on the CPU need to broadcast modifications to shared data from one thread to all other threads attempting to read or write the same shared data. But on the GPU, threads cannot communicate with each other directly, because inter-threads communication occurs only via global memory. However, seting a monitor to a global memory address is expensive since many threads have to access that address frequently, and even in this way the amount content which can be communicated is very limited. Hence none of the technolo-

gies based on broadcasting are suitable for GPU concurrency control.

As a result, some new principles for concurrency control on the GPU need to be explored. Current researches into concurrency control, and especially on transactional memory on the GPU, is being carried out by different groups around the world.

2.2.4 Summary

This section has compared the different features of CPU and GPU in both hardware and software layers. We can see that the CPU is better for conducting parallel computation that involves complicated thread execution, but provides fewer threads. Meanwhile, the GPU is excellent with simple multiple work on large amount of data. And although their execution modes are different, both require concurrency control and this can be approached in different ways. At the moment, concurrency control on the CPU is very advanced and has resulted in many transactional memory approaches developed to remove concerns about deadlock from the programmer. The GPU can also get considerable benefits from transactional memory as it is easy to use, but performance is slightly worse. The next section introduces some famous types of transactional memory on the CPU and some which have been recently developed for the GPU.

2.3 Related Work

As mentioned before, in blocking synchronization algorithms, mutual exclusive access to critical section is used which is usually guaranteed by locks. When a process tries to acquire a lock that has already been obtained by another process, it needs to stay in a waiting state until that lock is released. And this waiting state can cause various problems such as deadlock. However, the use of a non-blocking synchronization algorithm manes that no waiting state is needed and so many problems can be avoided. Some new problems such as live-lock can be introduced by non-blocking synchronization, but they can be effectively solved by contention management. As non-blocking synchronization has many benefits compared to blocking synchronization, and since transactional memory is a typi-

cal non-blocking synchronization construct that has been researched for decades, the present study also uses software transactional memory to resolve concurrency control. The following sections we will introduce some previous research conducted in this area.

2.3.1 Transactions on CPU

The First Software Transactional Memory

The idea of transactional memory was firstly proposed by Herlihy and Moss in [32], who described an architecture that extended multiprocessor cache coherence protocols. In this protocol, instructions would be stored in an instruction set provided by the system for access to shared memory positions. After this proposal, a more delicate system called Software Transactional Memory was implemented by Shavit and Toutitou [49]. Here, a system-wide declaration would be made if a transaction tried to update a shared memory location. This declaration is actually the owner of a particular concurrent object and referenced to that specific shared memory location. As a result, all other transactions can know that an update is being made to this location. After the update has succeeded, the ownership of this concurrent object should then be released. Both acquiring and releasing ownership of a concurrent object should be done atomically using the atomic compare and swap operation. To guarantee the correctness of the result, ownership must be exclusive, which means that at any given time each concurrent object can only have at most one owner.

In Shavit and Touiton's design, a concurrent object is a shared memory word, and each global declaration is another word called an ownership record. Concurrent objects and ownership records are associated one of each. The value of the ownership record can either be a null, which means no owner at all, or a reference to its owner transaction. Obviously, one ownership record can only have one owner transaction at any time. A transaction would try to own all ownership records it needs one by one, and it will release all of its acquired ownership records if it fails to acquire any ownership (when a concurrent object is already owned by another transaction). After it has acquired all of the ownership records it desires, it would commit the updates and then release all acquired ownership records. As

this is a non-blocking algorithm, deadlocks are avoided. However, livelock is still possible. To avoid livelock, ownerships are acquired in a global total order of concurrent object addresses. Additionally, a non-recursive helping mechanism is applied when conflict occurs. This means that when a conflict occurs between two transactions, one may take the other one's updates on its own behalf. And the help level is restricted to one in this solution, so it is non-recursive and livelock free.

There are three major drawbacks of this solution:

1. As it needs a global order to avoid livelock, prior knowledge of all of the concurrent objects one transaction accesses is necessary. It is not possible to achieve this in a real-time scenario as the concurrent objects which need to be accessed could be changed;

2. Memory usage is also a problem, as each shared memory address requires one associated ownership record. Even if the ownership record occupies one word, this means that the memory requirement is doubled.

3. The contention overhead during helping is also a drawback of this solution in terms of efficiency.

In order to overcome these limitations, some other techniques have been added in later studies.

Hash Table Based STM

Another word-based STM was proposed by Harris and Fraser in 2003 [23] which uses a hash table to store ownership records. In this new proposal, all of the complicated details are abolished and only some simple interfaces are provided for users, namely STMStart, STMRead, STMWrite, STMAbort, STMCommit, STMValidate, and STMWait. According to their literal meanings, they are invoked during different phases of a transaction.

Three main data structures are used to support this system:

1. *Application Heap.* This is the real shared memory in which all shared data is held.

2. *Hash Table of Ownership Records (orecs).* This is a hash table that maps shared memory locations. Information in it is similar to the previous structure but a version is used when there is no owner. This solved the memory spaces problem of the previous type of STM.

3. *Transaction Descriptors.* Each transaction has a transaction descriptor in which is held actual modification informations at shared memory locations along with the status of the transaction. Transaction descriptor consists of the address of the shared memory to be modified, the original value and version of that address, and the potential value and version after a commit action.

In a STMRead and STMWrite processes, an element will be added to the transaction descriptor if it was not already there. To guarantee consistency, all entries in one transaction descriptor that correspond to the same orecs must have the same old and new versions. Besides this, ownership is not acquired in this phase. Acquirsition only happens in the STMCommit phase so that STMCommit is a multi-word CAS. Whether or not the multi-word CAS is sued is an important difference between STMs.

STMCommit is the most important part of this system. In this phase, all orecs referred to its transaction descriptor need to be acquired. Atomic CAS operations would be used on every orec which has to be acquired. And if one transaction can gain all exclusive ownership, its status will be set to COMMITTED and it will be allowed to commit all updates. After that, it will release all acquired orecs with new version.

However, sometimes a transaction may find that another transaction's descriptor is already in an orec it needs to acquire. These are two kinds of conflicts here:

1. *Read Conflict* If the conflict happens during an STMRead session, then it is a read conflict. Under this circumstance, the current transaction will either be aborted immediately (if the conflict transaction is ACTIVE) or a new transaction entry is created in which the read version number and value is from the conflicting transaction (if the conflicting transaction is ABORTED or COMMITTED).

2. *Acquire Conflict* If the conflict happens during the acquirsition of an orec session, it is a acquire conflict. On this occasion, the current transaction also has to abort immediately if the conflict transaction is ACTIVE. But if the conflict transaction is ABORTED or COMMITTED, the current transaction will have to verify the consistency of the version number of the conflicted orec. If they are not consistent, then the current transaction has to abort and release all orecs it has already acquired. Otherwise, it has to check whether or not the valid updates from the conflicting transaction's descriptor have been made to the orec and the corresponding memory locations. The *Helping* mechanism mentioned previously may do so, but this cause reduced efficiency issue, and so the authors introduced another mechanism which is called stealing.

In the stealing mechanism used in this system, a current transaction that encounters a conflict will merge the transaction entries from the conflicting transaction's descriptor with its own transaction descriptor. The owner reference is then set to its own transaction descriptor as well. During the acquire session, if the current transaction is ABORTED by another transaction then it will release all orecs it has already acquired. After that, the current transaction will change its status to COMMITTED and write all modifications back to the memory locations corresponding to its transaction descriptor, and then release all owned orecs. However, it is possible for another transaction to abort the current transaction during the release phase, so-called 'dirty' updates may sometimes occur. For example, if the stealer transaction makes updates before the victim transaction does, then the victim transaction may overwrite the memory location with a incorrect value. To avoid this 'dirty' update, a redo operation is provided. When the current transaction detects an orec stealer, it will redo updates to the stolen orec from the stealer transaction's descriptor if the stealer is not ACTIVE. The combination of merging and redoing will guarantee the correct updates can be made to shared memory.

Contention management policy has a significant effect on performance. In the first hash table STM, the contention management policy is 'aggressive', where any conflicted transaction is aborted, which is not very efficient. But a alternative 'polite' was also introduced later in which, if one transaction faces conflict, it

will backoff in exponential increased time until get the limitation of backoff times then abort the conflicted transaction. This 'polite' policy shows a much better result than 'aggressive'.

One disadvantage of this system is that it is a word based system, which is not very practical in solving real problems. Moreover, not all platforms support the use of multi-word CAS operations.

DSTM

Recently, more research into STM has focused on object-level synchronization. This is because a lot of applications need to use data structures dynamically, which is more suitable for an object-based system. Compared to lock-free STM, obstruction-free STM has the advantages of simplicity, flexibility and efficiency. This section discusses an obstruction-free STM system which is called Dynamic Software Transactional Memory (DSTM) as proposed by Herlihy [27].

DSTM introduces a dynamic transactional memory object (TM object). A TM object is likely to be a wrapper surrounding concurrent data. It has a pointer to a locator object which is similar to the orec in a hash table STM. The locator points to a transaction descriptor that has most recently tried to modify the TM object, and both old and new versions of the data object. The transaction descriptor can be in any one of three phases: ACTIVE, ABORTED or COMMITTED. If the transaction is ACTIVE or ABORTED, the recent valid version is the old version referenced by the locator; otherwise, the new version is the new version referenced by the locator.

The locator is quite similar to the orec in Hash table STM, but there are three main differences:

1. The locator is referenced by one TM object, but the orec is referenced to a hashed memory word.

2. The locator points to both old and new versions of the data object, but the orec only points to the descriptor which contains old and new versions, or points to nothing but contains a version number.

3. The locator has a pointer to the most recent valid version of the data object, and so it does not need a version number.

To access a data object, a transaction must open the corresponding TM object, which means that this transaction is expecting to use this object. The transaction will create a local copy of the locator object first, and then set the transaction pointer inside it into the transaction itself. The pointer in the local locator depends on the state of the previous transaction in the locator:

1. If the old locator points to a COMMITTED transaction, then the new locator's new object pointer will point to a copy of this new object and the old object pointer will point to the new object version referenced by the old locator.

2. If the old locator points to an ABORT transaction, then the new locator's new object pointer will point to a copy of this new object and the old object pointer will point to the old object version referenced by the old locator.

3. If the old locator points to an ACTIVE transaction, this means that there is a conflict between this current transaction and the previous one. Then the current transaction will either abort the conflicted transaction or this will be determined by a contention manager. After that, it can follow the above two scheme to update the local locator.

After all these steps, the transaction will try to replace the old locator with this new locator using an atomicCAS operation. If this succeeds, then the update is valid and all other transactions can view it. If not, it means that some other transactions have acquired the object, which leads the current transaction to try again to acquire that TM object.

Invisible-read is an important mechanism used here to reduce unnecessary contention. As some transactions only need to read some concurrent object, if open operations were applied to all those read-only objects then a lot of access attempts would be rejected. This situation is not necessary. To solve this problem, the authors provide a separate read list of objects and these are only visible to the transaction itself. However, this may lead to inconsistency in views of memory. An incremental validation is applied to avoid data inconsistency where a transaction must check the data consistency for the whole read list whenever it opens a TM object or tries to commit.

Many more transaction memory systems on the CPU have been proposed during the last decade. It can be observed that STM is well developed on the CPU. However, as the GPU has only become popular in the last few years, there are only a few STM techniques suitable for it. The next section introduces some recent GPU STMs and their advantages and disadvantages.

2.3.2 Transactions on GPU

The first proposed STM for the GPU

As mentioned previously, the availability of the GPU has recently expanded into the area of general purpose programming, giving rise to a new genre of applications known as GPGPU. Although GPU applications offer more parallelism than a typical CPU application, restrictions on data access mean that the semantics of GPU applications tend to differ significantly from those of their CPU counterparts. Besides this, most existing CPU concurrency control methods do not work on the GPU, while the requirements of concurrency control for GPU computing is urgent and research providing concurrency control on the GPU has become vital. This has recently led to increased interest in utilising the GPU for transactional memory (TM). As with TM on the CPU, TM approaches on the GPU can also be categorized as hardware transactional memory or software transactional memory.

A first STM on the GPU that operates at the granularity of a thread-block (as opposed to the granularity of individual threads) has been proposed [7]. In this article, the author tried to transplant STM framework from CPU to GPU because of the advantages STM as we discussed before. The author discussed a blocking STM as well as a non-blocking STM in the article, both of which have four sessions. A begin function that can mark the initiate point of a transaction, a read function that can obtain a snapshot of a memory location, a write function that can log the updates should be commit if the transaction is successful, and finally a commit function that can perform the update log if there is no conflict or restart the transaction otherwise.

The author argues the progress guarantees to provide is the most important choice when design STM. For instance, When the contention rate is low, more

basic guarantees can achieve better performance as the tradeoff of conflict detection is tiny. However, more advanced guarantees can provide more independence from the scheduler but cost more in complexity. With this concern, the author designed a blocking STM uses a very simple structure which requires lowest possible resource to improve performance, and a non-blocking STM based on the Harris and Fraser's design [23] as it is more complex but offers better progress guarantees.

This system uses a Log or Undo-Log to deal with the contention condition. It is a kind of invisible read as it copies the object from shared memory location to local log whenever it performs a read, and commits at the end of one transaction. The conflict detection time is either at the first time one object accessed or at the time when transaction finishes. The author did not clearly mention which one they used.

The thread-block approach avoids dependency between threads within a single block and can solve the problem caused by lock-step execution mode provided by the GPU. Furthermore, this relatively coarse granularity reduces contention due to the typically high thread numbers available on GPU, but it does not accommodate workloads more appropriate for GPU execution.

GPU-STM

An approach labelled GPU-STM, which does operate at the granularity of the individual thread was also been developed [54, 55]. In this proposal, they took GPU main characters such as massive threading, SIMT execution mode and memory access coalescing into account to achieve a more suitable solution for the GPU. And unlike the first STM, they use single thread as the granularity which can make use of the more parallel resource from GPU.

The technique is based on a hierarchical validation system which is a combination of time-stamp and value-based validation. Value-based validation means record the actual value from locations read by a transaction to detect conflict. However, to avoid inconsistent view of memory, system with value-based validation has to perform incremental validation [20]. Systems on the CPU can shrink this non-trivial overhead by a single global sequence lock. But because thousands of threads have to update this single lock frequently and it may lead to a serialized memory update at commit time, it cannot be scale well on the GPU. On the

other hand, the timestamp-based validation uses global version locks to manage entire memory, each version lock masters a piece of memory. A transaction in marked as invalidated when the snap shot of version locks it requires not match. So timestamp-based validation can reduce off-chip traffic caused by comparisons. However, there arises a new problem marked as false conflict, which happens when transactions access locations managed by the same version lock, and that can be avoided by value-based validation. This is not a huge issue on the CPU, but on GPU, the lock-step execution mode would exacerbate the side effect as the some thread lane would be masked off and re-execute later. In their solution, a timestamp-based validation is performed firstly, and if the version lock does not match, a value-based validation is further performed to avoid false conflict.

GPU-STM uses locks to avoid memory inconsistency. As simple using locks may lead to livelock, and the common solution exponential back off cannot be applied on the GPU as we discussed previously, they proposed encounter-time locking coupled with commit-time locking. Specifically, each thread owns a local lock-log, and whenever transactional reading or writing takes place, a lock ID must be inserted to a proper position in that lock-log. The order of locks in local lock-log is sorted by their lock IDs, so that a global order can be obtained when acquire locks. One thing need to be clarified is one lock is not necessary to insert if it already exists in local lock-log to avoid duplicate locks. To avoid the possibility of livelock, at the commit time, a transaction should sequentially process the locks within each bucket. In this way, a global order of acquiring locks is obtained among all transactions, so that livelock is avoid without exponential back off mechanism.

Furthermore, GPU-STM leverage memory access coalescing mechanism to reduce the overhead of global memory accessing. All transactions within one warp merges their read/write set in a way that all transactions can access consecutive locations. For each transaction within a warp, it can use its index in that warp to access the independent region of merged read/write set.

This GPU-STM is the most efficient one at the time of study, and most important parts are discussed in detail. It is very sapiential to combine encounter-time lock sorting with commit-time locking acquiring to avoid live lock, but the time complexity of lock sorting can still lead a nontrivial overhead with large scale

read/write set. The technique described in the present study avoids the necessity for sorting locks due to the introduction of a static priority rule. Further on, GPU-STM is utilised as the comparator during the evaluation of the system.

Lightweight STMs on the GPU

Three lightweight techniques for software transaction management on the GPU have been described [34]. The author argued a backoff mechanism together with adding a delay before restarting aborted transaction can guarantee livelock free. As we discussed previously, this can only be a guarantee if aborted transactions are inter-warp transactions, otherwise, different delays may lead to the same restarting time because of the lock-step execution mode on the GPU. So we assume this proposal using a warp as a basic execution grain. And for each memory location, there is an one-to-one mapped shadow entry which is using to detect conflicts and 32 bits in size. They also defined a undo log for each thread which contains the set of memory locations it accessed and the old values as well. This undo log will be used if the transaction fails and restore original states of those locations.

Firstly, eager read-write conflict detect STM (ESTM) detects conflicts eagerly by observing both read and write information in shadow entries. Different threads are allowed to read a memory location while only one thread is permitted to perform a write. There are three kinds of conflicts as the author classified. 1) RAW violation which means a thread tries to speculatively read a memory location that another thread already speculatively written to. 2) WAW violation which means a thread tries to speculatively write a memory location that another thread already speculatively written to. 3) WAR violation which means a thread tries to speculatively write a memory location that another thread has speculatively read. A thread is able to add one entry to its read/undo log only if none of above conflicts are detected. The author claims the design of ESTM has two main weaknesses: 1) all read operations have to modify the shadow entry which is located in global memory; and 2) distinguish read and write operations need more memory space for the shadow entry and slows the performance. So they proposed the following two designs at the same time.

Pessimistic STM (PSTM) is a simpler version of ESTM that treating reads and writes in the same manner. This increases the effectiveness when transactions

regularly read and write to the same shared data. However, other applications do not perform such access patterns will suffers more from false conflict. The author argues this can benefit in two aspects. First advantage is it can detect earlier if some locations are first read and then written, and the second is simpler mechanisms can lighten conflict detection and commit overheads.

Finally, invisible STM (ISTM) can represent invisible reads to reduce conflicts during a transaction. All speculative reads are invisible to other threads and only validate at the end of one transaction. Therefore, the cost of read is much lower than previous two solutions because read operations do not need to modify the shadow entry. The shadow entries in ISTM is changed to version locks and can only be modified before commit. In such a manner, conflicts on writes are still detected eagerly while conflicts on reads are uncertain.

However, none of these three techniques allow a thread level parallelism and the memory overhead of shadow entry is huge as each memory location requires one. These factors make the solution not so suitable for GPU as it cannot take the advantage of massive threads in the GPU and not quite scalable. Besides, when the performance of each algorithm was compared with the CPU [34], only basic fine-grain and coarse-grain locking benchmarks were employed.

Other GPU concurrency control researches

Some work has also been carried out on to develop hardware transactional memory (HTM) for the GPU. A technique using value-based validation has been proposed [14] which is called KILO TM and uses combined value-based validation, RingSTM and scalable transactional coherence and consistency [15]. However, this technique requires significant changes to the architecture of the GPU itself to handle divergences in the control flow caused by the aborting of transactions.

A generic modification solution has also been proposed [41]. This uses morph algorithms, which are often used to deal with neighbourhood conflicts, modified so as to be GPU friendly. In this study, a suitable scheme to reduce the abort ratio by changing kernel configuration is also applied.

GPU techniques to speed up execution by reducing the usage of atomic operations have also been explored [40]. Two methods were used: barrier-based processing and exploiting algebraic properties were used to avoid atomic instructions from a program and the result showed some improvement in performance.

2.4 Summary and Thesis Contribution

This chapter has discussed some frequently used concurrency control methods. Pessimistic approaches were introduced first, basically involving locking. Then some optimistic approaches were mentioned, focusing on transactions. After that, some basic primitives for semantic resolution were introduced, followed by a brief description of the latest solutions. The differences between two most popular platforms for parallel computing, CPU and GPU, which lead to different demands in concurrency control technologies were then discussed. Finally, a number of recent studies of transactions on both the CPU and GPU were considered.

In summary, some key conclusions may be drawn:

1. The benefit of locking is that it is simple and intuitive for less complicated problems, and it is used in a lot of existing projects where programmers can use such methods with abundant support documentations.

2. However, locking can easily introduce errors into sophisticated applications, especially deadlock and livelock, and cannot be applied for general purposes because it is customized.

3. Optimistic techniques have become more practicable with the increase parallel computing resources. In particular, transactional memory is the most popular solution as it can provide an intuitive interface for programmers without the risk of introducing deadlock or livelock, and it can be generalised.

4. Unfortunately, transactional memory does not performs as well as expected when contention rates for shared data are high. So there is now much focus on contention management which can coordinate access of the shared data and enhance performance.

5. Three kinds of contention management policies are applied to coordinate transactions and reduce the possibility of concurrent conflict. Furthermore, the universal construction technique is used to handle particular failure conditions and is applied in transactional memories to deal with semantic conflict.

6. Transactional memory is advancing for use on the CPU, a number of different approaches have been proposed which have been applied to different scenarios.

7. There exist a few transactional solutions for either hardware or software on the GPU for concurrency control, and some contention management methods have been proposed to reduce the possibility of transaction conflicts. However, these contention managers have different disadvantages and none of them consider semantic conflict.

Given this background, the present study focuses on software transactional memory to solve both concurrent conflict and semantic conflict for GPU architecture. The main contributions of our prototype can be supported by the following objectives:

General Purpose – As we implement our system in software and without any limitation on a particular application or system, it is a general purpose implementation with interfaces to handle different scenarios.

Efficiency – We consider any issues which are baleful to performance on GPU architecture and the benefit that GPU platform can provide, to provide the most efficient prototype on the GPU.

Correctness – As all threads that wish to update modifications have to validate before commit, and commitment to shared data is isolation from other threads, consistency is assured.

Chapter 3

PR-STM : A Priority Based Software Transactional Memory for the GPU

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

3.1 Introduction

The availability of Graphics Processing Units (GPU) has recently expanded into the area of general purpose programming, giving rise to a new genre of applications known as General Purpose GPU (hereafter GPGPU). The principle benefit of using the GPU is the relatively high degree of parallel computation available compared to the CPU. Furthermore, programming APIs, such as CUDA, have

grown in sophistication with every new advancement in GPU design. As such, GPGPU programmers now have at their disposal tools to enable them to write complex and expressive applications which can leverage the power of modern GPUs.

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

To address the difficulties of multi-threading on the CPU, significant progress has been made in providing Concurrency Control techniques to aid the concurrent programmer as we discussed before. One notable technique is Transactional Memory (TM), which allows the execution of transactions in both Software and Hardware. TM provides an intuitive interface to aid programmers of multi-threaded programs. The TM system guarantees that programs are free of data inconsistency issues while handling the intricacies of thread coordination and contention management. Besides, a preeminent TM system can be efficient as well as forbid all possibilities of errors (such as deadlock and livelock).

At the time of writing, implementing an efficient TM technique for the GPU remains an area with much potential for development. Although semantic conflict solving is the ultimate objective of the whole research, the work in this first step aims to contribute to a efficient concurrent conflict solving capability, by providing the following:

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

been produced between STM techniques for the GPU and the CPU.

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

Section 3.2 describes the design and implementation of our STM and, finally, Section 3.3 concludes the chapter and discusses next step work.

3.2 System Design and Implementation

3.2.1 Overview

The operation of the GPU differs considerably from the CPU and this must be taken into account when implementing transactional algorithms on the GPU. First of all, the degree of available concurrent threads are far more than they appear on the CPU. And in addition to the high degree of threads, groups of GPU threads execute as part of a 'warp'. Threads belonging to the same warp share the same instruction counter and thus execute the same instruction in a 'lock-step' fashion. In addition to the risk of high contention given the high number of threads, deadlock and livelock are possible because threads of the same warp cannot coordinate their accesses to locks as they can on the CPU (see Figure 3.1(A)). For example, a frequently used technique to deal with the possibility of deadlock on the CPU is retreat and retry in different period of time. However, as threads in the same 'warp' share one same instruction counter, this method is no longer achievable as those threads can only at the same time. As a result, a new technique should be invented to solve this problem.

To prevent the possibility of deadlock and livelock, we use a 'lock stealing' algorithm which requires each thread be assigned a static priority, here is the unique thread ID in CUDA. This allows a thread with priority n to steal a lock which is currently owned by any thread with a priority less than n (see Figure 3.1(B)). As every thread has a unique priority, this addresses the possibility of deadlock because any thread can always determine its next action when encountering locked data. And for one locked data, all attempted threads would only try steal the lock once and find out which threads have the highest priority and the acquire

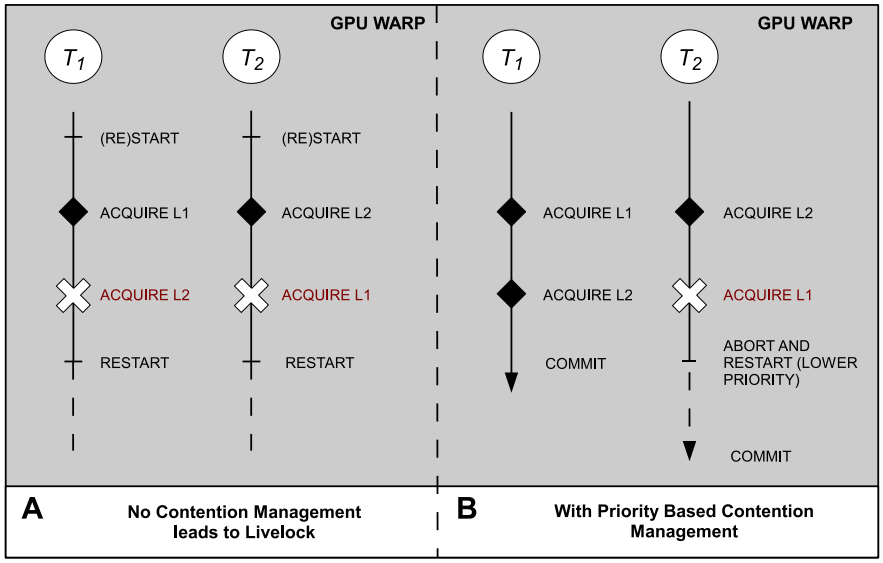


Figure 3.1: Livelock and Contention Management in GPU Transaction execution.

it. Livelock is also addressed as threads will never attempt to perpetually steal one another's locks.

PR-STM implements a commit time locking approach where threads attempt to acquire locks at the end of their transactions. Before committing, threads first attempt to validate their transactions by tentatively 'pre-locking' shared data and check current versions. Pre-locked data can be stolen based on the thread priority rule. Threads can only pass this validate phase if all locks can be 'pre-locked' without stole by others and all local hold read data versions are still consistent with them on global memory. If validation is successful the thread may commit its transaction. We implement invisible reads and threads maintain versions of the data they have accessed so that they can abort early if a conflict is detected. This has the benefit of reducing the costs of false conflict where a thread needlessly aborts when encountering data locked by a transaction which itself will abort in future.

3.2.2 Metadata

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

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

- *local read set* is a set of read entries each composed of a memory location, version and value read by the current thread;

- *local write set* is a set of write entries recording the memory location and value written by the current thread;

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

Most of our STM metadata is local to a thread because this has some benefits over global meta data. In particular, some local metadata may reside in the register which allows faster access by a single thread. And although some local metadata may be located in hardware global memory which has slower access than block shared memory, the block shared memory has strict restricted space that can not store all local metadata.

Algorithm 1: PR-STM functions

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

3.2.3 STM Operations

The *PR-STM* system can be divided into the main frame and contention manager, and we will discuss the main frame in this section while leaving the contention

manager to the next section. *PR-STM* is comprised of several functions that are executed during significant events during a transaction's execution. Specifically: *txStart*, *txRead*, *txWrite*, *txValidate* and *txCommit*. Algorithms 3 and 2 provide the pseudo code.

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

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

txWrite records each write a thread wishes to make in its *local write set*. The thread first checks if the data is already locked by another thread and if so it means another thread is modifying the value at that moment, so this thread sets its abort flag to true indicating the transaction must abort and restart when the function returns (line 19). If the data is not locked the thread checks if the data is already in its *local write set* (line 14) and overwrites it. If the data has not been previously written the thread creates a new write set entry, which contains the lock information (lock index and version) to be added into *local lock set* and the attempt modified value to be added into *local write set* (lines 15-18).

txValidate is invoked before the transaction can commit. The thread attempts to lock all shared data that it intends to modify and performs validation of all the shared data it has read. The thread invokes *prelock* on all data read/written (line 20) to determine whether it has the highest priority value. Then the thread

validates all the data in its read set by checking that their versions have not changed (lines 21-22). If validation is successful the thread will try to lock all data (line 23). If this is successful then the thread can now commit its transaction. If any of these steps fail, the transaction must abort.

txCommit is invoked only when a transaction has already successfully validated. The thread writes to all global shared data in its *local write set* (line 26) and executes a 'thread fence' (line 27). CUDA provides a thread fence function to ensure memory values modified before the fence can be seen by all other threads. Without a thread fence, the weak memory model of the GPU might cause a reordering of a thread's instructions, which could lead to inconsistent shared data. The thread fence ensures that modifications to shared data are visible to all threads before any locks are released. The thread then updates the version bit in the *global lock table* for each lock in its lock set. The version bit is either incremented (line 30) or reset (line 31) if the version value has reached the maximum value to avoid version overflow.

3.2.4 Contention Management Policy

In *PR-STM*, the most basic element is lock, and 32-bit memory words are used to represent locks. Each lock can cover from one single memory address (most efficient way but cost most memory space for locks) up to all memory addresses (similar to block solutions). We use locks for both protecting shared data and implementing our priority rule policy. The various bits of each lock represent the following:

- The first 11 bits of a lock represent the current version of that lock. The version is incremented whenever an update transaction is successfully committed to any data covered by this single lock.
- Bits 12-30 represent the priority of whichever thread has currently prelocked this lock (if such a thread exists). A lower value represents a higher priority.

Algorithm 2: PR-STM functions

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

- The 31st bit indicates whether this lock is pre-locked. Pre-locked locks may be stolen from threads with lower priorities and acquired by threads of higher priorities.
- The last bit represents whether the lock is currently locked. Once this bit is set, no other threads can acquire this lock.

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

tryPreLock is called whenever a thread attempts to pre-lock shared data. For each lock in its *local lock set*, the thread checks whether the lock versions are inconsistent (line 35) and whether the lock is locked by other threads (line 36). Finally, if a thread can pass the previous inspections the thread will check whether the lock has been pre-locked by another thread with a higher priority (line 37). If

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

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

releaseLocks is called when a thread commits or aborts. All pre-locked/locked locks are released. Pre-locked locks must be released by CAS (line 50) in case the lock has been stolen by another thread. However, locked locks can be simply released by change value as it is isolated form other threads after locked.

3.3 Summary

In this chapter we have presented *PR-STM*, a new scalable STM technique for the GPU which uses static thread ranking/priority to efficiently resolve contention for shared locks. We will demonstrate the performance of our approach against both GPU (*GPU-STM*) and CPU (*TinySTM*) software transactional memory libraries which, to our knowledge, is the first time such testing has been done in Chapter 5. Results for transactional throughput and scalability demonstrate that our approach performs better than both *GPU-STM* and *TinySTM* in almost all cases (could be view in 5.2).

The results in 5.2 suggest that the GPU is particularly effective at processing large numbers of short transactions, while the presence of read-only transactions provides only a small improvement to GPU performance. Further testing will allow us to formulate transaction allocation strategies, assigning work to either the CPU or the GPU based on the effectiveness of each processing element to execute

that work. We believe there exists much scope for expanding our approach. After this first step, we would like to enhance our Contention Management Policy to accommodate dynamic priorities(this was done then but proved no improvement in most cases) and application semantics (this has been shown to provide substantial performance improvements). The new upgrade version of *PR-STM* will be presented in the next chapter, which will have the capability to deal with semantic conflict as well as concurrent conflict. Besides, the way of generating transactions will be transferred from GPU to CPU, which will be copied to GPU as a global transaction table.

Chapter 4

Resolving Semantic Conflict in a Parallelised Contention Management Policy on the GPU

The increased parallel nature of the GPU affords an opportunity for the exploration of multiple solutions to contentious transaction ordering. A contention management policy (CMP) is presented for the GPU which resolves transactional contention by simultaneously assessing multiple ordering solutions and selecting the optimal transaction schedule. The CMP is extended to account for semantic transactions (i.e. those whose ordering is dependent on the application itself), as well as concurrent conflict. This approach removes the requirement for the application programmer to correctly order transactions which are dependent on conditions specific to the application. The results show that, while there is an increased overhead in dealing with the possibility of semantic conflict, our algorithm can be applied to all situations, whether there is semantic conflict present or not.

4.1 Introduction

A contention Management Policy (CMP) for parallel exploration of solutions to transactional conflict using the GPU is described. The approach utilises a priority

rule based technique to explore multiple schedules of transaction permutations on the much more highly threaded GPU, in the context of resolving conflict in software transactional memory (the technique is labelled PR-STM).

We then extend our CMP for resolving concurrent conflict on GPU to also address semantic conflict. A semantic conflict occurs when there are application-specific factors affecting the order in which transactions must be completed (for example, funds must be placed into a bank account before they can be withdrawn). A CMP which only addresses concurrent conflict will often stall when confronted with a semantic conflict. We introduce a semantic conflict management policy whereby a thread that is executing a transaction must check whether a semantic conflict occurs. If so, the transaction is delayed and the thread searches for the next uncommitted transaction. In order to achieve this on the parallel architecture of the GPU we also introduce a global transaction table which tracks the commit status of multiple transactions in parallel.

The effectiveness of PR-STM new version (hereafter PR-STM2) on GPU is demonstrated by injecting semantic conflict into three well-known benchmarks for contention management policies. We utilise the *Bank* benchmark (which is provided within *TinySTM*), the *Vacation* benchmark (commonly associated with Stanford Transactional Applications for Multi-Processing (STAMP) and the *Skiplist* benchmark. The performance figures presented show that our approach handles semantic conflict well as it is introduced into the benchmarks running on the GPU. The comparator CMP (GPU-STM) fails to progress when semantic conflict is introduced.

This is the first time that semantic conflict has been considered in a CMP that operates on the GPU. While there is an additional overhead in handling semantic conflict, our generalised approach to contention allows the CMP to be used for arbitrary permutations of transaction, without requiring any sorting by the user application.

4.2 Design and Implementation

4.2.1 Overview

Unlike theads on the CPU, groups of GPU threads execute in 'lock-step' fashion, sharing a single instruction counter. In GPU terminology, this group of threads is called a 'warp' and may comprise of 1 to 32 threads. In Chapter 3 we introduced a priority-rule based approach (PR-STM) for resolving concurrency conflicts on the GPU within software transactions. PR-STM avoids livelock, which is caused by a warp of GPU threads continuously generating the same concurrency conflicts due to lock-step memory access.

When semantic conflicts are introduced to the application, however, they both (a) increase the possibility that transactions must abort and (b) reintroduce the possibility of livelock. For example, a transaction may now abort either due to a concurrency conflict or the semantics of the application preventing the transaction from completing (i.e. attempting to remove from a shared buffer which is empty). Futhermore, if the shared buffer remains empty, then the transaction may never commit and is in a state of livelock.

It is the objective of PR-STM2 to reduce the rate of aborts caused by semantic conflict and reduce the possibility of livelock. Eliminating livelock completely, however, requires that some transaction always exists that will resolve the semantic conflict. For example, a transaction which appends items to an empty buffer or deposits funds in a bank account from which another transaction is attempting to dequeue/withdraw. Meeting this requirement is outside the scope of PR-STM.

Figure 4.1(A) shows transaction execution without a semantic CMP. Thread x attempts to withdraw funds from a bank account that is empty and aborts/retries until the account has been deposited with funds by thread y. The number of extra retries depends on the time taken for another transaction to deposit funds into the account which can be arbitrarily large causing much wasted execution. Furthermore, as thread numbers grow so does the potential for a greater amount of wasted activity.

In Figure 4.1(B) PR-STM2 CMP is introduced and all transactions are now stored in a *global transaction table*. As the scenario begins both threads en-

counter semantic conflicts. Unlike Figure 4.1(A), semantic conflicts now cause each thread to abandon their transactions rather than retrying. Instead, the threads acquire new transactions from the global transaction table. At some future time the abandoned transactions are re-executed, possibly resolving the original semantic conflicts if conditions have changed sufficiently (if withdrawals can now be made because deposits have now taken place, for example).

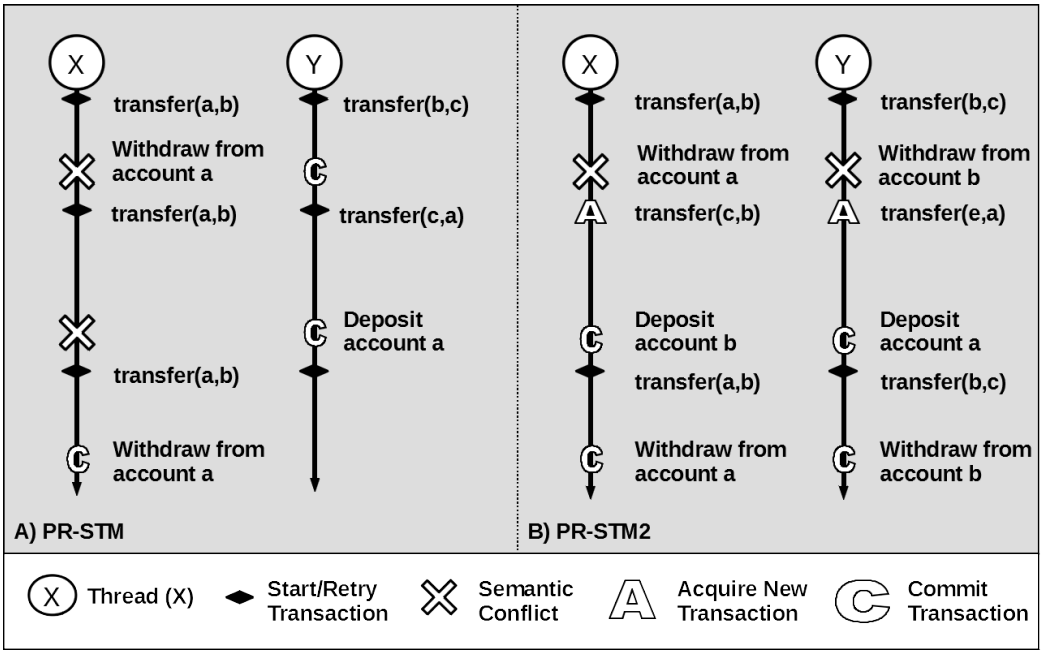


Figure 4.1: Unnecessary Retries are avoided by acquiring new transactions when a semantic conflict occurs.

PR-STM2 implements a commit time locking approach where threads only attempt to acquire locks at the end of their transactions, and detect semantic conflicts before modifications are made to the local write set. The user only need to label what is a semantic conflict in their scenario (e.g a withdrawn from an account with no insufficient funds) and whenever this condition meets, PR-STM2 will handle it in another manner than concurrent conflict. This approach results in a reduction of needless concurrent conflicts when a thread encounters a semantic conflict because all concurrent conflicts caused by this transaction can be averted.

4.2.2 Metadata

Two types of metadata are used in PR-STM2 which are same as PR-STM: global metadata which is shared among all threads and local metadata which is private to each thread. Most metadata are the same as those used for PR-STM but we introduce an additional metadata in the form of global transaction table.

- *Global transaction table*: contains all transactions that need to be committed. Each element consists of three parts: instruction (i.e. transfer, deposit, read-all etc), data (i.e. which accounts to access), and a finish flag indicating whether the transaction has been committed.

The introduction of the global transaction table allows us to generate transactions on the CPU, and pass them to the GPU for processing. This ensures that our STM system can be employed in a generalised fashion.

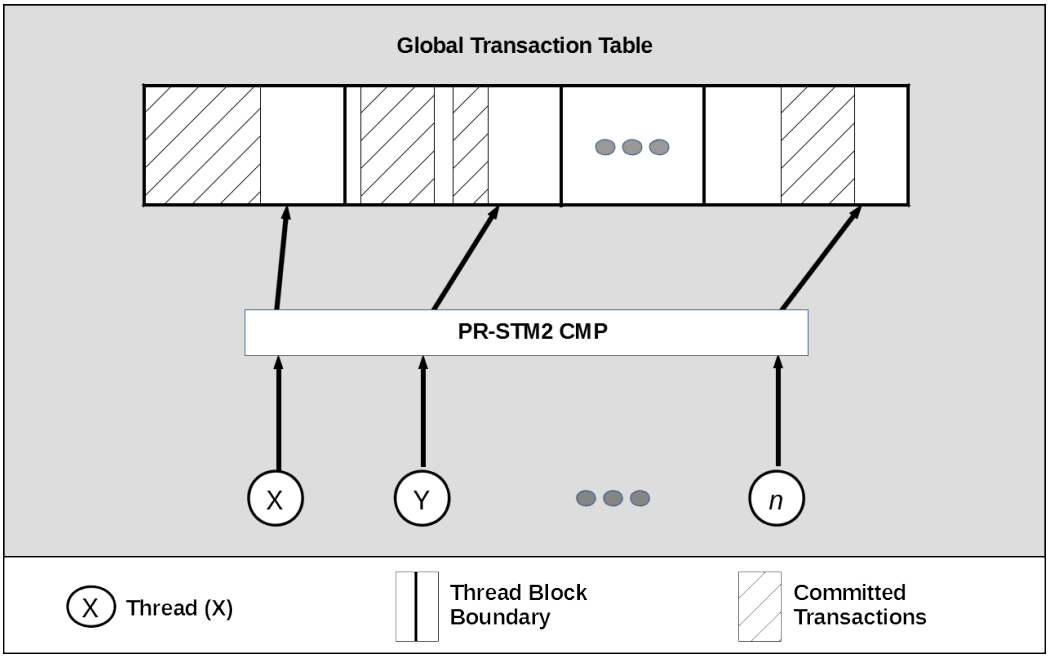


Figure 4.2: PR-STM2 maintains a global transaction table to provide GPU threads with transactions to execute.

4.2.3 Semantic Contention Management Policy

In PR-STM2, we use a postpone strategy to deal with semantic conflict and a global transaction table to store transactions (see Figure 4.2). When semantic conflict occurs, the thread aborts the transaction and reads the next uncommitted transaction from the global transaction table. As the search wraps back to the start of the allocated transaction block, threads will retry previously aborted transactions. Pseudocode for the management of semantic conflict in this manner is shown in Algorithm 3. It consists of two functions:

Algorithm 3: PR-STM2 functions

```
    function mainKernel()
53      startIns ← threadIdx * transEachThread;
54      finishIns ← (threadIdx + 1) * transEachThread;
55      while startIns < finishIns do
56          anyNotFinish ← false;
57          for i ← startIns to finishIns do
58              if g_insTable[i].finish = false then
59                  if doTransaction() = succeed then
60                      g_insTable[i].finish ← true;
61                  if doTransaction() = concurrentConflict then
62                      doTransaction();
63                  if doTransaction() = semanticConflict then
64                      anyNotFinish = true;
65              if anyNotFinish = false then
66                  startIns ← i + 1;
    function doTransaction()
67      if txRead() = false then
68          return concurrentConflict;
69      if semanticConflictDetect() = true then
70          return semanticConflict;
71      if txWrite() = false then
72          return concurrentConflict;
73      if txValidate() = true then
74          txCommit();
75          return succeed;
        else
76          return concunrrentConflict;
```

mainKernel is invoked when the GPU launches a thread to carry out a block of transactions. Firstly a thread gets a transaction block allocated (lines 53 and 54), then it tries to execute all transactions of that block sequentially. When a thread reads a transaction from the allocated block of the global transactions

table, it checks whether this transaction has already been committed. If this transaction has not been committed, it will call the *doTransaction* function to execute it (lines 59, 61, 63). If a transaction is successfully committed, this thread will set the finish flag to true in the global transaction table (line 60).

When a thread encounters a concurrent conflict and cannot complete because of lower priority, the thread will try to execute the transaction again (line 62). If a semantic conflict is encountered, however, the thread will skip the transaction and begin the next available transaction and setting a variable to indicate that a previous transaction has been skipped (line 64). The thread is finished when all transactions in the allocated transaction block have been committed.

doTransaction is called after a thread reads a transaction from the global transaction table. There can be different types of transaction (e.g deposit, transfer or withdraw) defined by the transaction type read from the global transaction table. If the executing transaction is not read-only, a thread has to check (line 69) whether it encounters a semantic conflict (e.g a withdraw transaction occurring before a required deposit transaction). If so, a $semanticConflict$ flag is returned. Threads also check for concurrent conflict using the described PR-STM policy (lines 67, 71, 73). If a concurrrent conflict is encountered the thread will abort with a $concurrentConflict$ flag (lines 68, 72, 76), otherwise, it will commit updates with a *succeed* flag (lines 74, 75).

4.2.4 PR-STM2 Handlers

A number of handlers are invoked during significant events in a transaction's execution. Specifically: *txStart, txRead, txWrite, txValidate* and *txCommit*. The handlers were developed for PR-STM, and the pseudocode was presented in previous chapter. In order to understand the functions of PR-STM2 easily, we will describe in what situation they will be invoked and the functionalities, but not the details about how they work.

- *txStart* is executed before a thread begins or restarts a transaction. The function initializes all the thread's local metadata.
- *txRead* is called whenever a thread attempts to read shared data from global memory. This handler checks if any global data is is being modified by

another thread so that it can abort and re-execute a transaction earlier if necessary.

- *txWrite* records each write a thread wishes to make in its local write set. Semantic conflict detection happens before this handler is invoked.

- *txValidate* is invoked before the transaction can commit. The thread attempts to lock all shared data that it intends to modify and performs validation of all the shared data it has read. A 32-bit word is used to represent locks in PR-STM, and we use the same priority rule as PR-STM to determine which thread can execute when concurrent conflict occurs.

- *txCommit* is invoked only when a transaction has already successfully validated and therefore can be guaranteed to safely update all modifications to global shared data.

4.3 Summary

A contention management policy (labelled PR-STM2) has been introduced which utilises the increased parallelism of the GPU to explore transactional ordering solutions to contention. The technique entails a priority rule based approach to contention resolution. The approach is the first for GPU to incorporate a technique for the resolution of semantic conflict without relying on the application programmer to pre-order transactions appropriately. This approach allows the technique to be applicable in any situation. Conventional CMPs will quickly reach a state where transactions can not be completed when confronted with semantic conflict, so such situations must be planned for and avoided by the application programme, requiring an additional overhead in both processing time and application coder understanding of the processes. Our approach automates the resolution of semantic conflicts, in coordination with transactional conflict.

The performance of PR-STM2 for GPU was compared against the existing solution for GPU known as GPU-STM will be presented in the next chapter.

In the future we expect that exploring our session locking mechanism within a distributed STM application will raise some interesting possibilities. The greater

scalability which session locking provides for distributed STMs is especially promising. Also of interest, will be combining the GPU and CPU within a heterogeneous transaction manager. The GPU is effective when processing large numbers of short transactions, while read-only transactions result in minimal performance improvements on GPU. Consequently it should be possible to formulate a transaction allocation strategy, which assigns thread exploration of transaction schedules to CPU or GPU as most appropriate.

Chapter 5

Evaluation

5.1 Overview

To evaluate our two systems, we compare the performance of our systems separately with different systems. For the very first version, which only deal with concurrent conflicts, we compared the performance with one popular CPU STM (*TinySTM*) and one up to date GPU STM (*GPU-STM*) as well. With this comparison, we could have a first impression about the performance improvement of moving parallel scenario from CPU to the GPU, as well the advance of our system to other existing systems. After that, we compared our upgraded system which can also handle semantic conflict with *GPU-STM*. By means of these more comparison scenarios, we could view the improvement of new version, especially when semantic conflict is taken into account. Besides, as two more scenarios are applied to evaluate, the performance of our system could be tested when the conflict rate is relative higher (vacation) and the data structure is more complex (skip-list).

5.2 Evaluation of the first version

In this section we present results from a series of benchmarks to demonstrate the performance of our first version system. We compare the performance of *PR-STM*against a recently developed STM system for the GPU called *GPU-*

STM and a widely used STM system for the CPU called *TinySTM*. The tests were carried out on a desktop PC with Nvidia Fermi GPU (GeForce GTX 590) which has 16 SMs, operates at a clock frequency of 1225 MHz and has access to 1.5 GB of GDDR5 memory. All shared data and the *global lock table* are allocated in global memory, while all local meta-data is stored in thread local memory. The *global lock table* data accessed the L2 cache, while local memory accessed both the L1 and L2 caches. The CPU tests were carried out on 2 x dual-core 3.07GHz Intel(R) processors with 16GB of RAM. We used the Windows 7 Operating System. *TinySTM* used the Time Stamp Contention Management Policy with the Eager Write Back configuration (with invisible reads).

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

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

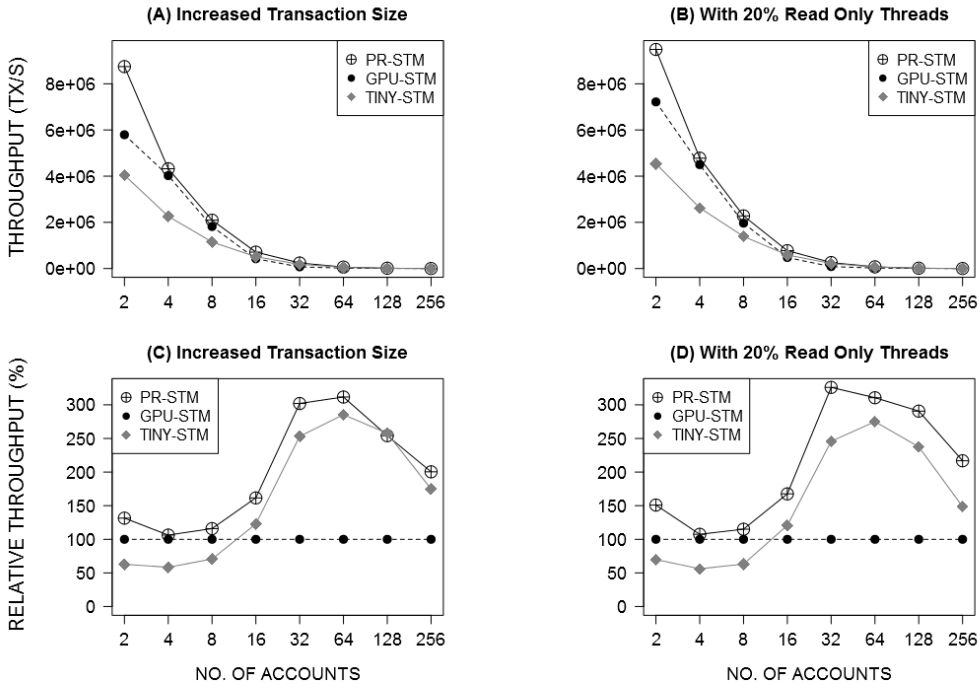


Figure 5.1: Average Throughput with Increasing Transaction Size

5.2.1 Transaction Throughput

Figure 5.1 shows the degree of transaction throughput when the number of accounts accessed per transaction is increased. The number of threads used was kept constant at 512 threads for the GPU and 8 threads for the CPU. These values were used as they provided the best performance in each system. In Graphs 5.1(A) and Graphs 5.1(B), Y-axes show the number of transactions committed per second and X-axes show the number of bank accounts accessed in each transaction. The higher this number is, the more difficult one transaction succeed to commits. As the GPU has many more threads than the CPU both *PR-STM* and *GPU-STM* outperform *TinySTM* when the number of accessed accounts is low (below 16). As expected, when the transaction size increases the throughput of all three STMs drops because inter-transaction conflicts are now more likely. The sharpest drop in performance is witnessed in *GPU-STM* as the higher thread numbers exacerbate the degree of conflicts. In the results with 20%

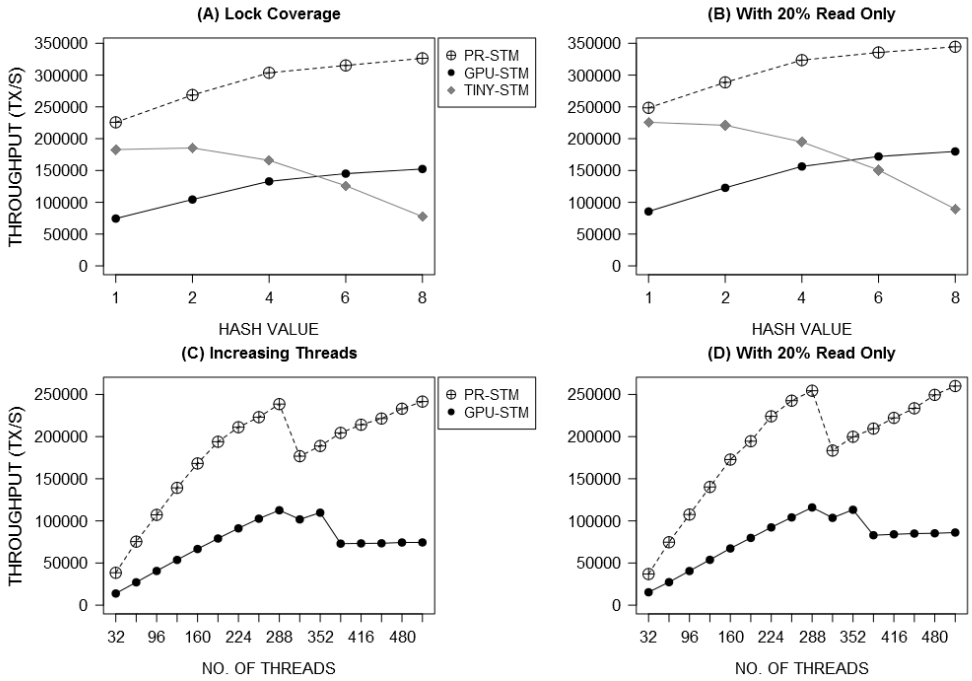


Figure 5.2: Average Throughput with Increasing Lock Coverage and Increased Threads

read only transactions (Graphs 5.1(B) and (D)) throughput is marginally better. This is because fewer locks are acquired and so fewer conflicts occur.

Graphs 5.1(C) and 5.1(D) show normalised throughput instead of the absolute values shown in Graphs 5.1(A) and 5.1(B). This helps to differentiate the performance when the transaction size increases beyond 16 accounts, where the values are too close to read in absolute terms. Y-axes show the relative throughput of *PR-STM* and *TinySTM* if we treat *GPU-STM* as 100%. With more accounts accessed we can see both *PR-STM* and *TinySTM* outperform *GPU-STM*. One possible reason for this is that our algorithm does not have to sort the local lock array at every read or write step (like *GPU-STM*) while the higher number of threads enjoyed by *PR-STM* remains a benefit to performance rather than a hindrance. However, for *GPU-STM*, when more accounts need to be access in one transaction, more sort step needs to be done during threads execution, therefore more unnecessary operations need to be abort if conflicts occur.

5.2.2 STM Scalability

Graphs 5.2(A) and 5.2(B) show the degree of transaction throughput when the hash function is modified. The hash function determines the number of accounts covered by a single lock; the lower the hash value the less chance that threads will try to access the same lock when reading or writing to different shared data. Both the number of threads used and the transaction size were kept constant at 512(GPU)/8(CPU) and 128 respectively. Once again the Y axes show the throughput in transactions per second and the X-axes show the hash function value as the number of accounts covered by a single lock.

Graphs 5.2(A) and 5.2(B) provide a comparison between *PR-STM*, *GPU-STM* and *TinySTM* with different hash values. As the hash value increases the performance of *TinySTM* deteriorates due to the increased likelihood of false conflicts. Both *PR-STM* and *GPU-STM*, however, show increased throughput. This is because *PR-STM* and *GPU-STM* can both take advantage of reduced lock-querying (due to their lock-sets) and memory coalescing to reduce bus traffic when querying the status of locks held. In Graph 5.2(B), with 20% read only threads, performance is only slightly improved in all three techniques, but mostly in *TinySTM* which gains the most benefit from invisible reads.

In Graphs 5.2(C) and 5.2(D), we increase the number of threads. In these two graphs we only compare the performance of *PR-STM* and *GPU-STM* because *TinySTM* is limited by the relatively small number of threads afforded by the CPU. Transaction throughput rises until 258 threads are used where inter-thread conflicts begin to occur at a substantial rate. Below 258 threads, the possibility of conflict is negligible because the high number of accounts used reduces the probability that threads will access the same account. As the number of threads increases, however, so too increases the rate of conflict and therefore the throughput decreases markedly. As thread numbers increase, however, *PR-STM* begins to improve once again, whereas *GPU-STM* levels out. The benefit of the work produced by extra threads is cancelled out by the overhead caused by inter-transactional contention. In Graph 5.2(D) we can see that performance improves marginally with the introduction of 20% read only threads. All other factors being equal, improvements in terms of read only transactions have little

effect on the GPU.

5.3 Evaluation of the latest version

To evaluate our latest system, we compare the performance of PR-STM new version with that achieved by GPU-STM. Evaluation is made in the context of transaction throughput and scalability (i.e. the response to increasing the number of threads and hash number) again. We also evaluate performance for generalised application usage (i.e. with semantic transactions present). In each case we compare results for instruction sets which have and have not been pre-ordered to avoid semantic conflict. We then present results which evaluate the performance of each CMP as the ratio of semantic transactions is increased from zero to 100%.

Three scenarios are used to benchmark our technique against GPU-STM to demonstrate that PR-STM is a generalised solution. The benchmarks *Bank* (a simple benchmark provided as part of *tinySTM*), *Vacation (from Stanford Transactional Applications for Multi-Processing (STAMP))* and *SkipList* (a frequently used benchmark for parallel computing) are used. *Bank* is a relatively straightforward scenario showing performance in a simple application. *Vacation* involves a much higher conflict rate due to the fixed number of room types being accessed by a high number of threads. *SkipList* is a commonly used benchmark for parallel computing systems which tests performance with a more complex data structure. With these tree benchmarks the performance of the system could be inspected with both low contention rate and high contention rate, which can indicate the excellence in any condition, and even with more complex data structure such as skip list, it can also express a decent result.

For each benchmark scenario, results are presented as three parameters are varied:

- The number of threads. While the overall computing power is raised with an increase in the number of threads utilised, the chances of conflict is also increased, as threads are more likely to compete for the shared resource. The default number of threads used in the experiments is 1280.

- The lock coverage. Each lock covers a greater amount of shared resource; the potential for conflict increases but there is a reduction in the memory requirement. The default hash value used in the experiments is 1.

- The ratio of semantic transactions. The performance of each system is assessed as the amount of semantic conflict introduced by the application increases. The default semantic transaction ratio used in the experiments is 100% (representing generalised usage).

For each of these experiments, four graphs are presented comparing the throughput of PR-STM and GPU-STM. The graphs show results from experiments where the transactions are either all read/write or 20% read-only, and for each of these cases results are shown with and without a stage when the transactions are presorted to avoid semantic conflict (replicating the task of the application programmer).

All experiments were carried out on a desktop PC with a CPU (i7-4770) running at 3.4 GHz. The GPU was a NVidia GeForce GTX 780 Ti with a clock speed of 1045 MHz, 3 Gb of GDDR5 memory and 15 streaming multiprocessors. The operating system was Win8. The two CMPs (PR-STM and GPU-STM) were implemented with the CUDA 7.5 runtime library.

The shared data, including global lock table, are allocated in off-chip global memory. In the PR-STM implementation, local metadata is stored in thread-local memory, whereas for GPU-STM, metadata is stored in global memory but the pointers to the data are in local memory. For both implementations, the local metadata is cached at the L1 and L2 levels, and the global data is cached only at the L2 level as the L1 cache is not coherent.

5.4 Implementation of Bank Benchmark

The *bank* scenario was the first to be used to benchmark the performance of the two systems. *Bank* consists of an array of bank account structures, and allows the execution of a number of transaction types on these simulated bank accounts (deposit, withdraw, etc).

As many threads are available on GPU a sizeable number of accounts were created in shared data for the *bank* scenario. A memory block of 10MB was set aside for the creation of roughly 2.5million accounts. This allows the observation of the effects of both low and high contention as the scenario parameters are varied. The *bank* scenario was adapted to our needs in a number of ways:

- The hashing function used by both CMPs was modified so that the amount of shared data covered by a single lock can be varied. This allows investigation into the amount of false sharing.
- Results are included where the number of threads is increased, to observe the contention caused by the high numbers of threads available on GPU.

Results are shown for the case when all threads perform update transactions (i.e. read and write operations), and also the case when 20% of threads execute read-only transactions. This approach allows the impact of invisible reads in the scenarios to be analysed. Each test lasted for 5 seconds, and was executed 10 times with the average results presented.

5.4.1 Performance with Bank Benchmark

The graphs in Figure 5.3, 5.4 and 5.5 show the transaction throughput achieved when the number of threads available for resolution is increased. In each case the performance of our PR-STM is compared to that of GPU-STM on the GPU. Graphs in Figure 5.3 and 5.4 show the results for generalised application (i.e. 100% semantic transactions with no pre-ordering). The transaction tables are completely random generated so that semantic conflicts may occur randomly. As GPU-STM is not designed to handle semantic conflict it performs poorly in these circumstances (the y axes are presented as logarithmic in Graph B to better distinguish the performance of GPU-STM as the thread count increases). Graphs in Figure 5.5 shows results where the application's instructions have been pre-ordered to avoid semantic conflict, so that the transactions are more suited to GPU-STM, allowing us to assess both CMPs in a less generalised context.

The number of threads available was varied between 256 and 2506. The throughputs achieved by PR-STM and GPU-STM for the cases when all threads

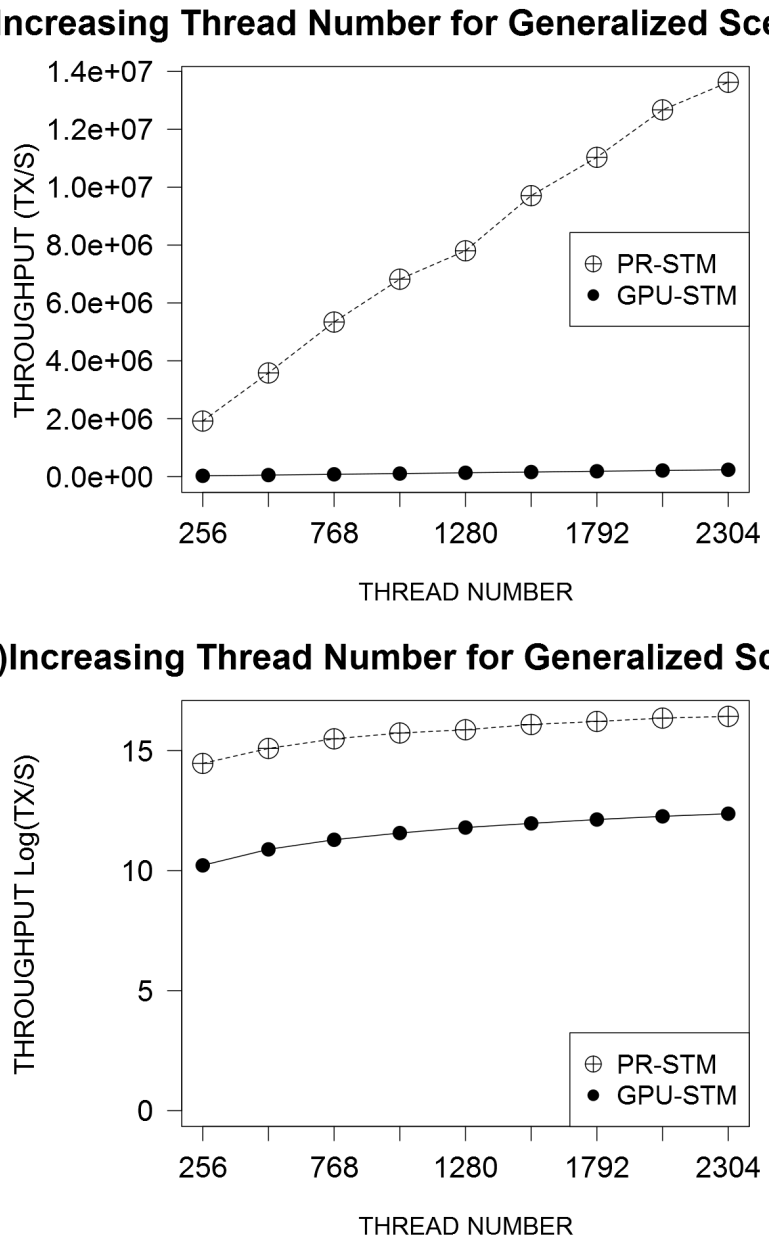


Figure 5.3: Generalized bank scenario average throughput with increasing number of threads and all read/write transactions. Graph A shows the value of transactions fulfilled in 1 second to observe the difference between throughputs, whereas graph B shows the Log of throughput to view the variation trend when thread numbers changes clearly.

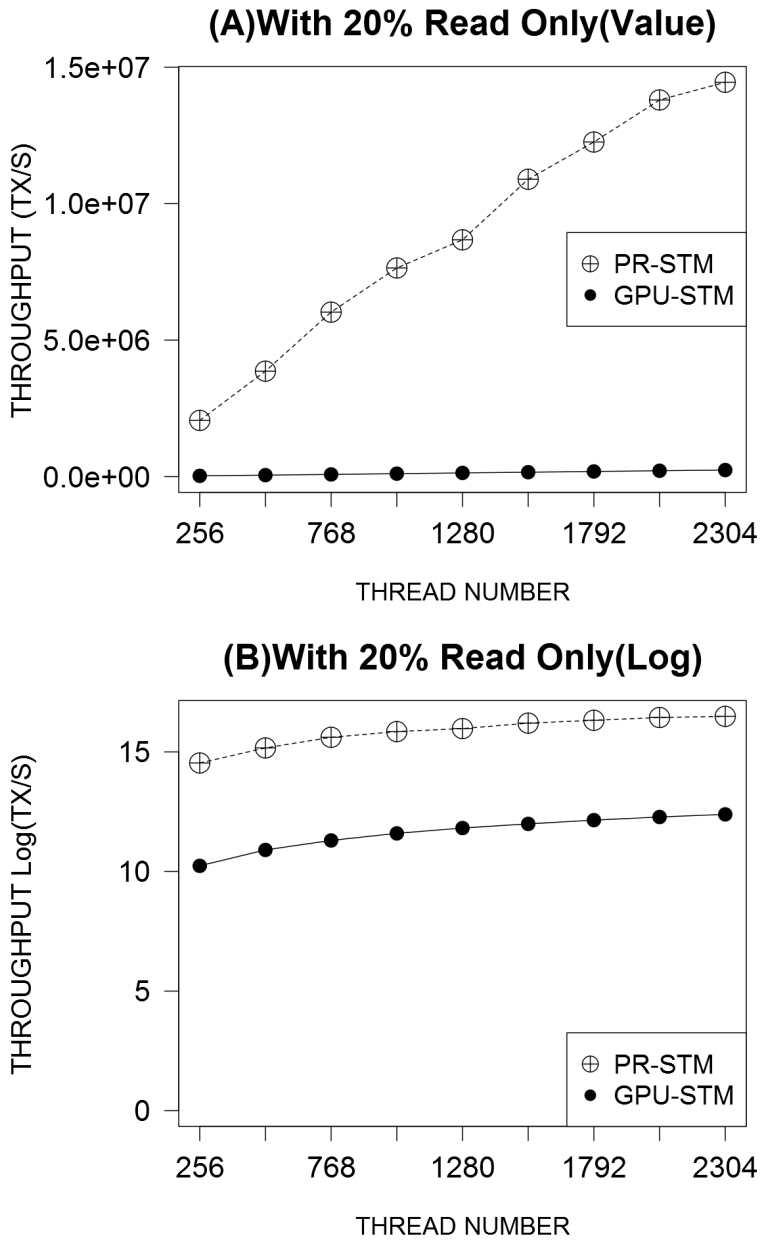


Figure 5.4: Generalized bank scenario average throughput with increasing number of threads and 20% read-only transactions. Graph A shows the value transactions fulfilled in 1 second to observe the difference between throughputs, whereas graph B shows the Log of throughput to view the variation trend when thread numbers changes clearly.

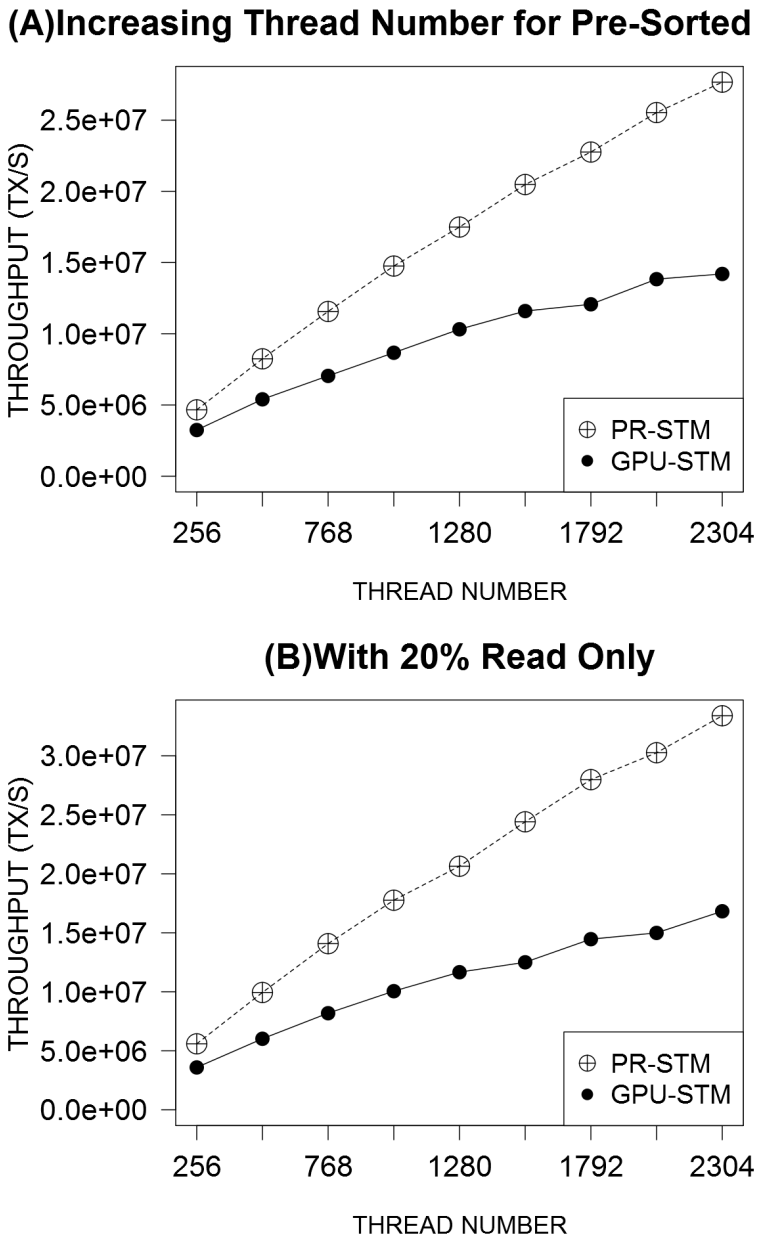


Figure 5.5: Pre-sorted bank scenario average throughput with increasing number of threads. The transaction tables are sorted by programmer with a certain order to avoid semantic conflict. Graphs A has all read/write transactions, graphs B has 20% read-only transactions.

perform read and write transactions, and when 20% of threads perform read-only transactions respectively are presented. In all cases PR-STM outperforms GPU-STM. The introduction of 20% read-only threads causes only a minimal change in throughput in both cases, so it appears that read-only transactions have little effect on GPU performance.

Increasing the number of threads means there is an increase in the available computing resource. and our results show an increase in throughput which is a little less than linear. This is because as more threads work in parallel, the conflict rate also increases (as the total shared resource is steady). The scalability of PR-STM is demonstrated by the increase in throughput, compared to GPU-STM across all thread numbers. An important aspect of a GPU based solution is it can launch a much greater number of threads than a CPU solution, so scalability is of interest.

Tests were also carried out with a modified hash function, which determines the number of accounts that can be covered by a single lock. The lower a hash value, the less chance of a thread trying to access the same lock when reading or writing to different shared data. The number of threads remained at 1280. The results are shown in Figure 5.6, 5.7 and 5.8 for the cases when all threads perform read and write transactions, and when 20% of threads perform read-only transactions respectively. The y-axes show the number of transactions per second, and the x-axes show the hash function value, which is the number of accounts covered by a single lock.

A decrease in throughput is seen in both GPU-STM and PR-STM as the number of accounts per lock is increased. The performance of both of the CMPs for GPU decreases because of increased false conflict, but reduced lock querying counters this somewhat (as both CMPs use lock-sets), as does reduced bus traffic when querying the status of those locks (due to coalescence of memory). When 20% of the threads are read-only, throughput is only slightly improved for both CMPs.

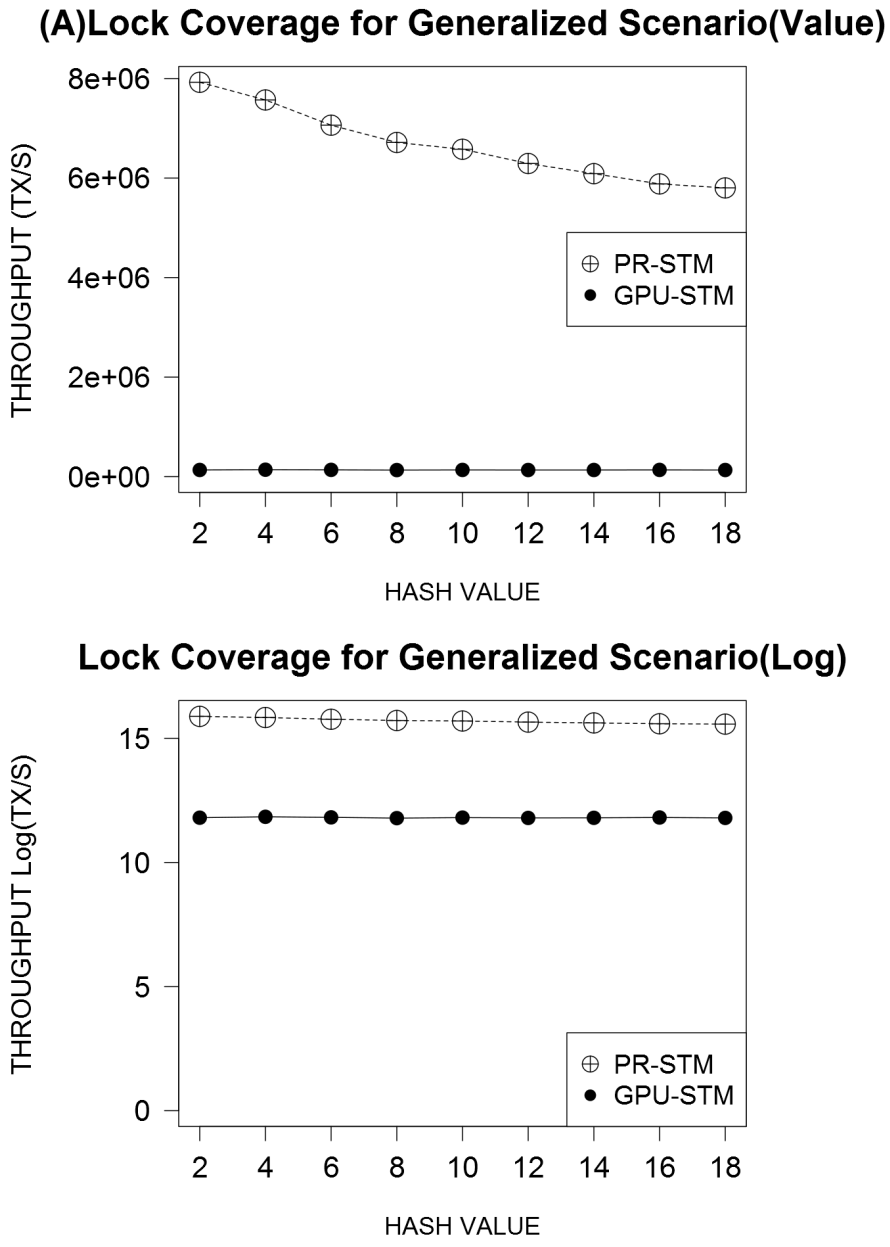


Figure 5.6: Generalized bank scenario average throughput with increasing lock coverage and all read/write transactions. Graph A shows the value of transactions fulfilled in 1 second to observe the difference between throughputs, whereas graph B shows the Log of throughput to view the variation trend when thread numbers changes clearly.

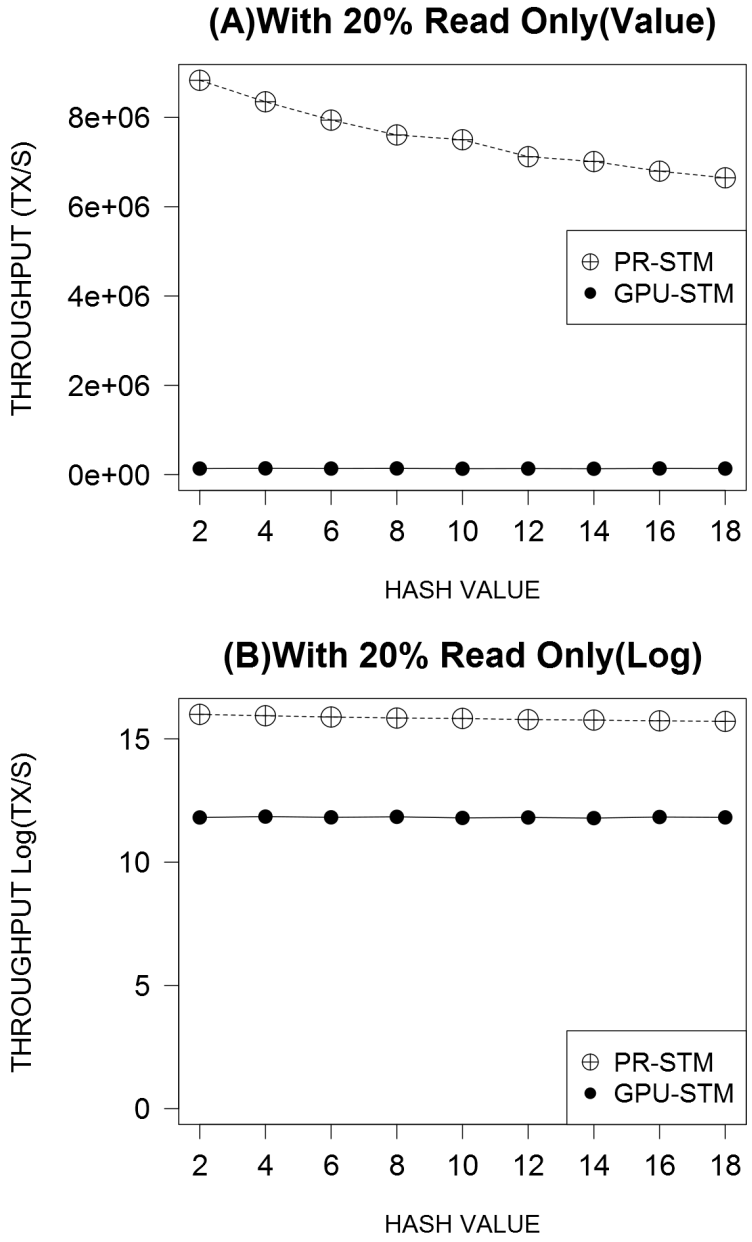


Figure 5.7: Generalized bank scenario average throughput with increasing lock coverage and 20% read-only transactions. Graph A shows the value transactions fulfilled in 1 second to observe the difference between throughputs, whereas graph B shows the Log of throughput to view the variation trend when thread numbers changes clearly.

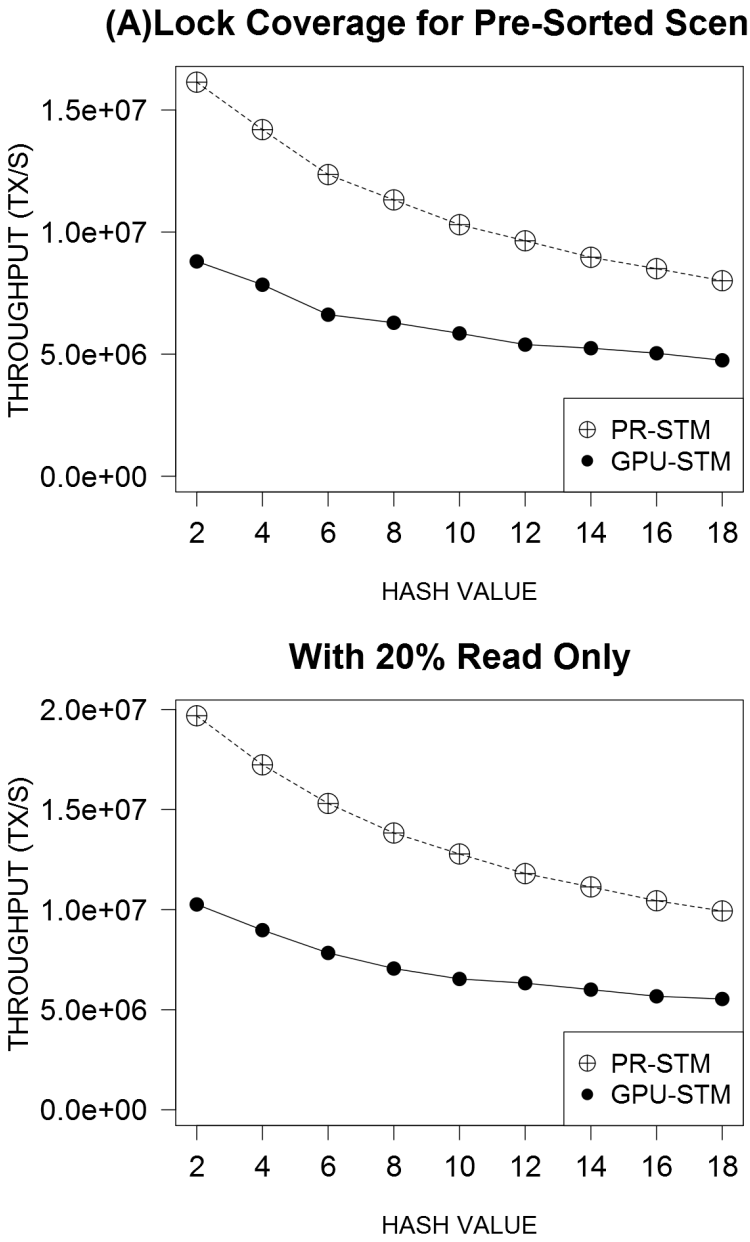


Figure 5.8: Pre-sorted bank scenario average throughput with increasing lock coverage. The transaction tables are sorted by programmer with a certain order to avoid semantic conflict. Graphs A has all read/write transactions, graphs B has 20% read-only transactions.

5.4.2 Throughput of Semantic Transactions

We now consider performance under varying levels of semantic conflict to assess the performance of our algorithm in comparison to that of GPU-STM in situations where transactional conflict is caused by the semantics of the application. Figure 5.9, 5.10 and 5.11 shows the rate of transaction throughput as the semantic transaction ratio is increased from 0% to 100% for two situations. In the first the application has not pre-ordered transactions to avoid semantic conflict,whereas in the second that ordering has occurred. The number of threads used was kept constant at 1280 and the hash number was 1 (i.e. every account gets a lock to avoid false conflict). In each graph, Y-axes show the number of transactions committed per second and X-axes show the percentage of semantic transactions in the transaction table.

Figures in 5.11 show results after the transaction table has been sorted to avoid semantic conflict. This allows us to compare performance of GPU-STM and PR-STM in a situation which both are expected to handle - the onus on sorting the transactions is left to the application programmer. For example, a semantic conflict could be resolved by ordering a withdrawal transaction shortly before a deposit transaction, thus allowing the withdrawal to succeed with minimum retries. PR-STM produced higher throughput than GPU-STM even when the transactions were sorted. As the semantic conflict rate was increased GPU-STM decreased in throughput very slightly while PR-STM began to improve. This is because PR-STM can benefit from temporarily abandoning transactions with semantic conflicts and searching for new transactions, causing fewer conflicts overall.

The graphs in Figures 5.9 and 5.10 show the results with no sorting of the transaction table (i.e. for a generalised situation). In this case, GPU-STM cannot deal with semantic transactions - the simulation aborts after 10,000 failed attempts. Introduction of semantic conflicts representing as little as 5% of the overall transactions shows a very rapid decline in performance. Note that the y-axes in graphs B are logarithmic to better illustrate the relative performance of GPU-STM as the ratio of semantic conflict increases. PR-STM, however, copes well with the increased ratio of semantic conflict, with little degradation after the

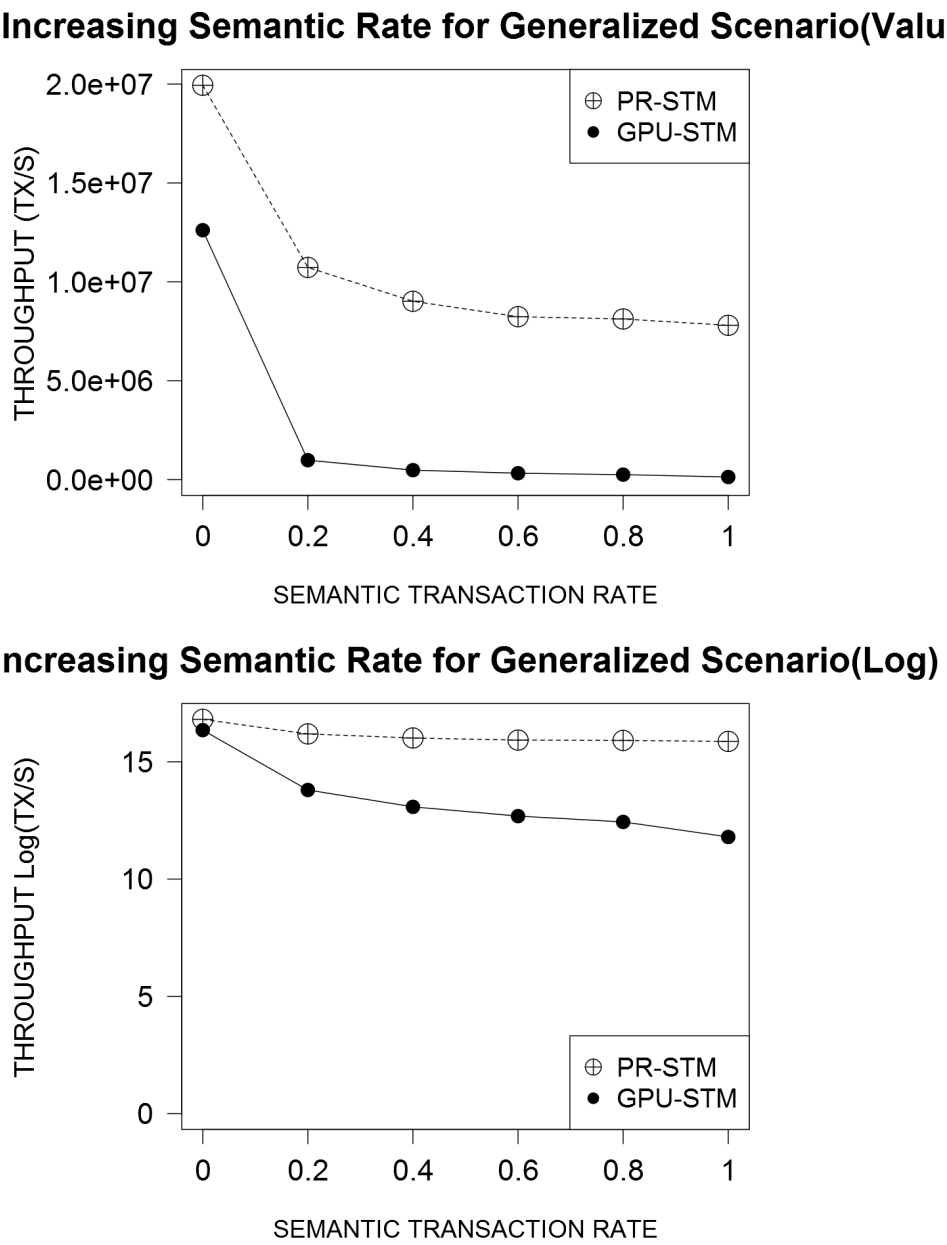


Figure 5.9: Generalized bank scenario average throughput with increasing ratio of semantic transactions and all read/write transactions. Graph A shows the value of transactions fulfilled in 1 second to observe the difference between throughputs, whereas graph B shows the Log of throughput to view the variation trend when thread numbers changes clearly.

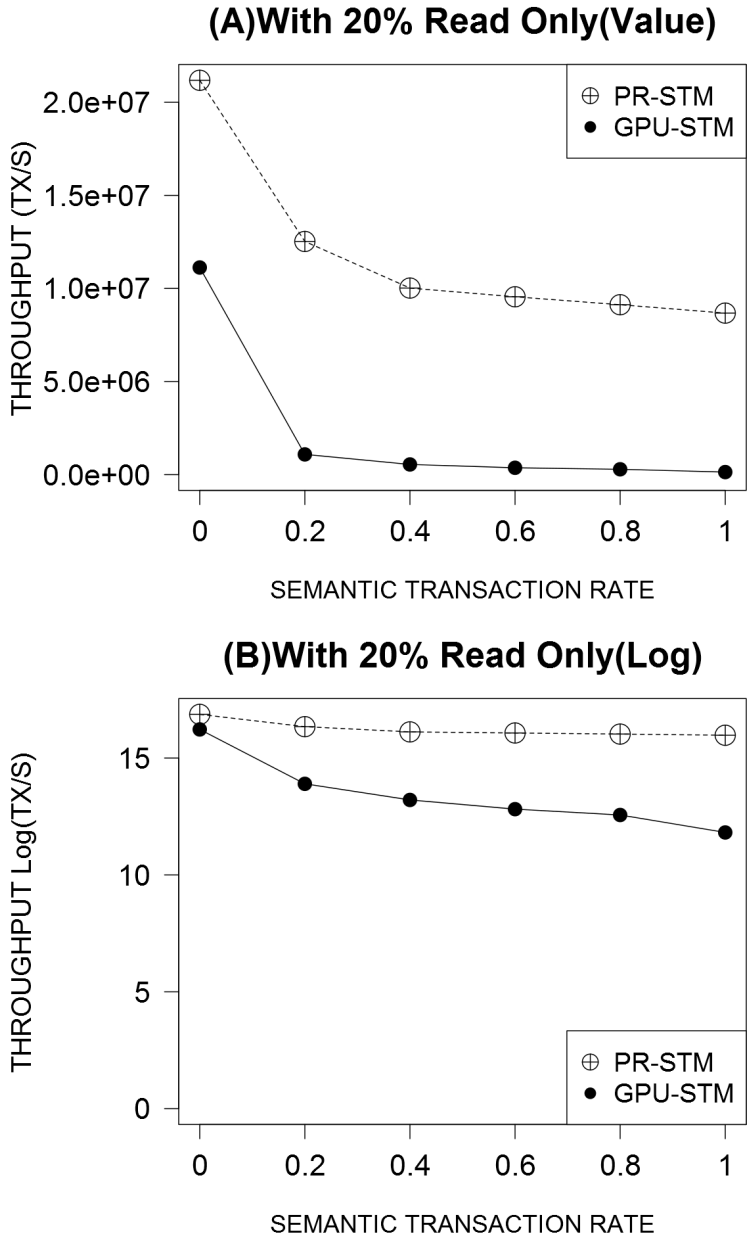


Figure 5.10: Generalized bank scenario average throughput with increasing ratio of semantic transactions and 20% read-only transactions. Graph A shows the value transactions fulfilled in 1 second to observe the difference between throughputs, whereas graph B shows the Log of throughput to view the variation trend when thread numbers changes clearly.

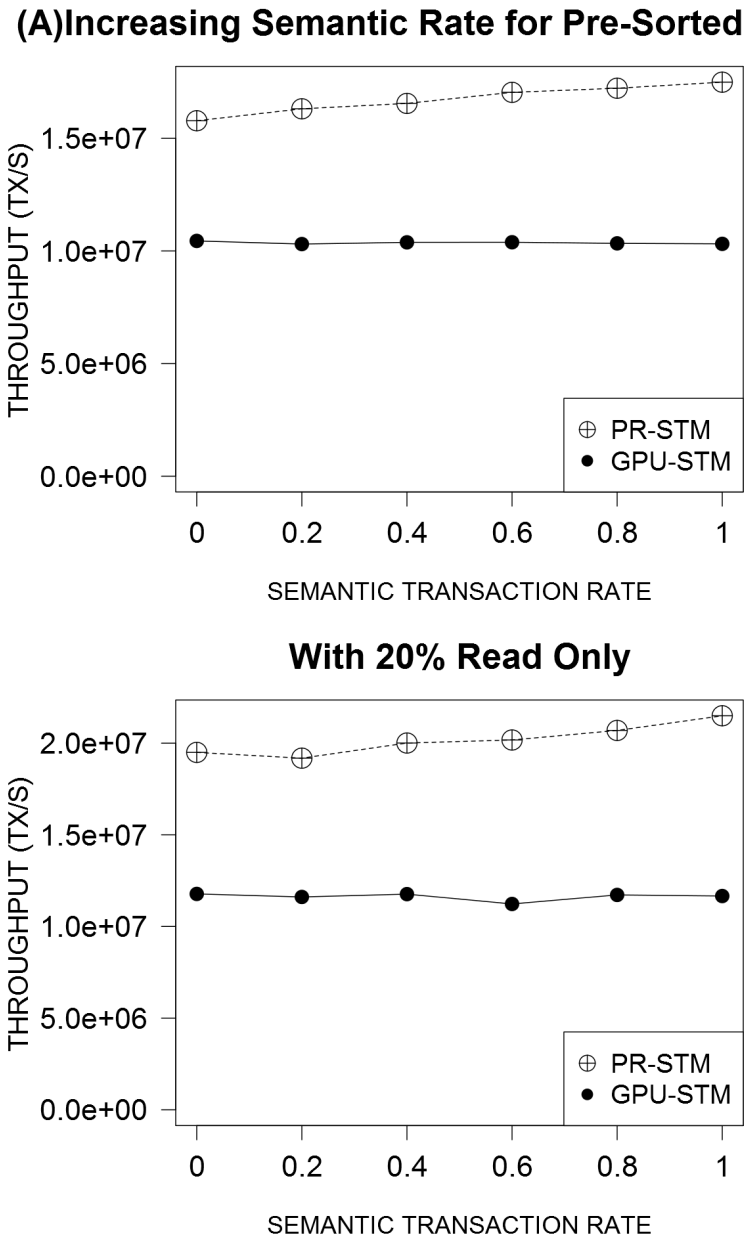


Figure 5.11: Pre-sorted bank scenario average throughput with increasing ratio of semantic transactions. The transaction tables are sorted by programmer with a certain order to avoid semantic conflict. Graphs A has all read/write transactions, graphs B has 20% read-only transactions.

initial drop-off.

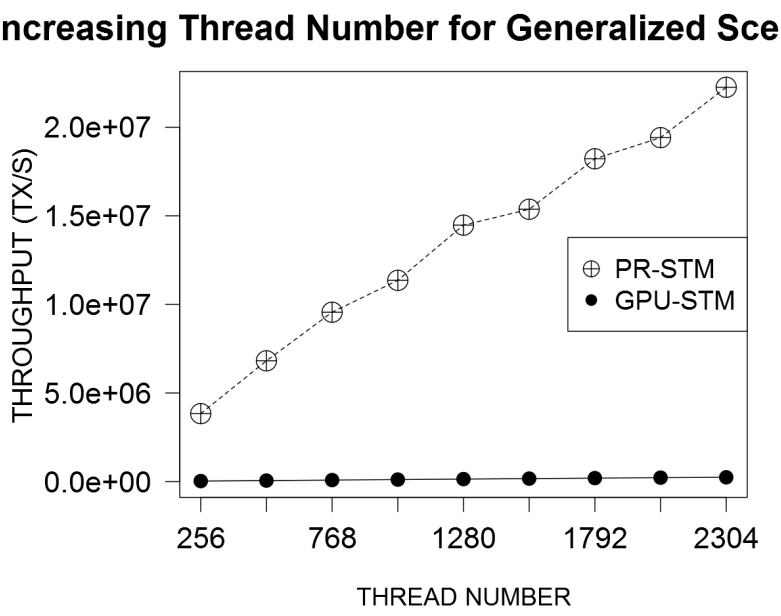
While there is an increased cost in PR-STM's ability to handle semantic conflict, the fact that it can handle any form of conflict makes it an appealing solution. The onus on the application programmer to correctly order transactions to provide no semantic conflict is removed, and a generalised solution is provided.

5.5 Implementation of Vacation Benchmark

The *Vacation* scenario implements a hotel room coordination system which handles the booking and cancelling of certain types of hotel room concurrently. It is commonly used as part of the STAMP benchmarking suite. The *vacation* scenario involves transactions which tend to execute more statements of greater complexity than those in the *bank* scenario. And one important fact of *vacation* scenario is the contention rate is much higher than *bank* scenario as there is only a small number of hotels, but all transaction need to deal with at least one from them. This high contention rate leads the overall performance of *vacation* scenario is worse than the *bank* scenario, but compare with other systems, our system is still second to none.

As the *vacation* benchmark is designed for CPU implementation, some adjustments were made to ensure compatibility with GPU operation. The controllable variables available in the *Vacation* scenario for evaluation include the hash number, and the number of transactions per thread (TPT). This provides the base environment for the performance testing. Both GPU-STM and PR-STM use a hash number approach to explicitly control how many memory addresses share one lock. With a higher hash number, a greater contention rate can be expected. The TPT is used to examine the vitality of a thread. The more transactions a thread executes, the longer it is active, which allows us to evaluate the ability a thread has to proceed through different concurrent situations.

In the *Vacation* scenario, semantic conflict occurs when a customer account attempts to book a type of room that is sold out. For sequential computing, it is easy to deal with this eventuality - the unsatisfied customer is added to a waiting list, and when a suitable room becomes available the first customer on the waiting list is allocated that room. However, in the context of parallel computing, there



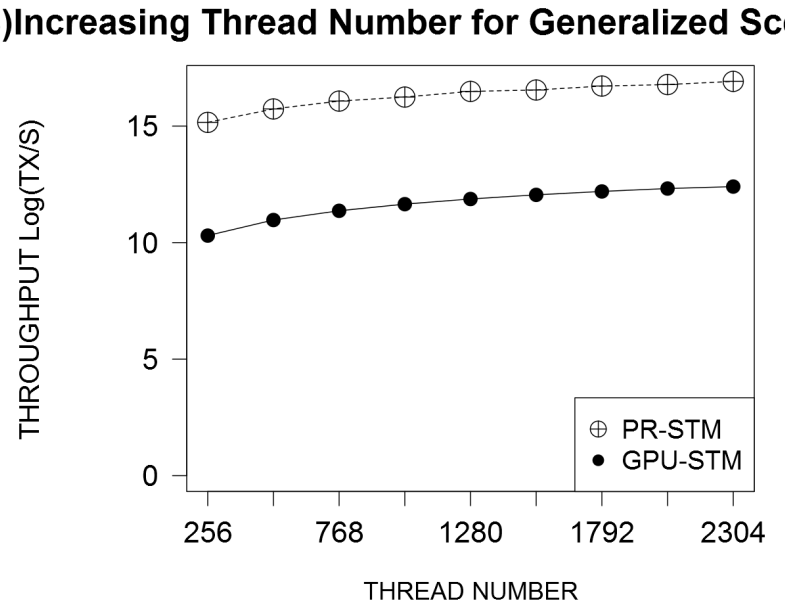


Figure 5.12: Generalized vacation scenario average throughput with increasing number of threads and all read/write transactions. Graph A shows the value of transactions fulfilled in 1 second to observe the difference between throughputs, whereas graph B shows the Log of throughput to view the variation trend when thread numbers changes clearly.

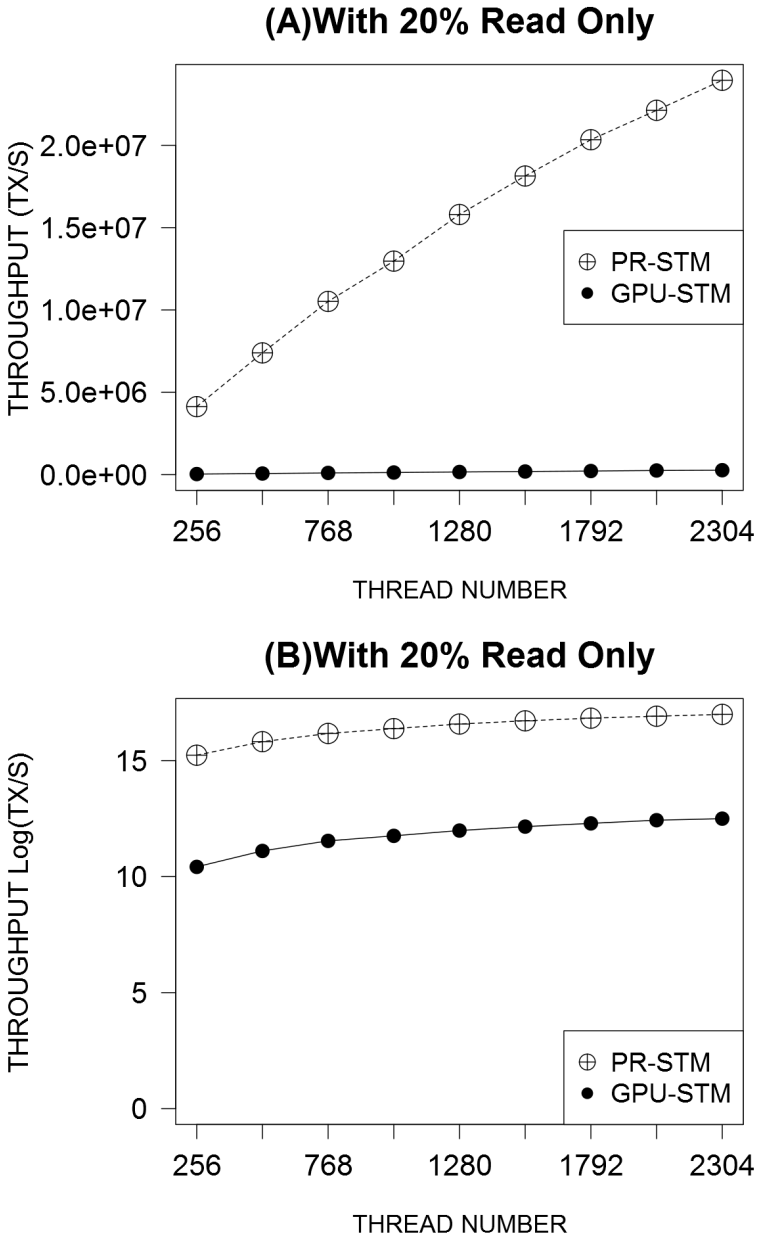


Figure 5.13: Generalized vacation scenario average throughput with increasing number of threads and 20% read-only transactions. Graph A shows the value transactions fulfilled in 1 second to observe the difference between throughputs, whereas graph B shows the Log of throughput to view the variation trend when thread numbers changes clearly.

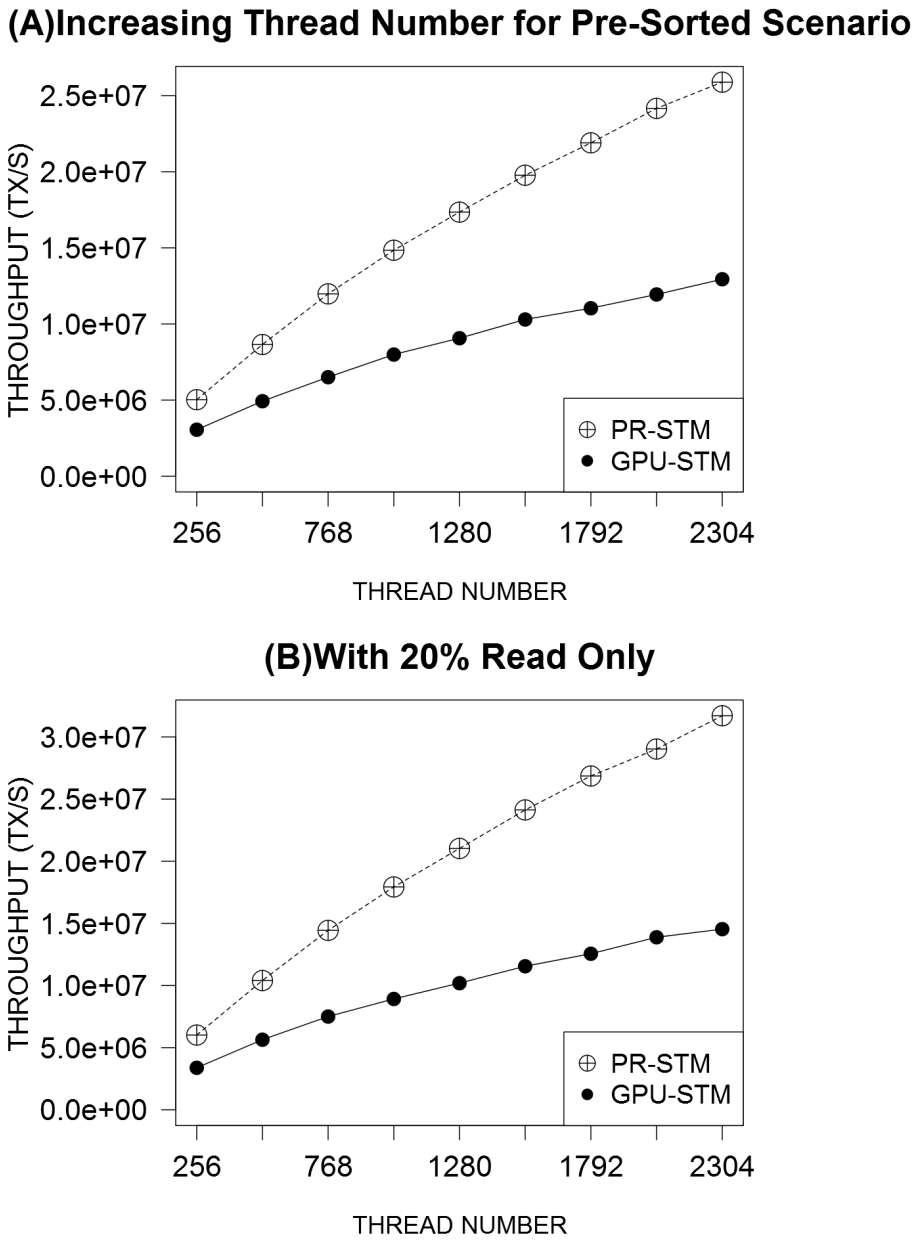


Figure 5.14: Pre-sorted vacation scenario average throughput with increasing number of threads. The transaction tables are sorted by programmer with a certain order to avoid semantic conflict. Graphs A has all read/write transactions, graphs B has 20% read-only transactions.

is no trivial solution to this issue because the instructions are executed in an arbitrary order and the threads are isolated from each other. One way to handle this issue is to have the threads that want to book an unavailable room listen to the account of the type of room. Once the type of room is available again, all the threads compete for it through atomic operation, with one being selected to be successful.

PR-STM provides a solution to semantic conflict. As the program runs through the instruction table, if a semantic conflict occurs, the instruction is added to an array of instructions which have incurred semantic conflict. After the kernel finishes the whole instruction table, it will continue to execute instructions from the recorded array until all the semantic conflicts are resolved or abandoned.

GPU-STM provides no semantic conflict solution. To make benchmarking more scalable for this evaluation, a simple semantic conflict solution has been added to GPU-STM. If conflict occurs(no matter concurrent conflict or semantic conflict), the instruction will not be aborted instantly. Instead it is given a limited number of chances to retry. Although it should retry infinite times when encounter concurrent conflict because concurrent conflict will eventually disappear after other threads close their transactions, as GPU-STM can not classify concurrent conflict and semantic conflict, a fixed retry time is necessary. The number of chances should be adjusted according to the size of data. As we use a fixed size of shared data, the retry limit for GPU-STM is set to 1000 for all benchmarks.

5.5.1 Performance in Vacation Benchmark

The results from the evaluation of PR-STM and GPU-STM in the *vacation* benchmark are shown in Figures 5.12, 5.13, 5.14, and 5.15, 5.16, 5.17 and 5.18, 5.19, 5.20. Again each set of results consists of six graphs, showing scenarios with and without pre-sorting of transactions to remove semantic conflict, scenarios with 100% read/write transactions or 20% read-only transactions, and logarithmic values for generalized scenarios. In each case the throughput achieved by GPU-STM and PR-STM is compared, as the number of threads, hash value and

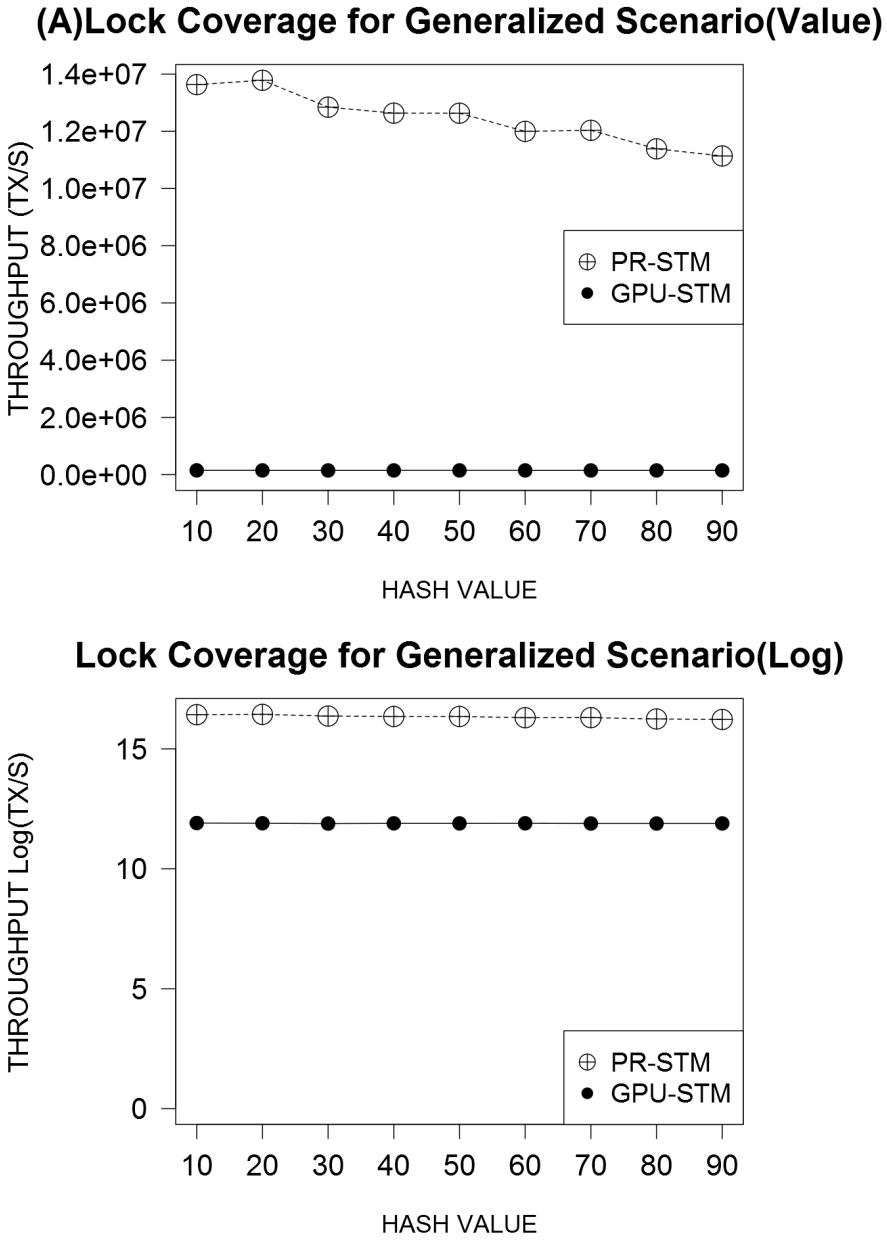


Figure 5.15: Generalized vacation scenario average throughput with increasing hash number and all read/write transactions. Graph A shows the value of transactions fulfilled in 1 second to observe the difference between throughputs, whereas graph B shows the Log of throughput to view the variation trend when thread numbers changes clearly.

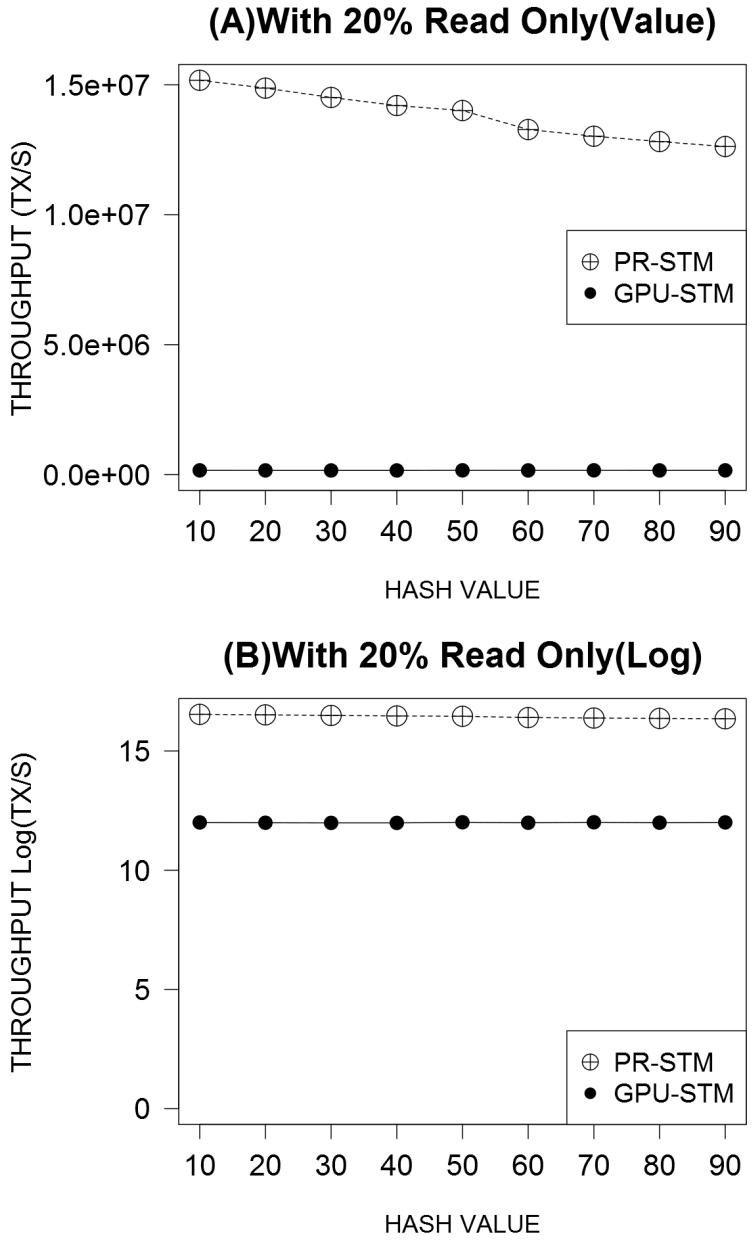


Figure 5.16: Generalized vacation scenario average throughput with increasing hash number and 20% read-only transactions. Graph A shows the value transactions fulfilled in 1 second to observe the difference between throughputs, whereas graph B shows the Log of throughput to view the variation trend when thread numbers changes clearly.

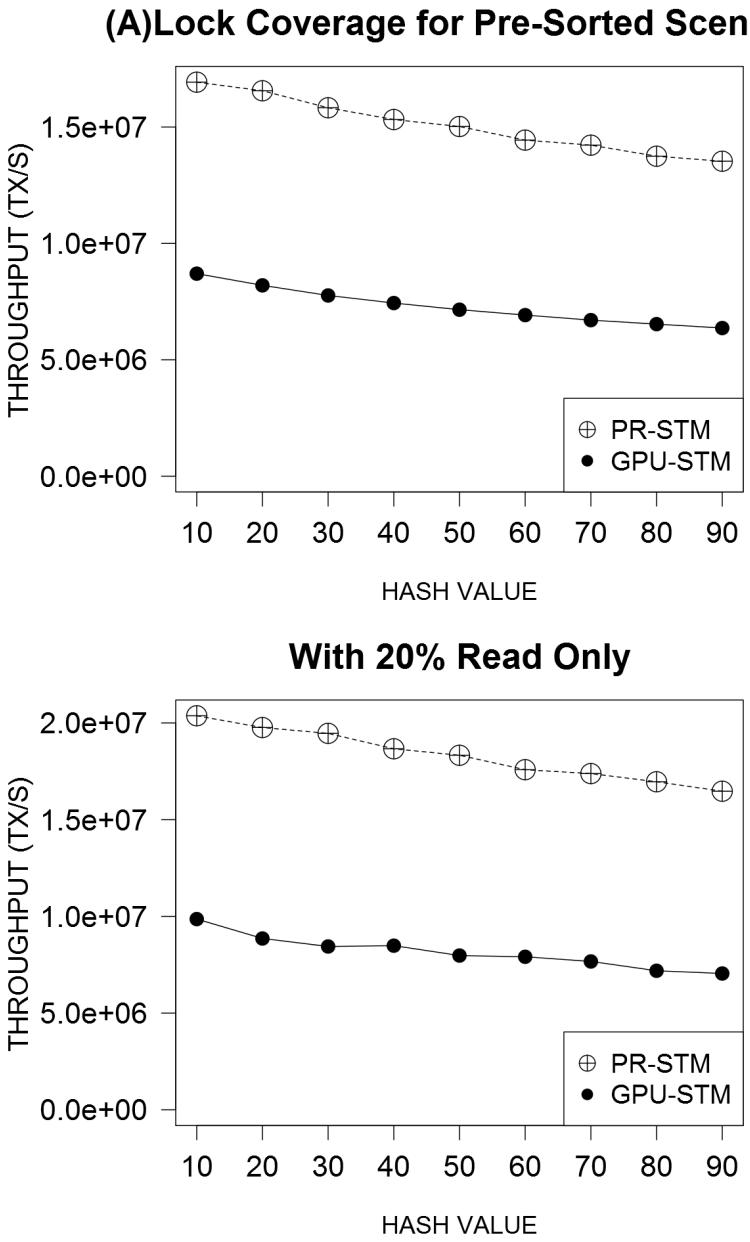


Figure 5.17: Pre-sorted vacation scenario average throughput with increasing hash number. The transaction tables are sorted by programmer with a certain order to avoid semantic conflict. Graphs A has all read/write transactions, graphs B has 20% read-only transactions.

ratio of semantic transaction respectively is increased.

The *vacation* benchmark scenario utilised consists of 150,000 customers attempting to book rooms of 1,000 different room types (with 150 rooms of each type available). For a booking transaction, a random customer is allocated a random room type to attempt to book. For a cancellation transaction, an already-booked customer is selected at random to cancel a room of the selected room type. The choice of hash number in each algorithm gives a certain number of consecutive data addresses only one lock, so it is necessary to make the room type account non-consecutive. Otherwise, the processing will simply occur sequentially. Therefore, we set the interval between each room type account to 100.

As with the results from the *bank* benchmark, the throughput achieved by PR-STM is greater than that achieved by GPU-STM in all cases of varying the number of available threads (Figure 5.12, 5.13, and 5.14) and the hash value (Figure 5.15, Figure 5.16 and Figure 5.17). The *vacation* benchmark involves considerably more contention(because all transactions have to access the small amount number of memory locations that present hotel) than observed in the *bank* benchmark, as well as more complex transactions. Again the introduction of 20% read-only transactions has minimal effect on throughput performance of either of the GPU solutions.

Figures 5.18, 5.19 and 5.20 show the results of increasing the rate of semantic transactions in the *vacation* benchmark for both PR-STM and GPU-STM. Graphs A and B show the results for a generalised situation (i.e. the transactions are not pre-sorted to avoid semantic conflict). Note that the y axes are logarithmic to allow more insight into the performance of GPU-STM in the presence of semantic conflict. As expected, GPU-STM struggles to deal with even low levels of semantic conflict, while PR-STM maintains its throughput up to the point where all transactions are of a semantic nature. Again the introduction of read-only transactions has little effect on the throughput. Graphs C and D show the results when the transactions have been pre-ordered to avoid semantic conflict (mimicking an onus on the application programmer for CMPs which are not generalised to handle semantic conflict). And the stability of performances of both systems can prove that the transaction table is pre-ordered so that semantic conflicts are removed prior to execution. Again PR-STM out-performs

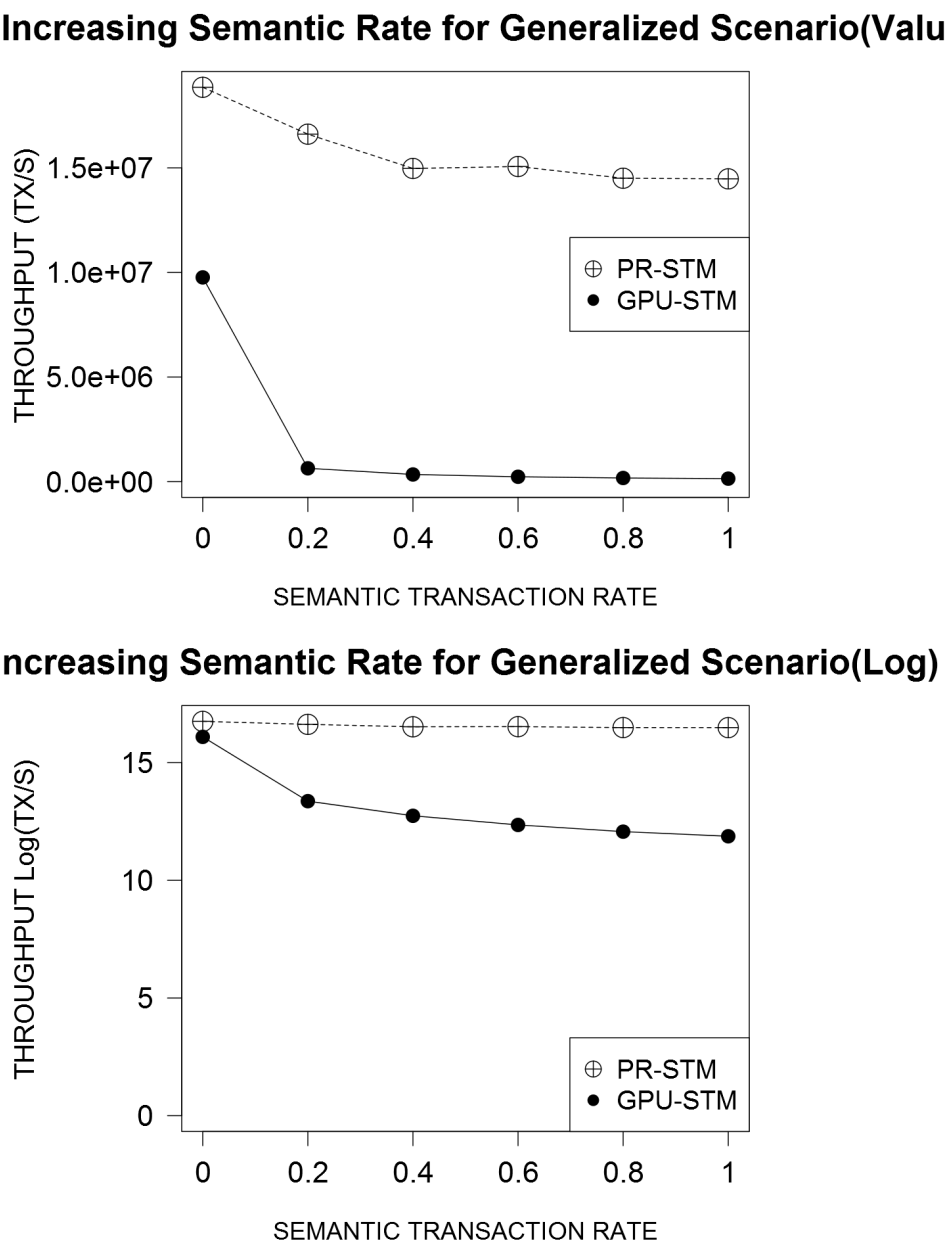


Figure 5.18: Generalized vacation scenario average throughput with increasing semantic transactions and all read/write transactions. Graph A shows the value of transactions fulfilled in 1 second to observe the difference between throughputs, whereas graph B shows the Log of throughput to view the variation trend when thread numbers changes clearly.

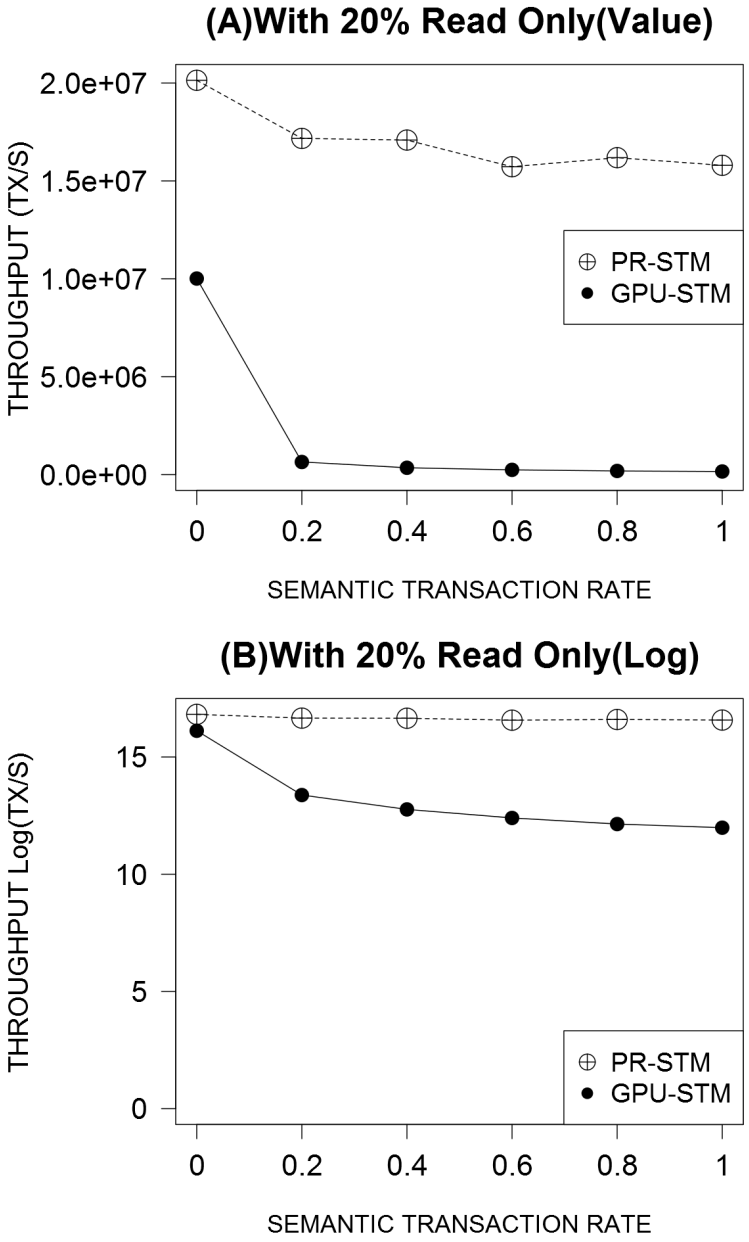


Figure 5.19: Generalized vacation scenario average throughput with increasing semantic transactions and 20% read-only transactions. Graph A shows the value transactions fulfilled in 1 second to observe the difference between throughputs, whereas graph B shows the Log of throughput to view the variation trend when thread numbers changes clearly.

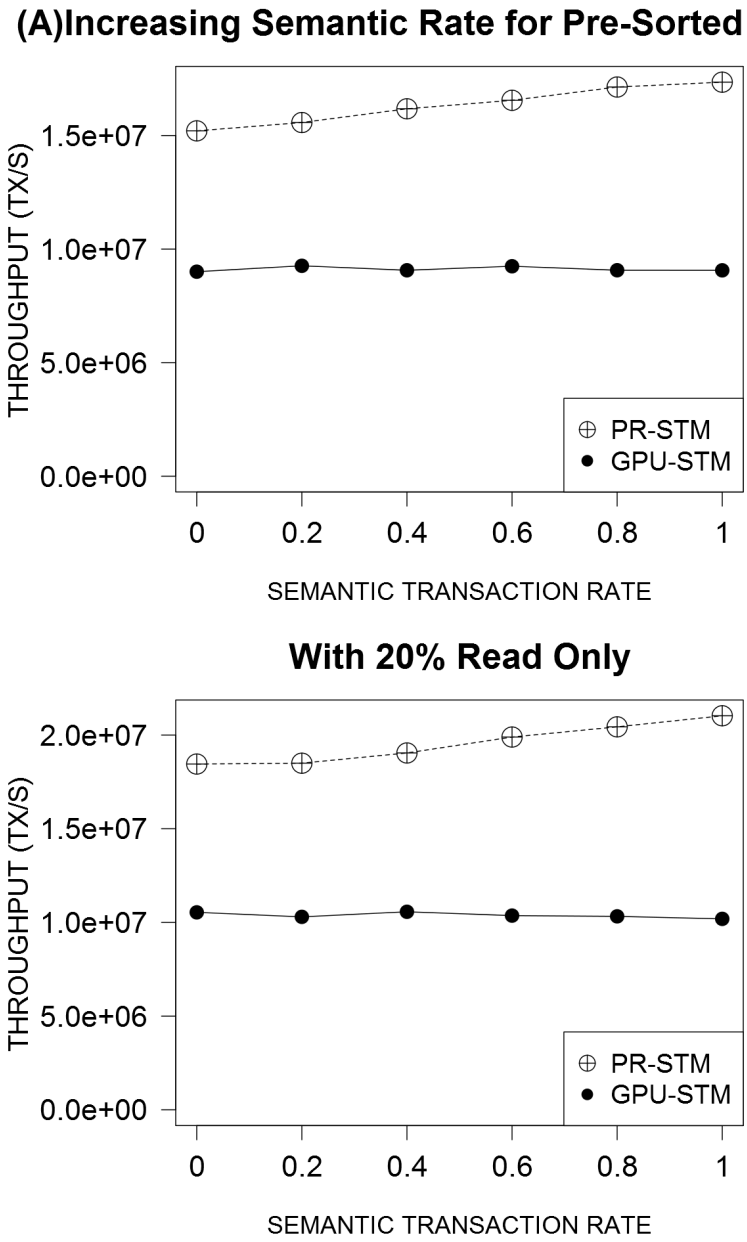


Figure 5.20: Pre-sorted vacation scenario average throughput with increasing semantic transactions. The transaction tables are sorted by programmer with a certain order to avoid semantic conflict. Graphs A has all read/write transactions, graphs B has 20% read-only transactions.

GPU-STM even the transactions are sorted.

5.6 Implementation of SkipList Benchmark

A skip-list is, effectively, a hierarchical link-list. The *skiplist* benchmark is commonly used to assess contention resolution in transactional memory. In our implementation of the benchmark we use an initial array of 5,000 entries, with sufficient memory allocated to expand this to 10 million entries (so that there are no allocation issues). The maximum number of hierarchical levels is set to 5. Three types of transactions are invoked by threads at random - a thread may insert a new element into the skip-list, delete an element, or simply search an element (i.e. a read-only transaction).

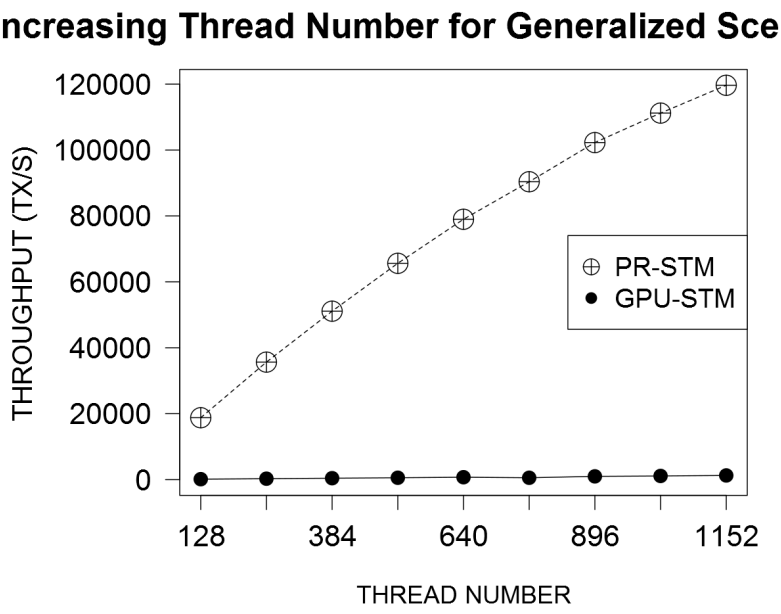
The use of *skiplist* as a benchmark allows us to further assess the scalability of the CMP. The hierarchical nature of the skip-list means that, in order to complete an insert or delete transaction, multiple nodes must be read from and written to. A higher rate of lock contention can therefore be reached than with the other two benchmarks considered.

5.6.1 Performance of SkipList Benchmark

The concept of semantic transactions doesn't apply to the *skiplist* benchmark, as the application can't attempt to insert a duplicate entry or remove a non-existent one. Evaluation was therefore limited to increasing the threads count and the hash number, allowing an assessment of PR-STM compared to GPU-STM for an application to which both are suited.

Figures 5.21, 5.22 and 5.23 shows the throughput achieved by both CMPs in the *skiplist* benchmark as the number of threads is increased. The hash number was again set to 1 for these experiments. As there is no semantic conflict, the performance of GPU-STM is improved in comparison to that achieved in the *bank* and *vacation* scenarios. However PR-STM still achieves higher throughput in all cases. Again the introduction of 20% read-only transactions has a minimal effect on throughput for both CMPs.

The results from evaluating performance as the hash number is increased



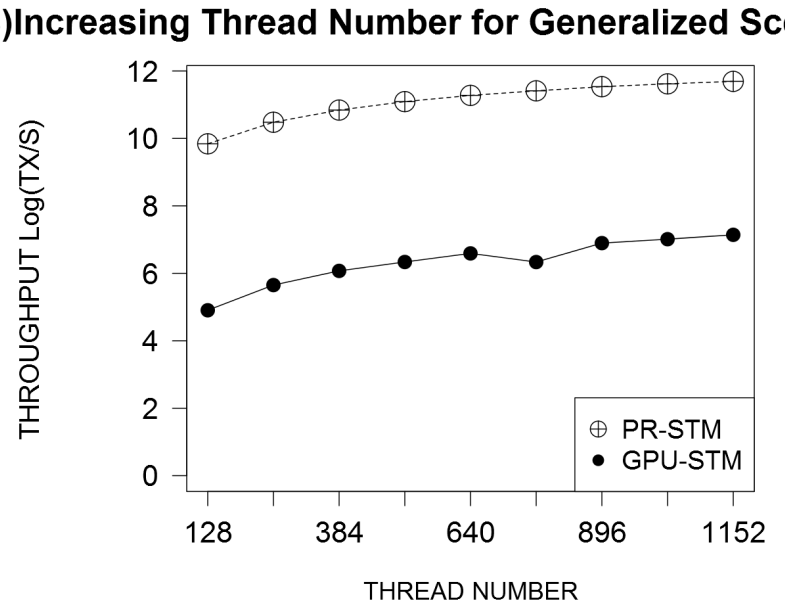


Figure 5.21: Generalized skiplist scenario average throughput with increasing number of threads and all read/write transactions. Graph A shows the value of transactions fulfilled in 1 second to observe the difference between throughputs, whereas graph B shows the Log of throughput to view the variation trend when thread numbers changes clearly.

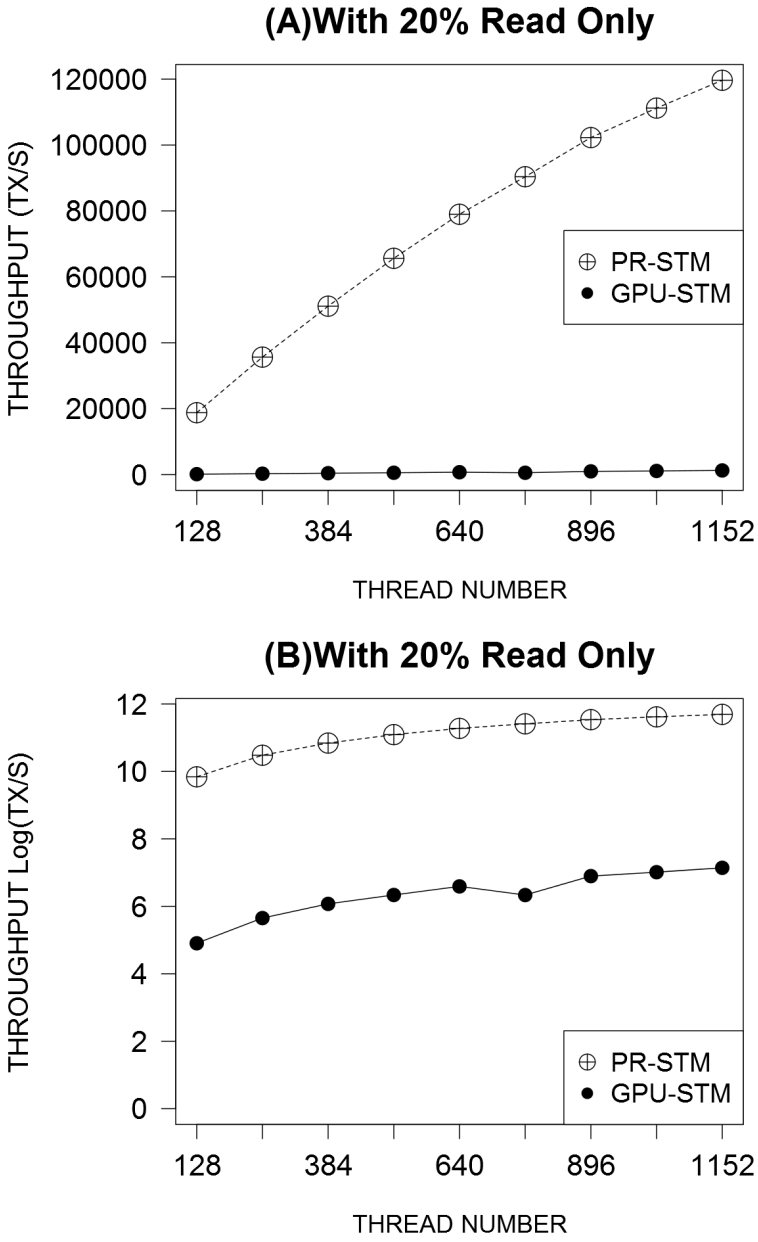


Figure 5.22: Generalized skiplist scenario average throughput with increasing number of threads and 20% read-only transactions. Graph A shows the value transactions fulfilled in 1 second to observe the difference between throughputs, whereas graph B shows the Log of throughput to view the variation trend when thread numbers changes clearly.

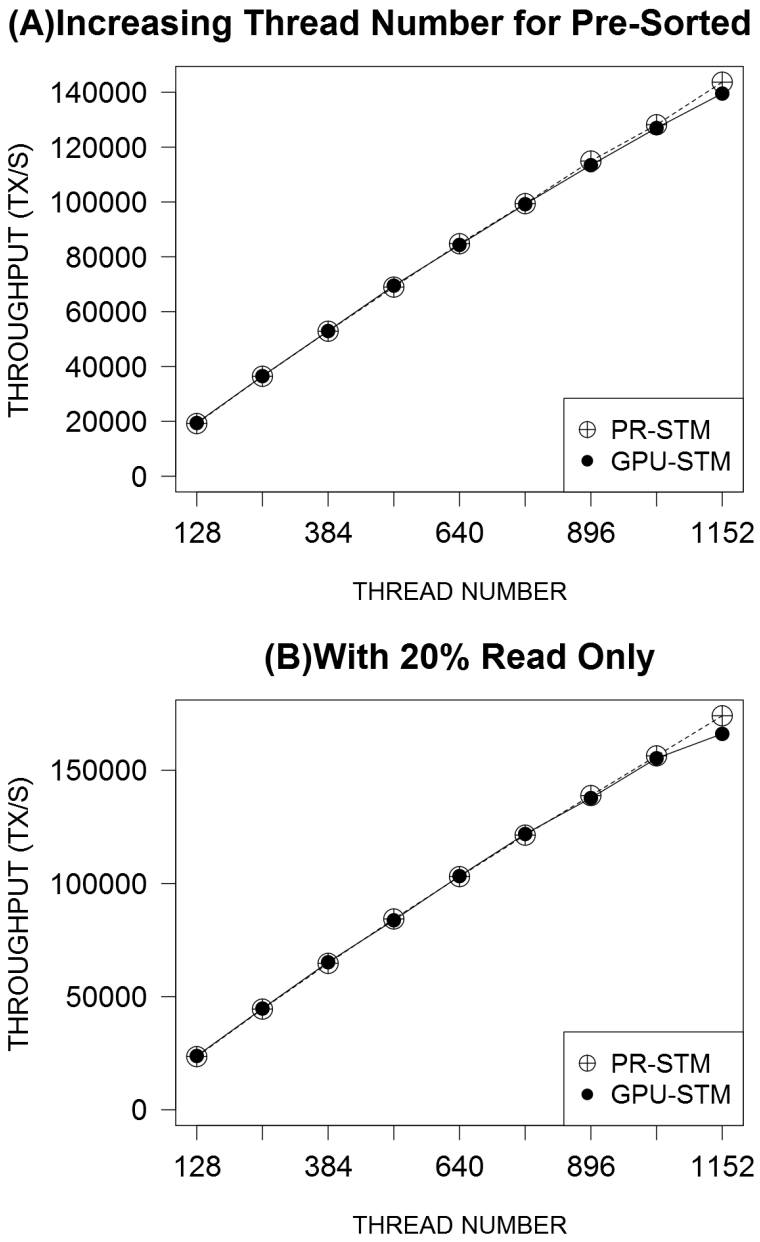


Figure 5.23: Pre-sorted skiplist scenario average throughput with increasing number of threads. The transaction tables are sorted by programmer with a certain order to avoid semantic conflict. Graphs A has all read/write transactions, graphs B has 20% read-only transactions.

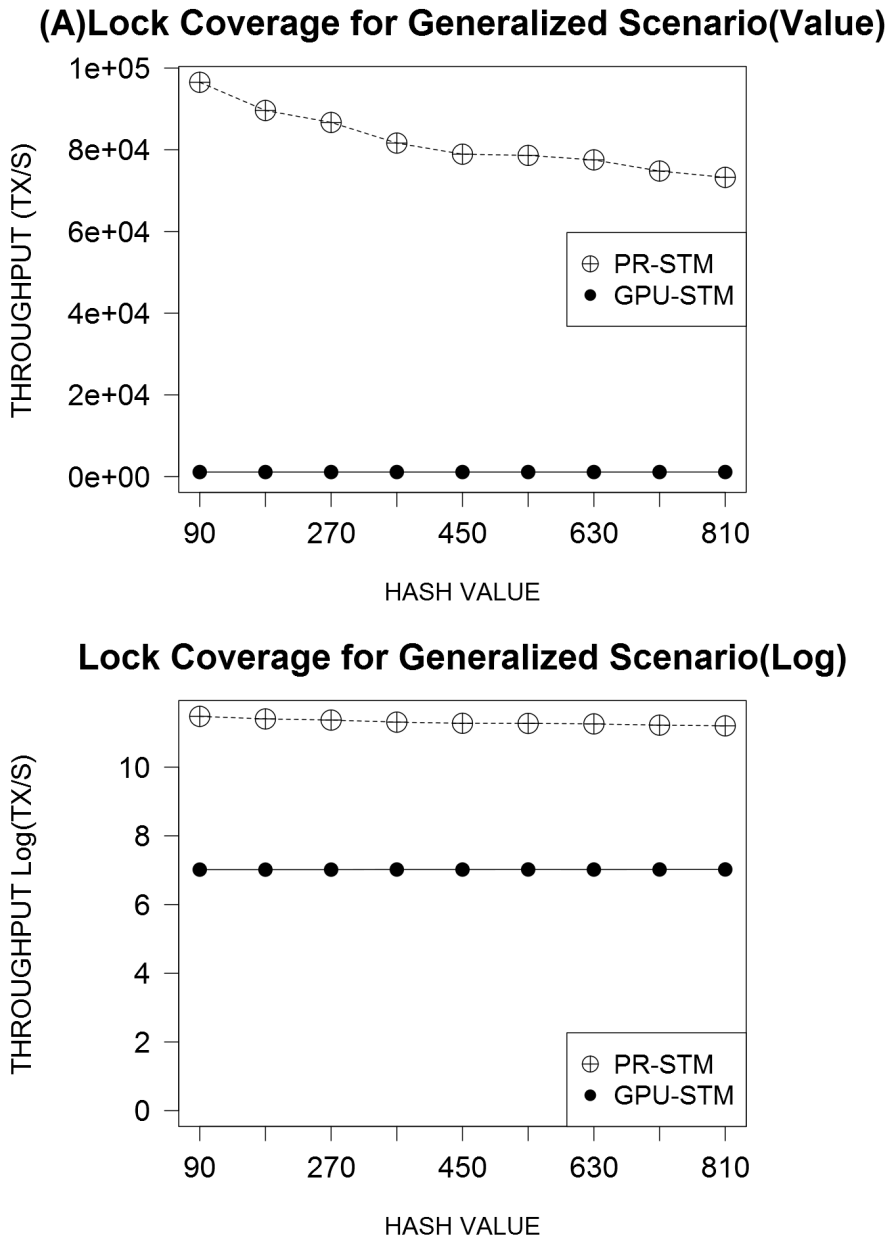


Figure 5.24: Generalized skiplist scenario average throughput with increasing hash number and all read/write transactions. Graph A shows the value of transactions fulfilled in 1 second to observe the difference between throughputs, whereas graph B shows the Log of throughput to view the variation trend when thread numbers changes clearly.

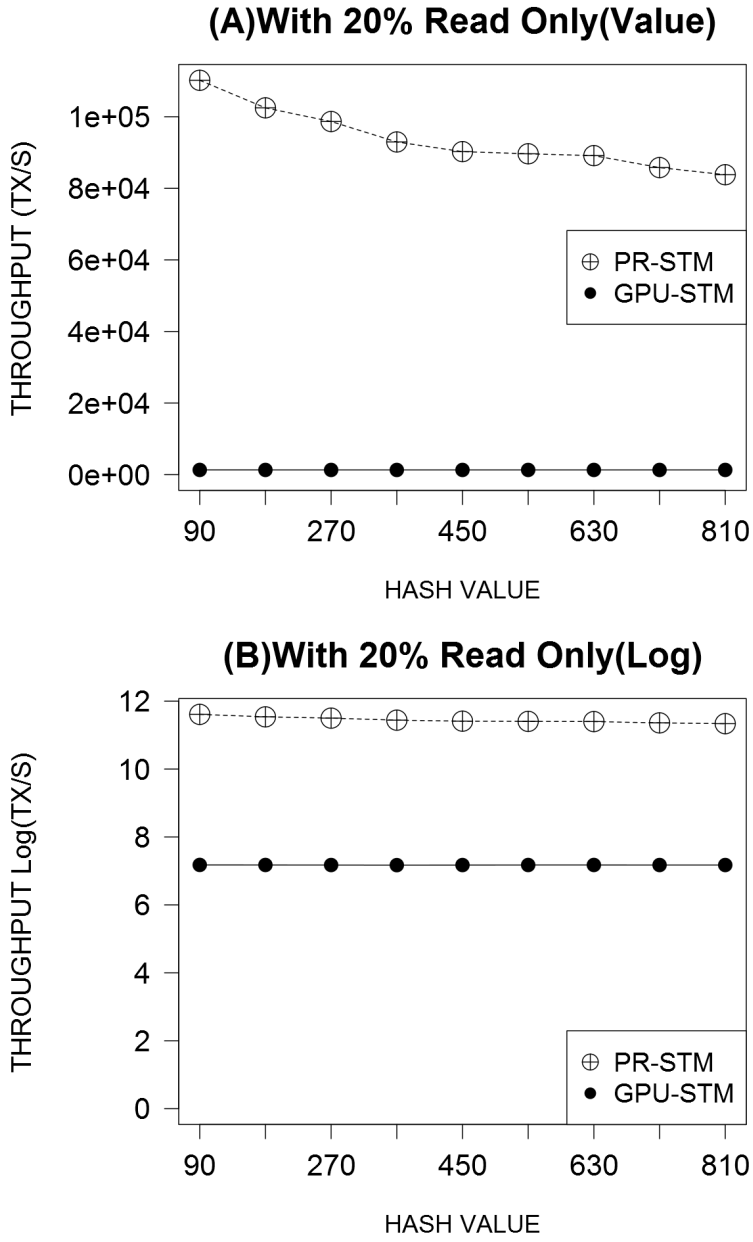


Figure 5.25: Generalized skiplist scenario average throughput with increasing hash number and 20% read only transactions. Graph A shows the value of transactions fulfilled in 1 second to observe the difference between throughputs, whereas graph B shows the Log of throughput to view the variation trend when thread numbers changes clearly.

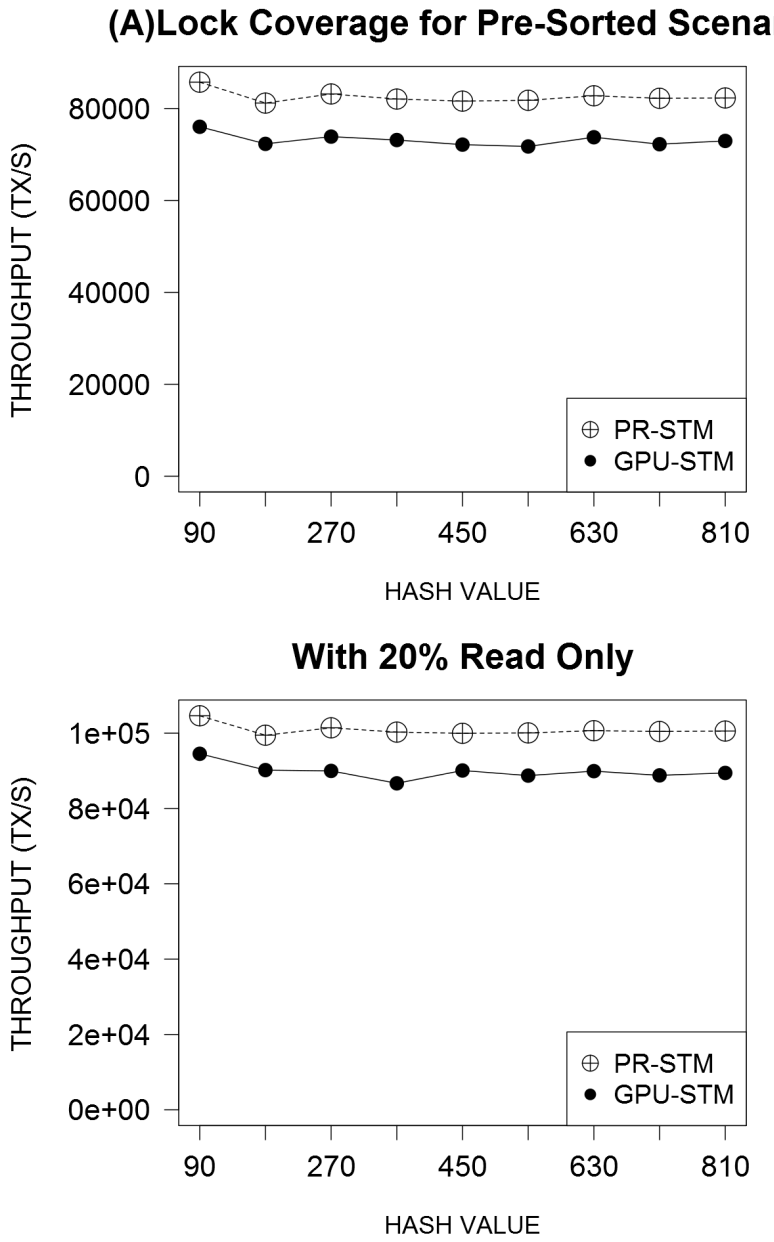


Figure 5.26: Pre-sorted skiplist scenario average throughput with increasing hash number. The transaction tables are sorted by programmer with a certain order to avoid semantic conflict. Graphs A has all read/write transactions, graphs B has 20% read-only transactions.

in the *skiplist* benchmark are shown in Figures 5.24, 5.25 and 5.26. In these experiemnts the thread number was set at 1024. Once again PR-STM is outperforming GPU-STM in all cases. The introduction of 20% read-only transactions also has a minimal improvement on throughput for both CMPs.

5.7 Summary

This chapter provides a set of experiments including both simple concurrent scenario and semantic awareness scenario. These experiments were performed for the sake of benchmark PR-STM which is proposed in this thesis compared to 'a state of art' software transactional memory GPU-STM. The results presented show that PR-STM outperforms GPU-STM in all cases. Of particular interest is the ability of PR-STM to address semantic conflict, which will result in failed progress of transactions with GPU-STM unless the application programmer has pre-ordered the transactions. To summarise, the results gathered from experiments in this chapter suggests:

1. Significant improvements in throughput when only consider conventional concurrent conflicts (race conditions).

2. When semantic conflict is introduced, the performance is admirable compared to existing GPU CMP. Furthermore, the correctness of PR-STM's result is guaranteed while the result by existing GPU CMPs may not be as expected.

3. This approach is generalised and fit different data structures. And programmers can relieve from sorting transaction order to avoid livelock caused by semantic conflict only with an acceptable overhead.

Chapter 6

Conclusion

In this chapter we summarise the material that has been presented in previous chapters, followed by a brief discussion about implications of our work, and end with some ideas for future work.

6.1 Thesis Summary

This thesis began with a description of a Software Transactional Memory system *PR-STM*. This system is presented as a optimistic approach to Concurrency Control based on a most fashionable parallel programming language for the GPU, CUDA. An implementation of this system was described by some pseudo code, with a Priority Rule based contention management policy. This contention manager is designed to adapt GPU architecture and provide concurrent conflict resolution. A comparison between this new system with previous GPU transactional memory and CPU transactional memory was presented and demonstrated the efficiency of our system. After that, the system is extended with a global transaction table and re-schedule semantic conflict aborted transactions to provide a conflict resolution for both concurrent conflict and semantic conflict which can make our system more generalised and suit all scenarios.

6.2 Main Contributions

The main contributions of this thesis can be described in three parts:

1. A transactional memory implementation on the GPU was presented in Chapter 3, which can take, as its batch input, arbitrarily ordered transactions and successfully execute them all without deadlock occurring using a priority system for determining contention resolution of locks. With the inherent priorities to sort threads order when manage conflicts during transaction time is optimal on GPU compare to previous work. Theoretically, that ordering execution is a scalable and lock-free solution for real-time concurrent problems.

2. A contention management system for the GPU that can handle semantic conflict at the application layer was provided in Chapter 4. Ordering of transactions can increase throughput and semantic correctness. For example, if two operations (deposit and withdraw)on a shared empty bank account are ordered withdraw first, then there would be a failure semantically. However, if the deposit was ordered before withdraw then this would be a correct semantic. Besides, as GPU threads are executing in a lock-step fashion, it may introduce the possibility of live-lock. A re-order of semantically aborted transaction can evade this. Our search is general purpose in nature as the re-order is independent of execution or data structure.

3. A comprehensive evaluative benchmarking of the previous two contributions demonstrating the improvement they provide in terms of throughput and efficiency compared to similar works in this area was illustrated in Chapter 5. These experiments were devised to show the effectiveness and generalization of proposed approach. The comparison was between our approach and a previous most efficient GPU transactional memory GPU-STM, varying degrees of both concurrent conflict and semantic conflict. Moreover, the scalability of systems is also taken into account with varying the memory usage and computation resource, and reveals a positive result.

6.3 Limitations

Although our solution can provide a software solution for concurrency control for the GPU, there are still some limitations or disadvantages:

1. Our solution has a better performance compared with current STM solutions on the GPU, but when comes to a specific scenario, it is still slower than dedicate and customized solutions. This is an overhead of generalization, and can be reduced by integrating more contention management policies into the system. If there are more contention management policies that specialized for some different scenarios, programmers can choose a most suitable one for their specific problem, and the performance can be better.

2. The memory space on the GPU is limited, this lead to a restriction of the space for global lock table. If the entire shared resource is quite huge, then there would only leave little space for a global lock table, which can result in a performance drop off. To achieve the best performance of this system, a large global lock table is suggested which needs enough memory space.

6.4 Future Work

The contribution of this thesis provides a novel software solution for concurrency control on the GPU. It opens a new door for future research as GPU parallel computing is no longer limited to nature parallel problems and inter-threads data communication is permitted. Besides, our approach to solve semantic conflict is just a initial step, semantic conflicts resolution remains a lot spaces to explore. In addition, CPU remains the superiority of individual thread computation speed and has advantage when conflict rate is high, so research about coordinate CPU and GPU for transactions can be meaningful. With respect to short-term developments, we suggest two approaches: (i)advanced semantic conflict prediction,(ii)cooperative transactions with both CPU and GPU. And here is an overview of each approach as a conclusion of this thesis.

Semantic Conflict Prediction Our approach in this thesis is providing a contention management and conflict resolution, which means only solve problems when concurrent or semantic conflict happens. An attractive alternative is to prevent conflicts from occurring by re-schedule transactions beforehand. More specifically, it could be like Shrink contention management which uses a probabilistic technique to order transactions a in a schedule that prevent conflicts, named Bloom Filters. And semantic conflict prediction can be more accurate for prediction as its deterministic nature. So a probably continue approach is to use a probabilistic structure such as Bloom Filter determines the likelihood of a semantic conflict, and if the possibility exceed a threshold then order this transaction to the same thread which carries all likely semantic conflict transactions concerning same shared data.

CPU and GPU cooperation To harness the maximal power of a computational node could perform, the cooperation between CPU and GPU is one of the obvious answers. CPUs have better clock speed than GPUs yet less of parallelism due to the emanation from overheating and energy-consuming. On the other hand, GPUs get clumsy when dealing with project that requires serial instruction flow with heavy workload where parallelism can not have its way. For the STM, the main conceptual idea behind it is the resolution to conflicting memory access. In contemporary computer production, CPU and GPU use separate memory space that is connected by PCIe.

However, this disadvantage could be taken as an advantage under certain circumstance. In a system that has both GPU and CPU run STM framework separately at the same time, the unpredictable overhead introduced by competing for mutual exclusive unit in either the lock-steal or the lock-table-sort system could be partially replaced by a controllable and predictable overhead that stems from the data transfer between CPU and GPU and other utilities needed to merge separate memory space together and, in the meanwhile, to maintain the program correctness. To succeed at such instance, the performance gain by CPU-GPU cooperation must be large enough to compensate for the overhead that comes from the communication between the CPU and GPU and the algorithm merging two memory spaces together.

One possible solution to it is to store the transaction table and its corresponding data set on both GPU and CPU RAM. The GPU starts executing transactions off first while CPU is listening from the GPU. When GPU STM has intense contention over a memory address, the GPU will out load some of the conflicting transactions to the CPU RAM via a buffer. Once CPU has received the buffer full of conflicting transactions, the CPU STM will execute the received transactions in a sequential manner so that a natural serialization of transactions competing for a certain range of memory addresses is kept. The performance gain in this solution rests on the idea that when intense contention happens, outsourcing some of the competing transactions are helpful to unblock the congestion and progress a warp control flow.

Another possible solution is to embrace the memory model current computer built upon, and recognize the fact that if GPU is executing transactions with a STM framework, CPU is not able to weigh in timely to work on the same memory space by synchronize two memory spaces in real-time due to the fact that overhead introduced by off-chip data transfer would be non-trivial. However, the CPU could be treated as a co-processor to help GPU with transaction serialization based on statistical data analysis to recognize the pattern of memory access and produce relevant parameters to adjust the behaviour of GPU STM. The GPU could keep a table that records the hit times of a memory address from its data set. If a memory address A is accessed more than a threshold that is calculated by the real-time CPU data analysis program, the GPU STM will establish a dedicated thread that only runs transactions requesting to access the memory address A. Therefore, a serialized execution model is incorporated. Because the nature of a STM scenario is to have several threads run transactions in arbitrary order, a non-unified memory space could lead to fatal failure for a transaction executed on CPU attempting to read an address resides on GPU RAM.

References

[1] Kshemkalyani Ajay D. and Singhal Mukesh. *Distributed Computing: Principles, Algorithms, and Systems*. Cambridge University Press, 2008. 18

[2] Mohammad Ansari, Mikel Luján, Christos Kotselidis, Kim Jarvis, Chris Kirkham, and Ian Watson. Steal-on-abort: Improving transactional memory performance through dynamic transaction reordering. In *High Performance Embedded Architectures and Compilers*, pages 4–18. Springer, 2009. 24

[3] Tongxin Bai, Xipeng Shen, Chengliang Zhang, William N Scherer, Chen Ding, and Michael L Scott. A key-based adaptive transactional memory executor. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–8. IEEE, 2007. 24

[4] Joao Barreto, Aleksandar Dragojevic, Paulo Ferreira, Ricardo Filipe, and Rachid Guerraoui. Unifying thread-level speculation and transactional memory. In *Proceedings of the 13th International Middleware Conference*, pages 187–207. Springer-Verlag New York, Inc., 2012. 25

[5] J. Bobba, K.E. Moore, H. Volos, L. Yen, and M.D Hill. Performance pathologies in hardware transactional memory. In *international symposium on Computer architecture*, pages 81–91. ACM, 2007. 21

[6] C. Boksenbaum, M.Cart, J. Ferrie, and J. Francois. Concurrency certifications by intervals of time-stamps in distributed database systems. *IEEE Transactions on Software Engineering*, pages 409–419, April 1987. 13

[7] Daniel Cederman, Philippas Tsigas, and Muhammad Tayyab Chaudhry. Towards a software transactional memory for graphics processors. In *Eurographics Symposium on Parallel Graphics and Visualization (EGPGV)*, pages 121–129, 2010. 41

[8] Phong Chuong, Faith Ellen, and Vijaya Ramachandran. A universal construction for wait-free transaction friendly data structures. In *Proceedings of the twenty-second annual ACM symposium on Parallelism in algorithms and architectures*, pages 335–344. ACM, 2010. 25

[9] Tyler Crain, Damien Imbs, and Michel Raynal. Towards a universal construction for transaction-based multiprocess programs. In *Distributed Computing and Networking*, pages 61–75. Springer, 2012. 25

[10] E. W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, **8**[9], 1965. 5

[11] Shlomi Dolev, Danny Hendler, and Adi Suissa. Car-stm: scheduling-based collision avoidance and resolution for software transactional memory. In *Proceedings of the twenty-seventh ACM symposium on Principles of distributed computing*, pages 125–134. ACM, 2008. 24

[12] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The notions of consistency and predicate locks in a database system. *Communications of the ACM*, **19(11)**:624–633, November 1976. 13

[13] M. J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, pages 948–960, 1972. 2

[14] Wilson WL Fung and Tor M Aamodt. Energy efficient gpu transactional memory via space-time optimizations. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 408–420. ACM, 2013. 45

[15] Wilson WL Fung, Inderpreet Singh, Andrew Brownsword, and Tor M Aamodt. Hardware transactional memory for gpu architectures.

In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 296–307. ACM, 2011. 45

[16] Weikum Gerhard and Vossen Gottfried. *Transactional Information Systems*. Elsevier, 2001. 16

[17] V. Gramoli. More than you ever wanted to know about synchronization: Synchrobench, measuring the impact of the synchronization on concurrent algorithms. *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 1–10, 2015. 32

[18] J. N. Gray, R. A. Lorie, G. R. Putzolu, and I. L. Traiger. Granularity of locks in a large shared data base. *Very Large Data Bases (VLDB)*, pages 428–451, 1975. 12

[19] J. N. Gray, R. A. Lorie, G. R. Putzolu, and I. L. Traiger. Granularity of locks and degrees of consistency in a shared data base. In G.M.Nijssen, editor, *Modeling in Data Base Management Systems*, pages 365–369. North-Holland, 1976. 13

[20] R. Guerraoui and M.Kapalka. On the correctness of transactional memory. pages 175–184, 2008. 42

[21] Rachid Guerraoui, Maurice Herlihy, and Bastian Pochon. Toward a theory of transactional contention managers. In *Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing*, pages 258–264. ACM, 2005. 24

[22] T. Haerder and A. Reuter. Principles of transaction-oriented database recovery. In *ACM Computing Surveys*. ACM, 1983. 5, 19

[23] T. Harris and K. Fraser. Language support for lightweight transactions. In *ACM SIGPLAN conference on Systems, Programming, Languages and Applications*. ACM, oct 2003. 36, 42

[24] Tim Harris, James Larus, and Ravi Rajwar. *Transactional Memory, 2nd edition*. Synthesis Lectures on Computer Architecture, 2010. 19

[25] Tomer Heber, Danny Hendler, and Adi Suissa. On the impact of serializing contention management on stm performance. *Journal of Parallel and Distributed Computing*, **72**[6]:739–750, 2012. 24

[26] M. Herlihy, V. Luchangco, and M. Moir. A flexible framework for implementing software transactional memory. In *ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 253–262. ACM, 2006. 22

[27] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer III. Software transactional memory for dynamic-sized data structures. In *ACM Symposium on Principles of Distributed Computing*. ACM, jul 2003. 39

[28] M. P. Herlihy. Impossibility and universality results for wait-free synchronization. *7th Annual ACM Symp. on Principles of Distributed Computing*, pages 276–290, 1988. 12

[29] M. P. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: Double-ended queues as an example. *23rd International Conference on Distributed Computing Systems*, pages 522–530, 2003. 12

[30] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, **12(3)**, 1990. 12

[31] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, **13**[1]:124–149, 1991. 25

[32] M.P. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *International Symposium on Computer Architecture*, pages 289–300. ACM, 1993. 35

[33] M.P. Herlihy and J.M. Wing. Axioms for concurrent objects. *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 1987. 32

[34] Anup Holey and Antonia Zhai. Lightweight software transactions on gpus. In *Parallel Processing (ICPP), 2014 43rd International Conference on*, pages 461–470. IEEE, 2014. 44, 45

[35] H. T. Kung and T. Robinson John. On optimistic methods for concurrency control. *ACM Transactions on Database Systems*, pages 213–226, 1981. 4, 13

[36] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, pages 690–691, 1979. 3

[37] Nakul Manchanda and Karan Anand. *Non-Uniform Memory Access (NUMA)*. New York University, 2010. 27

[38] R. Michel. Lock-based concurrent objects. In *Concurrent Programming: Algorithms, Principles, and Foundations*, pages 63–75. Springer-Verlag, 2012. 15

[39] Gordon E Moore. Cramming more components onto integrated circuits. In *Electronics*, **38**, pages 56–59. IEEE, 1965. 26

[40] Rupesh Nasre, Martin Burtscher, and Keshav Pingali. Atomic-free irregular computations on gpus. In *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*, pages 96–107. ACM, 2013. 45

[41] Rupesh Nasre, Martin Burtscher, and Keshav Pingali. Morph algorithms on gpus. In *ACM SIGPLAN Notices*, **48**, pages 147–156. ACM, 2013. 45

[42] C. Nvida. Compute unified device architecture programming guide. Nvida, 2007. 8, 28, 33

[43] C. Nvida. Programming guide. Nvida, 2008. 29

[44] Bernstein Philip, a., Hadzilacos Vassos, and Goodman Nathan. *Concurrency Control and Recovery in Database Systems*. Addison Wesley Publishing Company, 1987. 15

[45] T Riegel, P Felber, and C Fetzer. *TinySTM*. 2010. 22

[46] William N Scherer III and Michael L Scott. Advanced contention management for dynamic software transactional memory. In *Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing*, pages 240–248. ACM, 2005. 24

[47] Craig Sharp, William Blewitt, and Graham Morgan. Resolving semantic conflicts in word based software transactional memory. In *Euro-Par 2014 Parallel Processing*, pages 463–474. Springer, 2014. 25

[48] Craig Sharp and Graham Morgan. Hugh: a semantically aware universal construction for transactional memory systems. In *Euro-Par 2013 Parallel Processing*, pages 470–481. Springer, 2013. 25

[49] N. Shavit and D. Touitou. Software transactional memory. In *ACM Symposium on Principles of Distributed Computing*. ACM, aug 1995. 35

[50] A. Silberschatz and P. B. Galvin. Cpu scheduling. In *Operating System Concepts 4th Edition*, pages 176–179. John Wiley and Sons, 1994. 14

[51] Arons T. Using timestamping and history variables to verify sequential consistency. *13th Conference on Computer Aided Verification*, 2001. 17

[52] Andrew Tanenbaum. *Modern Operating Systems*. Prentice Hall, 2001. 20

[53] Jons-Tobias Wamhoff and Christof Fetzer. The universal transactional memory construction. Technical report, Tech Report, 12 pages, University of Dresden (Germany), 2010. 25

[54] Yunlong Xu, Wang Rui, Luan Zhongzhi, Lan Gao, Wu Weiguo, and Depei Qian. Lock-based synchronization for gpu architectures. In *Proceedings of the ACM International Conference on Computing Frontiers*, pages 205–213. ACM, 2016. 42

[55] Yunlong Xu, Rui Wang, Nilanjan Goswami, Tao Li, Lan Gao, and Depei Qian. Software transactional memory for gpu architectures. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, page 1. ACM, 2014. 7, 33, 42