TÉCNICO LISBOA

Hardware Accelerated Cloud Data Store for Data Mining over Big Spatio-Temporal Data

Ricardo Manuel Nunes Vieira

Relatório da disciplina de Introdução à Investigação e ao Projecto em Engenharia Electrotécnica e de Computadores

Engenharia Electrotécnica e de Computadores

Orientadores: Doutor Paolo Romano
Doutor Aleksandar Ilic

Júri

Orientador: Doutor Paolo Romano
Vogal: Doutor João Lourenço

Janeiro de 2016

Abstract

Over the course of the last decade, concerns with power consumption and demand for ever increasing processing power have lead micro-architectures to a highly parallel paradigm, with multiple cores available in each processing units. Notably the Graphics Processing Unit (GPU), which was previously a very rigid highly pipelined unit, now uses a more versatile manycore architecture. These architectural changes, along with their raw computation power, led researchers to useGPUs for general purpose computing, working similarly to multicore Central Processing Unit (CPU).

Whilst these parallel architectures allow for great peak performances, achieving those values imposes complex programming hurdles, especially in regards to synchronizing accesses to shared data. Applications that extract the inherent parallelism of heterogeneous systems equipped with both GPU and CPUs are particularly hard to develop, due to the nature of the limited communication between these units.

In this context, Transactional Memory (TM) has been proposed to facilitate concurrent programming. TM is an abstraction that moves the complexity of synchronizing shared data access in multi-threaded systems away from the programmer. Building on this basis, a new type of TM is proposed: a heterogeneous TM, with the goal of scheduling threads from the same application across CPUs and GPUs.

As a first step toward this goal, this report introduces the related work, as well as some preliminary ideas and results for this thesis.

Keywords

Transactional Memory; Graphical Processor Units; General Purpose Computing on GPUs; Heterogeneous System;

Resumo

Ao longo da última década a preocupação com o consumo energético, bem como a procura por maior poder de processamento levou as microarquitecturas a um paradigma altamente paralelo, com vários cores disponíveis em cada unidade. Especial atenção a Graphics Processing Unit (GPU), que de unidades altamente pipelined e pouco versáteis passaram a uma muito mais versátil arquitetura many-core. Estas mudanças na arquitetura, bem como a capacidade de computação, levou investigadores a usar GPUs para processamento de uso geral, funcionando de um modo semelhante a um Central Processing Unit (CPU) multicore.

Apesar destas arquiteturas paralelas permitirem, teoricamente, uma performance bastante alta, atingir esse ponto impõe desafios complexos de programação. Aplicações que extraem o paralelismo inerente entre GPU e acfcpu são particularmente difíceis de desenvolver, principalmente devido á natureza limitada das comunicações entre estas duas unidades.

Neste contexto, Memória Transacional (Transactional Memory (TM)) foi proposta para facilitar programação paralela. Transactional Memory (TM) é uma abstração, que procura esconder do programador a complexidade de sincronismo de dados partilhados em sistemas com múltiplas threads. Com base nesta abstração, este trabalho propõe a criação de um novo tipo de TM, um sistema heterógeno capaz de escalonar threads da mesma aplicação a tanto CPU como GPU.

Como um primeiro passo nesta área, este relatório apresenta o trabalho relacionado, bem como algumas ideais e resultados preliminares para esta tese.

Palavras Chave

Memória Transacional; Unidades de Processamento de Gráficos; Computação de Propósito Geral em GPUs; Sistemas Heterogéneos;

Contents

List of Figures

List of Tables

List of Acronyms

API	Application Programming Interface
TM	Transactional Memory
STM	Software Transactional Memory
HTM	Hardware Transactional Memory
HyTM	Hybrid Transactional Memory
TSX	Intel's Transactional Synchronization Extensions
HLE	Hardware Lock Elision
RTM	Restricted Transactional Memory
GPU	Graphics Processing Unit
GPGPU	General Purpose Computation on Graphics Processing Units
CPU	Central Processing Unit
ALU	Arithmetic Logic Unit
SFU	Special Function Unit
ISA	Instruction Set Architecture
CUDA	Compute Unified Device Architecture
SM	Stream Multiprocessor
SIMD	Single Instruction Mutiple Data
SIMT	Single Instruction Mutiple Thread
STAMP	Stanford Transactional Applications for Multi-Processing
TPC	Texture/Processor Cluster

1

Introduction

With years of improvement, single-core performance of current Central Processing Unit (CPU) has reached a plateau, making further improvements very difficult to achieve. Hence, the microprocessor industry moved to multicore architectures as a way to increase raw instruction throughput and allowing greater performances. Nowadays multicore and manycore architectures are a staple in everything from research to consumer grade products.

These new highly parallel architectures have a high potential in terms of performance, theoretically. However they pose new challenges, particularly in regards to synchronizing shared data access. As these are non trivial, even for experienced coders, research exists on how to use software/hardware libraries to abstract some of the difficulties in developing concurrent applications for these systems. One such example is Transactional Memory (TM) [6], which facilitates shared memory access in multi-threaded systems.

Multiple TM implementations exist for CPUs, but research in this area regarding Graphics Processing Units (GPUs) is scarce, in spite of the emergence of General Purpose Computation on Graphics Processing Units (GPGPU). This field focuses on using GPUs for tasks other than graphics processing. GPUs started as dedicated coprocessors, with highly rigid and pipelined architectures, but have moved to a more versatile manycore architecture, which allows for a programming model that is closer to that of multicore CPUs. Since GPUs are nearly ubiquitous in consumer oriented products, and their manycore architecture provides a much greater peak performance than the one on CPUs, fully exploiting their potential processing power is a hot research topic.

1.1 Motivation

As mentioned before, the trend towards parallel computing brings the cost of an increased complexity in programming, due to the need for synchronization between multiple threads or machines. Traditionally, for multi-threaded software, locks have been the predominant mechanism used to synchronize accesses to shared memory. The usage of locks is notoriously challenging, as the programmer needs to keep in mind situations like deadlocks and livelocks, which can be very difficult to avoid for complex software with unpredictable memory access patterns.

On GPUs, the Single Instruction Mutiple Thread (SIMT) architecture exacerbates this problem by increasing the number of ways in which a deadlock/livelock can manifest. Furthermore the usage of coarse-grained locks is particularly undesirable on GPU architectures as any serialization has a huge impact on performance. On the other hand, the usage of fine-grained locks for thousands of threads (which is a common situation on GPUs) requires facing the already mentioned programming challenges.

Synchronization between CPU and GPU is challenging to implement, as their nature as physically separated devices - in most cases - makes communication between these units costly in terms of efficiency. Most applications choose to focus on either one or the other, although some rare applications [7] already make use of tasks partitioning amongst both units.

The focus of this work is on TM, an abstraction that facilitates concurrent programming by allowing complex synchronization code to be abstracted away, to either a software library or a specialized hardware unit. TM solutions, for both CPU and GPU, already exist. However these solutions do not support sharing of data among threads running on processing units of different types. Thus, they fail to fully explore the computing capability of systems containing multiple units.

Building on what TM does for shared data amongst various threads of the same unit, a new solution is proposed: an heterogeneous TM. The goal is to abstract sharing data amongst multiple units, transparently and correctly. A library with these characteristics allows for a better hardware occupation as opposed to, for example, a CPU solution that leaves the GPU idle. As GPUs are present in nearly all computing devices, such a solution could potentially have a large scope of applications. On the other hand this is, to the best of our knowledge, a still unexplored approach in the literature, which makes it interesting from a research perspective.

1.2 Objectives

The objective for this thesis is to develop the already mentioned Heterogeneous Transaction Manager. The intended goal is that applications running this system are able to share computational power, transparently, between the CPU and the GPU, through the running of concurrent transactions.

The biggest challenges in developing this system lie in identifying which CPU and GPU TM can maintain efficiency in a high latency system, and developing a synchronization scheme which accounts for this latency.

Since explicit synchronization is not feasible for high latency situations, the proposed solution is to speculatively schedule transactions to both units. After execution concludes, or periodically, the pro-

duced results can be checked guarantee correctness.

Developing a scheduler with these capabilities is the core of this work, and presents probably the biggest challenge. Partitioning appears to be the most promising approach, however development of this solution will be highly iterative with the intention of finding the most optimal solution.

If scheduling is speculative, which it is likely to be, then another challenge is introduced in the form of conflict resolution. Whilst it is easy to implement policies for these tasks, maintaining efficiency requires extensive testing.

Taking this into account, the high level goals for this thesis are:

1. Research TM implementations for both CPU and GPU.
2. Develop a TM scheduler capable of partitioning transactional workloads, with a high degree of success.
3. Research a conflict detection and resolution policy for conflicts between units.

The benchmarks used are likely to be part of the STAMP suite [8], as they are the *de-facto* benchmarks for TM systems. Micro-benchmarks, such TinySTM's [9] Bank scenario, will also be used, especially during development.

1.3 Outline

This report is structured as follows: Chapter 2 analyses current research on TM and GPGPU. Chapter 3 presents the work plan for this thesis, as well as some preliminary results. Finally Chapter 4 presents some concluding remarks.

2

State of the Art

The switch to multi/manycore processors is a milestone in the history of computing. Whilst these architectures provide amazing potential for performance, they also come at the cost of an increased programming challenge. The biggest difficulty is ensuring synchronism amongst shared data, as in a situation where multiple threads are running, accesses can be made in any number ways, concurrent access can lead different threads to observe or produce inconsistent results.

Locks have been the predominant mechanism used to synchronize shared data. Coarse-grained locks provide bad performance but are easy to implement. On the other hand, fine-grained locks provide the best possible performance, but they are difficult to implement correctly, requiring extensive testing.

TM has been proposed as an alternative to locks [6]. The base premise of this concurrency control mechanism is that accesses to the shared memory can be grouped into a transaction, which is executed as if it was a single atomic operation. The objective is to hide the complexity of synchronization in a TM library that implements this simple abstraction of atomic blocks. As such, the programmer defines his intent - i.e., marking the blocks of code that need to be synchronized, much like marking critical sections of code - instead of having to define the implementation of the synchronization as with traditional locking. Implementations have been developed in both software and hardware, and even as hybrid solutions. TM has recently gained relevance with hardware vendors, such as Intel and IBM, who have included support for Hardware Transactional Memory (HTM). For instance Intel's Haswell, a consumer grade line of processors, includes a Restricted Transactional Memory (RTM) implementation [10].

Since the goal of this work is to develop a heterogeneous transaction manager, which schedules transactions across both CPU and GPU, the next sections survey the state of the art for TM in both areas, to determine the best individual approaches. As such the remaining of this chapter is organized as follows: first an overview of the different possibilities fot software, hardware and hybrid TM implementations is provided, followed by an analysis of the benchmarks used to evaluate them. Afterwards, a

special emphasis will be put into analysing general purpose computing on GPUs, by looking at the basic GPU architecture and the Compute Unified Device Architecture (CUDA) platform (a parallel computing platform and programming model for Nvidia GPUs). Finally, state of the art TM implementations on GPU are also studied.

2.1 Transactional Memory

In TM contexts, a transaction is defined as a set of operations that are executed in order and satisfy two properties: atomicity - i.e., if one of these operations fail, all others follow suit - and serializability - i.e., all transactions appear to execute serially [6]. Transactions are traditionally used in database systems to guarantee both isolation between concurrent threads and recovery mechanisms.

Inspired by this traditional and familiar concept of transactions, the abstraction of TM was proposed to synchronize concurrent accesses to shared data held in-memory. The first implementation devised was built on top of the cache coherency of multicore processors [6]. Thenceforth, many implementations of the TM abstraction have followed.

For instance, in a hash table implementation, where elements are constantly inserted and removed, the programmer would need to either protect all accesses using a single lock - serializing all operations even if they all targeted different entries in the table, or she could create locks for every base pointer - serializing accesses to the most popular accounts. Furthermore, even if fine grained locking is used, a insertion can be, for instance, dependant on multiple removals, prompting threads to acquire multiple locks, which increases the probability of deadlocks/livelocks.

When using TM the programmer marks a set of instructions with an atomic construct or a transaction start/end marker. In the previously mentioned case it would simply require marking a set of has hash table operations that need to be executed as if they were atomic. It is then up to the underlying TM system to ensure correct synchronization among concurrent threads.

When a transaction executes the marked instructions it generates both a read set (list of addresses it read data from) and a write set (list of addresses it has written data to). While the transaction is running, all writes are kept private - by being buffered in most implementations - and no changes to the memory are seen by the rest of the system, to guarantee isolation and to avoid inconsistent states.

When all operations are concluded, most TM implementations validate the read and write sets, which can lead to either a commit - saving the write set to the main memory - or an abort - reverting all the changes made by the transaction. Validation exists to guarantee correctness between multiple transactions, as they may be running at the same time and modifying the same addresses, causing what is referred to as a conflict.

It should be noted that not all conflicts lead to aborts, as there may be systems in place to resolve them, such as contention managers, which are investigated further ahead.

Since multiple transactions can be running at same time a correctness criterion is usually necessary. Researchers often use serializability [11], i.e., having an equivalent outcome with the parallel execution as one that could be obtained with a serial execution. This approach focuses only on the commit results

and ignores what happens to live and aborted transactions. As a result researchers have proposed Opacity [12] as a correctness criterion. Opacity states that all live transaction must see consistent memory states, and no modification made by aborted transactions can be visible to other transactions.

As previously mentioned, the abstraction of TM allows for several types of implementations to exist, making the choice of the system to use very important. Usually these implementations all have the same guarantees and ensure the same degree of correctness, however they may be better suited to certain workloads or programs. This variety of implementations makes classifying TM all the more important. TM are usually classified with respect to the following aspects:

- Granularity: The level at which conflicts are detected, or how much detail is saved in the read/write sets. Systems can range from Word-based (4 or 8 bytes depending on the processor) to the object level.

- Read Visibility: Visible reads imply keeping some sort of publicly accessible metadata to inform all transactions of reads occurring in other transactions. This allows for earlier conflict detection but also increases memory traffic. Invisible reads force each thread to verify both its read and write set, and lead to late conflict detection.

- Conflict Detection: Eager systems detect conflicts as soon as possible while Lazy systems leave this to the end of the transaction. Eager TM are favourable for long transactions, as the cost of restarting a long transaction at the commit phase is much greater the smaller ones.

- How conflicts are detected. Value-based systems record the actual read values in the read set (in addition to their address), and use them in the validation phase to check if the transaction contains outdated data, and thus must abort. Timestamp-based systems check if the read memory positions have a higher timestamp than the one of the transaction trying to lock them. Every time a transaction commits, the memory positions it wrote to are given the current timestamp. Lock-based systems avoid a validation phase by locking the memory positions accessed by the transaction, and preventing other transactions from accessing them.

2.1.1 Implementations of Transactional Memory

TM was first proposed by Herlihy et al. [6] as an architectural solution for data sharing in multicore CPU architectures. While initially proposed as a Hardware based solution, TM has seen implementations in Software and in combination of both - named Hybrid Transactional Memory (HyTM).

Software Transactional Memory

The cost of manufacturing processors, in hardware, coupled with the difficulty of testing hardware based solutions, has led researchers to explore software based solutions implementing TM. As a consequence of software's inherent flexibility, many different designs of Software Transactional Memory (STM) were developed, as hinted above by the list of characteristics that are possible to vary.

One such example is granularity. In HTM, granularity is dependant on the cache size, but STM does not incur the limitations of hardware, and may have different granularities. To guarantee Opacity, most STM implementations's writes are made to buffers - called deffered writes - and are only saved to the main memory at commit time, although some write directly to memory and use an undo-log to restore a consistent state in case of transactions' aborts.

Software implementations usually have a higher instrumentation overhead (overhead caused by the mechanisms used to ensure correct synchronization), but they are capable of running much larger transactions and can use schedulers and contention managers to avoid/solve conflicts. Table 2.1 shows a comparison between 5 state-of-the art STM implementations: NORec[13], TL2[14], SwissTM[15], TinySTM[9] and JVSTM[16], with regards to some key parameters for STM.

The most common synchronization strategy for STM is time-based, as it seems to be the best performing in state of the art implementations. Time-based STMs use a global-clock, that is incremented with each commit and saved locally at the start of each transactions. This clock serves to establish a snapshot that defines what transactions can access. If a transactions issues a read that requires a more recent snapshot then a conflict has happened, and needs to be resolved. Using this system no locking is necessary for read-only transactions as, if validation of the read set succeeds - meaning that every value has the same snapshot value as the thread's - there is no need to lock anything.

NORec [13], meaning no ownership records, is an example of a time-based STM. It was designed with the intent of having as little memory overhead as possible, and uses a single global lock for commits, which is merged with the global clock. This comes at the cost of scalability, as the validation overhead increases greatly with the number of threads. To avoid memory overhead, NORec uses value based validation, which is triggered on every global clock increment and before its commits. This forces NORec to revalidate its read set. In case of success the transaction updates its snapshot to match the current one and continues execution, otherwise it aborts. NORec uses deffered writes to facilitate aborts.

Since NORec was designed for a low number of threads its performance doesn't scale well in high concurrency scenarios, as the existence of a global lock serializes commits. TL2 on the other hand, uses fine-grained locks to achieve better performance in higher concurrency scenarios.

TL2 [14] is a word/object based STM. It was one of the first STM to use the concept of time-based snapshotting which was presented earlier. TL2 uses timestamp based validation, meaning that it saves the timestamps of each memory position accessed by a transaction, and uses them to compare with the thread's snapshot. For writes, it uses commit time locking, meaning that it only locks the access to positions it is writing too when committing. It also uses the commit process to change the version value in the metadata for each updated word/object.

Whist the two previously analysed systems are focused on optimizing for different levels concurrency, SwissTm [15] was developed with throughput in real world scenarios in mind. It was designed to perform best in mixed workloads, i.e., both large and small transactions, and is lock and word-based. What distinguishes it is the use of mixed conflict detection strategies.

SwissTM eagerly detects write conflicts, but lazily detects read conflicts. It does this since write after write conflicts (when the positon the transaction intends to write is written to) guarantee aborts,

while write after read conflicts (when a positon the transaction has read is written to) may not lead to an abort. This approach aims to increase parallelism by having some conflicts solved at commit time, whilst still preventing transactions that are ”doomed” to fail from wasting resources. SwissTM also uses a Contention Manager which transactions prompt on a conflict detection, so that it decides the optimal outcome. Contention Managers goals (resolving conflicts) are lazier version of a scheduler's (preventing conflicts), which are analysed further ahead.

TinySTM [9] is a popular alternative to SwissTM. It is a very efficient word-based and lock-based STM. It was designed to be lightweight. However it allows for extensive configuration and has great performance which makes it a baseline to which most current research is compared against. It is also open-source, allowing for adaptations for specific workloads. For these reasons it is likely to be the implementation of choice for the work herein proposed.

It uses - in its recommended configuration - encounter time locking for writes, meaning that when a transaction needs to perform a write to a certain memory position, it needs to acquire its lock first (unless it already has it), preventing other writes to that location. Locks are then released either on commit time or on abort. Some other parameters like the number of locks, can be individually tuned, or left to the system which performs automatic tuning.

This STM provides both in place and deferred writes, however the latter forces the use of Eager conflict detection, whilst the former allows for a choice between Eager and Lazy. For both modes, reads are invisible and are done by checking the lock and the version, reading the value and then checking the lock/version once more, to guarantee it wasn't either locked or updated.

Although TinySTM and SwissTM perform efficiently in TM benchmarks, many real workloads tend to be much more read-oriented with large reader transactions and small rare update transactions, than the workloads present in typical TM benchmarks.

As a result, this led to the design of JVSTM [17]. JVSTM is a object-based STM, using Multi-Version Concurrency Control, a mechanism that stores the history of values for each variable, along with a version value. When starting a transaction, the current version number is saved, and all reads target that version number. This allows read-only transactions to read all values corresponding to its version number (or the most recent one that's older then the transaction's), and then commit without any validation as the memory state it observed was always consistent. Transactions with writes need to first validate their read-set as being the most recent one and only then acquire a global-lock to commit.

Newer versions of JVSTM [16] are actually lock-free. Committing threads are put in a queue, where they wait before their turn to commit, however, unlike spinning for a lock, JVSTM's waiting threads assist the ones before them with their commit operation. This way, even if a thread that acquires the global lock gets de-scheduled by the processor, others can complete its transactions for it, without having to wait for that thread to continue execution.

Hardware Transactional Memory

Herlihy et al. [6] first proposed HTM as an additional cache unit, a transactional cache. This cache had an architecture similar to a victim cache [18]. Whilst running, transactions write to this cache. Since

	Granularity	Conflict Detection	Writes	Synch Strategy
NoRec [13]	Word	Lazy	Deferred	Lock-Based
TL2 [14]	Word/Object	Lazy	Deferred	Lock-Based
SwissTM [15]	Word	Mixed	Deferred	Lock-Based
TinySTM [9]	Word	Mixed, Eager, Lazy	Both	Lock-Based
JVSTM [16]	Object	Lazy	Deferred	Lock-Free

Table 2.1: Comparison between the most popular state of the are STM implementations.

the transactional cache is not coherent between multiple processors, transactions are isolated amongst the various processors. When validation succeeds the results are copied to the next level cache and propagated to all processors. On the other hand, in case of an abort, the cache can simply drop all the entries updated by the aborting transaction, ensuring both atomicity and isolation.

Validation is done by modifying cache coherency protocols [19], since accessibility control is pretty similar to transactional conflict detection. The modified protocol is based on using 3 different types of access rights, which are already present in most cache coherency protocols: in memory and thus not immediately available, shared access (allows reads) and exclusive access (allows writes). Acquiring exclusive access leads to blocking other threads from getting any access rights to that position, whilst shared access can be available to multiple processors at the same time. These access rights have equivalents in transactional memory and thus serve as validation, if a transaction successfully accesses all its read and write set then it can commit safely.

Benchmarks [6] showed that this implementation was competitive with lock-based solutions in most situations, even outperforming them in some. However the usage of a small cache limits transactions to a small set of reads/writes, which may not reflect real workloads. While increasing the cache size seems like an obvious solution, it comes with an increase in cost and energy consumption. It might also force architectural changes due to how TM's cache size and the processors cache size are related, as, for instance, we can never have a commit that overflows the cache's size.

As mentioned before Intel's Haswell and Broadwell lines of processors implement support for HTM using Intel's Transactional Synchronization Extensions (TSX). Two software interfaces exist: Hardware Lock Elision (HLE), a backwards compatible implementation which allows optimistic execution of code sections by eliding writes to a lock, and RTM, which is similar to HLE but also allows programmers to specify a fallback for failed transactions, whilst HLE always falls back to a global lock. The latter implementation gives full control to the software library, allowing for smarter decision [20]. This favours the usage of Hybrid TM, which will be analysed latter. It is important to know that TSX, like all HTM implementations, provides only a best-effort policy, so transactions are not guaranteed to commit and a policy for dealing with aborts must be defined [5].

Whilst TSX's interface to programmers is well defined, the actual hardware implementation is not documented. TSX's documentation specifies validation - which is done by the cache coherency protocol - and that its read and sets granularity is at the level of a cache line. Independent studies [21], point to the use of the L1 cache to support transactional operations, which in addition to the data gathered from TSX's documentation points to a very similar implementation as the one proposed by Herlihy et al.[6]. Just like the work presented before, transactions may fail to commit by overflowing the cache, making

Table 2.2: HTM implementations of commodity processors. Adapted from [5]

Processor	Blue Gene/Q	zEC12	Intel Core i7-4770	POWER8
Transactional-load Capacity	20 MB (1.25 MB per core)	1 MB	4 MB	8 KB
Transactional-Store Capacity	20 MB (1.25 MB per core)	8 KB	22 KB	8 KB
Conflict-detection Granularity	8 - 128 bytes	256 bytes	64 bytes	128 bytes
Abort Codes	-	14	6	11

TSX not suitable for long transactions.

Benchmarks show that TSX is very effective for small transactions [22], but results are highly dependant on tuning the TM to work correctly, especially in regards to the retry policy, with settings such as the number of retries on abort and reaction to aborts influencing the results heavily. Research has been done into both finding the correct values for multiple types of applications, as well as automatically tuning these values by using machine learning and analytical models [20][23].

When using RTM the fallback mechanism also needs a lot of consideration. Intel recommends the use a single global lock as fallback. Aborted threads, said to be running pessimistically, acquire this lock and block commits from other threads until they commit. This can lead to the *lemming* effect, where threads are constantly aborting by not acquiring the lock forcing them into executing pessimistically and blocking further threads, eventually serializing execution [24].

Sun Microsystems also announced hardware support for transactional execution in its Rock processor. However this processor was never made commercially available. Vega processors from Azul Systems also support HTM, however its programming interface was not disclosed. IBM first added HTM support to its Blue Gene/Q supercomputer, and now supports it in several different products, namely the POWER8 processor and the zEnterprise EC12 server [5].

IBM and Intel's implementation of HTM are pretty similar (with the exception of Blue Gene/Q), providing machine instructions to begin, end, and abort transactions. Blue Gene uses compiler-provided constructs to transactions, but it does not allow for custom abort logic, only tuning of the system provided code.

The main difference between the aforementioned processors's HTM are compared in Table 2.2. Transaction capacity (the maximum number of positions that can be accessed during a transaction) and conflict detection granularity are very much related to the processors cache characteristics, such as their capacity and the size of their cache lines. The number of abort codes is mentioned as it facilitates the building of fallback mechanism for handling aborted transactions.

Hybrid Transactional Memory

Since most HTM implementations are best-effort, proving no forward success guarantees, software fallbacks are necessary. This has led to development of systems called HyTM, where transactions execute both in software and hardware. While initial research used the software only as a fallback for aborted transactions, current research focusses on concurrently running transactions on both software and hardware. Notable examples of HyTM, with concurrent execution include NORecHy [25] and Invyswell [26].

NORecHy is based on the NoRec STM. Both hardware and software transactions check the Global lock, as previously reported. However some extra instrumentation exists, to avoid conflicts between in-flight software and hardware transactions. Hardware transactions first check the global lock, if they detect a software transaction committing, they wait for it to conclude and then abort. When committing, hardware transactions increment a per-core counter, instead of incrementing the global clock, although they also need to check it to avoid overlapping their commits with software-side commits. This means that in-flight software transactions need to check all counters before committing. The benefit of these per-core counters is that it avoids hardware transactions conflicting between each other. NORecHy was implemented for Sun's Rock processor and AMD's ASF proposal.

Invyswell is another HyTM, targeted at Intel's TSX architecture and making use of the RTM mode. It is based on InvalSTM, a STM that performs commit time invalidation, i.e., it resolves and identifies conflicts with all other in flight transactions. To achieve this InvalSTM stores its read and write sets in Bloom filters, allowing fast conflict detection. Using invalidation allows for good conflict resolution, making InvalSTM ideal for long transactions and high contention scenarios, which is the opposite of Intel's RTM, which is great for short transactions with low contention.

In Invyswell hardware transactions can be launched in two modes. The basic hardware transactions detect conflicts in a similar way to NoRec Hy. The more complex hardware transaction record their read and write sets in bloom filters, allowing invalidation to come from committing software transactions. This gives hardware threads the possibility to commit after a software commit with no conflicts, which always caused an abort at the hardware side in NoRec Hy.

Performance Comparison

All these different types of implementations of TM make performance comparisons between them a necessity. Lots of works [22],[5] exist in this area, which allow for a good overview of the differences between these implementations, and to find which workloads are better fit for each one.

Important metrics to classify TM performance are: transaction throughput, abort rates and energy efficiency. Transaction throughput is the obvious metric for comparisons. However, abort rates and energy efficiency, despite being related to throughput, can tell a lot about the TM's efficiency. These metrics are tested under different workloads, which typically depend on the following parameters: level of contention (i.e., number of expected conflict), transaction length, concurrency (numbers of threads), percentages of writes and temporal locality (number of repeated memory accesses to the same position).

An interesting result highlighted by Diegues and Romano [22] is that, at least for the set of workloads and platforms considered in that study, existing HyTM solutions are not competitive with both HTM and STM, both in regards to throughput and energy efficiency. Further study has found that the very expensive synchronization mechanisms, that need to exist in order to guarantee Opacity, between hardware and software transactions often introduce large overheads. Since most HTM do not implement escape actions - non transactional operations during transactions - HyTM implementations often resort to using global locks, which severely degrade performance.

When comparing STM and HTM the differences in performance are much more workload dependent.

Research has found that the main disparity between these, that is not dependent on the capacity limitations of HTM, is handling contention. STM perform better under high contention scenarios, in terms of throughput. Whilst decreases in locality affect performance of HTM greatly, they have almost no impact for the software solutions.

Since HTM implementations are limited by physical resources, they cannot scale past a certain number of threads or a certain transaction duration. Long transactions can be handled much better in software as conflicts can be solved in many different ways. Software also scales much better with the number of threads as the instrumentation overhead starts to become less relevant with the performance boost gained when running multiple threads. Most STM implementations also have increased energy efficiency when running multiple threads as spinning for locks is more probable.

However when running in low thread counts HTM proves to be much better then STM in most benchmarks, as the instrumentation overhead is more influential. When running benchmarks with low contention, HTM outperforms STM in both throughput and energy efficiency, and scales well until high levels of contention. In high contention scenarios HTM performs much worse, and expends more energy due to the large number of aborts. Performance measurements for HTM need to be taken with some care as most testing is done in a single configuration, and disregard the tuning problem talked about earlier. Studies [20] have shown that the current tuning can generate speedups of as much as 80%.

Amongst STM SwissTM and TinySTM are typically the best performers, altough NoRec is competitive in most scenarios, especially for low thread counts. In regards to HTM none of the implementations available are clearly superior, with each being the best in specific workloads.

2.1.2 Scheduling in Transactional Memory

Contention Managers were previously mentioned as being a reactive solution for conflicts. The one used in SwissTM is an example of this - upon detecting a conflict, it acts, causing one, or both, of the transactions involved to abort. This presents an advantage over always aborting the thread that detects the conflict as, for instance, it can protect longer running transactions's execution.

Schedulers are a more proactive solution to conflicts, trying to avoid them before they occur. This is done by deciding when it is best to execute a transaction. It is important to note that, contention managers and schedulers have the same final goal, which is maximizing performance, and in some situations they may even be run in tandem.

Steal-on-abort [27], is an example of this, a scheduler that is compatible with contention managers. Steal-an-abort keeps a queue of transactions to execute. This queues exist per thread, however threads may steal transactions from others' queues when theirs are empty. Upon conflict detection, instead of immediately restarting the aborted transaction, Steal-on-abort changes it to the same queue as that of the transaction that aborted it, as transactions that have conflicted in the past are likely to do it again.

Another approach to scheduling is ATS [28]. ATS uses a per thread contention-intensity (named CI) variable. This variable's value increases with each abort and decreases with each commit, however it does weigh these values the same. With a correct balance of these weights CI can be used to predict the likelihood of a transaction aborting. In this way CI is used to detect high contention, and ATS takes action

when the CI passes a certain treshold, serializing transaction execution. In the worst case scenario this degenerates to what is effectively a single global lock, however since the CI value is per thread, it is likely that some parallel execution still occurs.

To work properly ATS only needs to know if transactions have committed and/or aborted, making its overhead very small. On the other hand Steal-on-Abort needs to know which pairs of transactions conflicted, which is much more costly or not even possible when using HTM. Obtaining this sort of precise/detailed information introduces a much larger overhead, which can be affordable for STM, but is prohibitive in HTM contexts.

Seer [29] is a scheduler that was designed with HTM in mind. Since the information needed to correctly implement a scheduler is not available on HTM, Seer uses a probabilistic, self-tuning model to infer the most problematic conflict patterns among different transaction types. This approach would be very inefficient in software, where this information can be acquired in other ways, however in hardware it is the only approach possible. It then exploits the inferred information on transactions' conflicts to perform a priori lock acquisitions and avoid aborts in the future.

2.1.3 Benchmarking Transactional Memory

Since TM is a relatively new abstraction, very few applications have been written that support this abstraction. Thus to evaluate TM researchers need to resort to using benchmarks. Just like there is a variety of TM implementations, there also exist several benchmarks. These benchmarks can usually be split in two categories: microbenchmarks and realistic benchmarks. The first refer to simple problems - e.g. manipulating data in a red-black tree. The second, for which several examples will be presented bellow, tries to mimic the data access patterns of real applications.

A good example of realistic benchmarks is STMbench7 [30]. STMbench7's core data-structures are based on OO7 [31], a well-known object-oriented database benchmark. It contains 45 different operations on a single graph-based shared data structure, which range from small reads to complex alterations. This variety of operations is what distinguishes realistic benchmarks from microbenchmarks, as real applications' access patterns usually have a large variety in duration and type. In the STMbench7's case the memory access patterns mimic those of CAD/CAM applications.

An other example of a realistic benchmark is Lee-TM [32]. Lee-TM is based on Lee's routing algorithm - a popular circuit routing algorithm. It was developed with the intent of being a non-trivial benchmark, meaning it is difficult to implement with fine-grained locks, but with large potential parallelism and with varying types and durations of transactions. Being based on a real-world application increases the confidence level of this benchmark. Another advantage of this is that results produced by running Lee-TM can be easily verified by comparing the final data structure with the initial one.

The most popular benchmark suit for TM is Stanford Transactional Applications for Multi-Processing (STAMP) [8]. STAMP is a suite comprised by 8 different benchmarks, each based on a different real world application. Since it uses multiple benchmarks, with varying access patterns and characteristics, it allows for a more complete analysis of TM. It is also portable across multiple types of TM, including software, hardware and hybrid. This is an important property as it allows comparative testing between

these systems. It is also important to note that is an open-source project, which facilitates porting it to new implementations - such as the one proposed in this work.

Like the two previously mentioned benchmarks, STAMP's benchmarks were designed to be realistic benchmarks, however, unlike STMBench7 which requires user configuration for the desired access types, each of the STAMP's benchmarks has their own properties. These cover a wide range of use cases, including consumer focussed applications and research software. This variety makes it so that running all 8 benchmarks on a system provides a good characterization of its strengths and weaknesses.

Few applications make use of TM, however those that already do so can also be used as benchmarks. Memcached [33], a popular distributed memory object caching system, is an example of this. Memcached's typical workloads are read heavy, with few writes, making them ideal for TM. In addition it sheds light on the performance of TM in real world applications. Memcached requires lock acquisition for accesses to the hashtable that is its central data structure, but also uses locks for memory allocation - which require irrevocable transactions. This makes Memcached easily transactifiable. Further, since the lock-based version is already pretty optimized, it is a great benchmark to compare different TM implementations to a baseline lock implementation.

2.2 General Purpose Computing on GPUs

GPUs are hardware accelerators traditionally used for graphics processing. The first GPU was released by NVIDIA in 1999. The huge market demand for realtime, high-definition 3D graphics both in the games industry and other speciality industries like CAD design has led to a steady increase in processing power - putting their raw operation throughput over that of most CPUs [4]. This demand also shaped their architectures, whilst GPUs were originally special purpose accelerators with a very rigid pipeline, they now feature a more unified architecture [4].

This core unification, along with other architectural changes, has turned GPUs into a highly parallel, multi-threaded, manycore processor. The combination of these changes, along with the great performance, has made them a hot topic amongst researchers. Even before the existence of programming models for GPUs, researchers were using graphic Application Programming Interface (API) such as OpenGL to process applications other than graphics. Research focused on using these GPUs, already present on most consumer grade computers, for non-graphics applications is known as GPGPU.

GPGPU is now a very active research subject as GPUs have proven to be very efficient, especially for highly data-parallel problems where their many core architecture can be fully exploited. Currently GPUs are used in several applications domains, ranging from scientific research [34] to real-time video encoding [35]. However achieving high efficiency on GPUs is a non-trivial task, as it necessary to adopt specific programming models and extract parallelism at a much finer grade level than that of multi-threaded CPU code.

Two programming models exist for GPGPU, namely: OpenCL and Nvidia's CUDA. OpenCL [36] is the open programming standard for heterogeneous platforms, developed by the Khronos Group, and its programming model is based on CUDA's. However, due to being platform agnostic, it has a bigger over-

head, when compared to CUDA's and also lacks some of the more powerful device-specific primitives, specifically those that allow for finer grained control over the hardware. For this reason, the following analysis will be focused on CUDA and NVidia's architecture.

2.2.1 GPU Architecture Overview

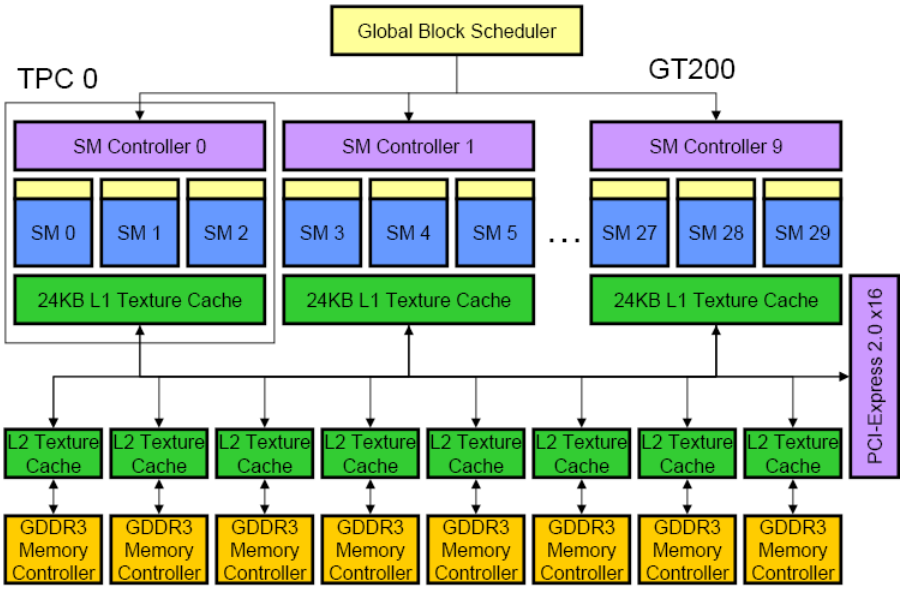

Figure 2.1: Architecture for a second generation Tesla GPU. Adapted from [1].

Nvidia first introduced a unified graphics pipeline with the Tesla architecture. Previously, graphics cards consisted of multiple specific processing units such as vertex or pixel processors. Whilst this provided great peak performance, achieving that was difficult as for example pixel processors could not execute any tasks directed at the vertex processors. With the Tesla architecture this cores were merged into a single, more versatile core - named a CUDA core.

CUDA cores are then grouped into Stream Multiprocessor (SM), were they execute instructions as if the SM was a Single Instruction Mutiple Data (SIMD) processor. This method of execution is named SIMT. Each SM has its own instruction unit, core specific register file and even their own shared memory. A GPU contains multiple SMs, all connected to the global DRAM. In all architectures, except for Tesla, the SMs feature their own Read-only Texture Cache and an L1 cache. The global memory, and the L2 cache in all architectures after Tesla, are shared by all SMs.

The base architecture released with the Tesla line, depicted on Figure 2.1, has undergone slight modifications in subsequent Nvidia architectures, most notably the differences around the composition of the SMs, and the unification of the L2 cache.

As presented in Figure 2.1, the Global Clock Scheduler distributes blocks of threads in a round-robin fashion to SMs which have sufficient resources to execute it. Each SM executes their threads independently of the other SM's state, unless synchronization is specifically called for. SMs then use their own schedulers to decide how to execute the blocks of threads attributed to them. Each instruction issued by the SMs is sent to multiple CUDA cores, i.e., it is executed across a group of cores.

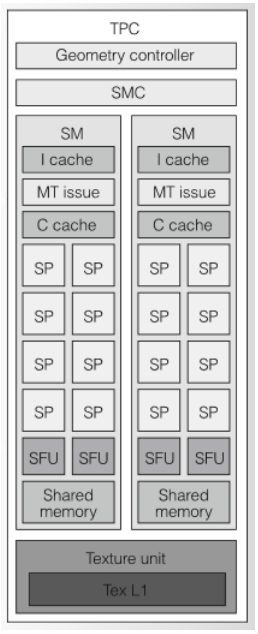

Figure 2.2: First generation Tesla Texture/Processor Cluster (TPC). Adapted from [2].

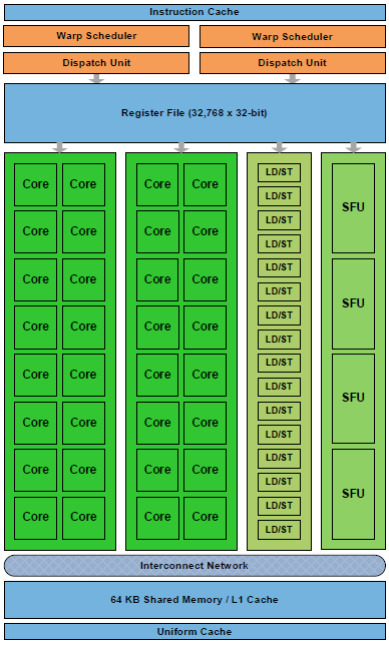

Figure 2.3: Fermi's Streaming Multiprocessor. Adapted from [3].

Nvidia Tesla Architecture

The first Tesla cards were released in 2008. This architecture featured 8 CUDA cores per SM, which are further grouped into a TPC. Each TPC has its own read-only L1 texture cache, shared between all SMs within the TPC [1]. The TPC's architecture is presented in Figure 2.2

Tesla's TPC features a SMC, which distributes work amongst the contained SM. Each SM features its own instruction fetch, decode and dispatch stages. To achieve maximum efficiency, the GPUs use multi-threading, but they lack speculative execution, as most CPUs use. At the instruction issue time, the unit selects a warp - group of threads - that is ready to execute and issues its next instruction [2]. To increase memory throughput, accesses to consecutive Global Memory positions within the same SM are coalesced.

The TPC also features a per SM shared memory, which is used for inter-thread synchronization. However this memory is only accessible by CUDA cores within a single SM, hence synchronization between SMs requires special instructions to stall other SM's pipelines. Each SM also features 2 Special Function Unit (SFU) units, which are used for transcendental functions such as sines or square roots. Moreover the SFUs also include two double precision floating-point units, since each CUDA core only has an integer Arithmetic Logic Unit (ALU) and a single precision floating-point unit.

Nvidia Fermi Architecture

Fermi is the successor of the Telsa architecture. It was launched in 2010, and features numerous changes over the Tesla line. The most notable ones are the removal of the TPC, making each SM fully independent, and the unification of the L2 cache [3]. Its SM's architecture can be seen in Figure 2.3.

Fermi's SM features 32 cores, 16, Load/Store units, 4 SFU units and 2 instruction schedule and

dispatch units, meaning that each clock cycle two different instructions from two different warps are issued. The Shared Memory and the L1 cache are now under a single unit. This unit has 64KB of memory, which can partitioned, by the software, between the Cache and the Shared memory.

CUDA cores were also changed. The core's integer ALU, was changed to support full precision in multiplication operations. Additionally each CUDA core now features a new floating-point unit. The new floating-point unit supports double precision and is fully complaint with the latest IEEE standards, supporting operations like the Fused Multiply-Add instruction. Since the floating-point logic is now inside the CUDA cores the SFU units are now only used for transcendental functions.

Fermi also introduced an extended Instruction Set Architecture (ISA), which aims to improve both programmability and performance. This improvements include features such as unified address space and 64-bit addressing so as to provide the C++ support in the CUDA programming model.

Nvidia Kepler Architecture

The Kepler line was launched in 2012. Nvidia's goals with this architecture were to improve performance and energy efficiency. Kepler's SMs are named SMX. The SMX features 192 CUDA cores, 32 load/store units and 32 SFUs. Each SMX has a bigger computational power than Fermi's SMs, however Kepler GPUs feature less SMXs for bigger power efficiency [37].

Along with the increase in cores per SM, each SMX features 4 warp schedulers. Unlike previous architectures, for each warp scheduler, Kepler uses 2 instruction dispatchers. This means that for each warp, 2 independent instructions are launched in parallel. Additionally high performance Kepler GPUs feature 64 double-precision units, to further increase performance in this area.

Nvidia Maxwell Architecture

The latest line by Nvidia, Maxwell, was released in 2014. Following the change in naming schemes already present in the Kepler line, Maxwell's SM are named SMM. SMMs feature less CUDA cores per SM when compared to the SMX, 128 as opposed to 192, as a non-power of two number of cores made efficient usage of the hardware by coders very difficult. The previously high-performance only double-precision units are now baseline, and 4 exist in each SMM [38].

The biggest change is the split between L1 cache and Shared Memory. The L1 cache is now in the same module as the Texture cache, using the same partitioning scheme it already had with the shared memory, but this time with the Texture cache. The Shared Memory is now independent has a larger capacity. These memory improvements, along with better scheduling, provide increased performance in each CUDA core, making the SMM's performance very similar to the SMX's.

2.2.2 CUDA Programming Model

Nvidia's CUDA is a scalable programming model, based on the C programming language [4], working across several different micro-architectures of Nvidia GPUs. A CUDA program starts with a single-threaded function, which is the base unit of execution. This function is called a kernel and its the code that the GPU runs.

Since GPUs use a SIMT execution model, a single thread can not be launched alone, which is why threads are grouped into warps. A warp is a group of 32 threads, running simultaneously. Conditional branches may cause threads within the same warp to execute different instructions, a phenomenon called thread divergence. When this happens both branch paths are executed serially, whilst the threads that took the other branch are deactivated. This continues until all threads reconverge, which may severely impact the kernel's performance.

Additionally, warps are also grouped into blocks. The blocks are what the Global Clock Scheduler issues to each SM. The block structure facilitates scaling the code across multiple architectures. If an older GPU features less SMs, more blocks are issued to each SM, whilst on a newer card blocks are split across all the available SM increasing performance. Both block size and number of blocks are defined by the coder, while the warp size is fixed. This organization, along with the GPU's memory hierarchy is pictured in Fig 2.4.

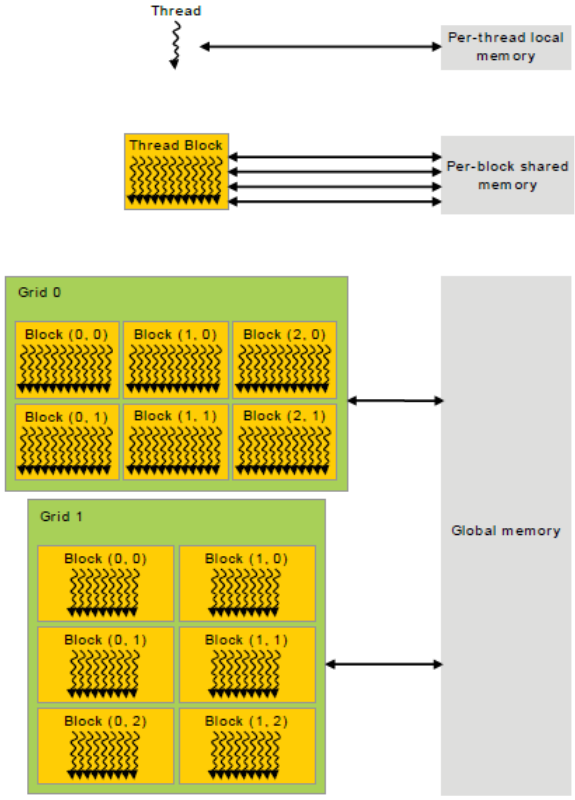

Figure 2.4: CUDA thread and memory organization. Adapted from [4].

To create a CUDA program, the coder starts by defining the kernel. Before running the kernel, global memory needs to be allocated and any data that is necessary must be transferred from the host to the GPU. Launching the kernel involves selecting which kernel to launch, along with the number of blocks and the number of threads per block. All data needs to explicitly transferred both to and from the GPU. To help coders with paralleling their code CUDA provides several important primitives so that coder can leverage its internal organizations, such as *threadIdx* (per-block unique thread identifier) or the *blockIdx* (unique block identifier).

It is important to note that GPUs are hardware accelerators and physically separated devices, and

therefore communication between them and the CPU is performed over the PCI Express busses, usually with very limited bandwidths. In a nutshell the GPU communicates with the CPU to either transfer data or to notify it of completing the tasks that the CPU has ordered. Whilst this easily allows CPU and GPU to execute code in parallel, it also introduces some difficulties when it comes to sharing data, as both devices have their own separate memories.

To counteract this Nvidia introduced zero-copy. Using zero-copy, the host (CPU) can pin a region of its own memory to be used for the GPU. Using this approach, the coders see the system as if the GPU global memory is replaced by the host memory, where all data is transferred to and from the GPU transparently. Building on this approach, CUDA 6.0 introduced Unified Memory, where the CPU and the GPU share a single addressing space, using the same pointers for data that may be in one, or the others physical memory. This allows both systems to operate on the data concurrently, however race conditions may still happen between devices - which hampers usage of this system.

Regarding the GPU's memory, some considerations need to be taken by the programmer. As with any multi-thread system, memory races are a problem. To avoid the interference of other threads when executing an instructions coders can use Atomic operations. These operations are conflict-free between multiple threads. However interference from other threads may still occur when accessing the Global Memory, as accesses to this memory's are weakly-ordered. This means that, for instance, each thread sees a different order of writes to the Global Memory. To guarantee a consistent ordering, coders must make use of Nvidia's memory fences, which make all writes visible to other threads.

2.3 Transactional Memory on GPUs

As research develops on the field of GPGPU, ensuring that concurrent accesses to shared memory are consistent is a more relevant issue than ever. In an environment with hundreds of threads, bugs can manifest in new manners, and situations such as livelock or deadlock are much more common. Locking in a GPU architecture is a much complex task.

First of all, fine-grained locking on GPUs is more prone to livelocks, as the large number of threads increases the probability of circular locking occurring. Circular locking occurs when 2 threads, t1 and t2, both need to acquire 2 locks, l1 and l2, and do so in different orders: t1 acquires l1, and t2 acquires l2. Now t1 must acquire l2, and t2 must acquire l1, so both threads either get stuck waiting for each other (deadlock) or they abort. If threads abort and restart execution at the same time, this can lead to a situation where they get stuck in a loop of locking each other out and aborting, which is called a livelock. On the other hand, coarse-grained locking is heavily undesirable on GPUs, as it kills the performance gain from running multiple threads.

The memory model must also be considered. The unordered accesses to the global memory cause significant hurdles for any synchronous code, as it allows for situations like two threads acquiring the same lock. To solve this, programmers must make extensive use of the memory fence primitives. It is also important to keep in mind the fact that the L1 caches are not coherent and must not be used to store any critical shared data necessary for synchronization.

Table 2.3: Comparison between existing GPU TM implementations. Mixed conflict detection refers to Eager read-after-write detection and Lazy write-after-read detection.

	Conflict Detection	Reads	Writes	Synch Strategy	Type
ESTM [40]	Eager	Visible	In-place	Metadata Based	Software
PSTM [40]	Eager	Visible	In-place	Metadata Based	Software
ISTM [40]	Mixed	Invisible	In-place	Lock Based	Software
GPU-STM [41]	Mixed	Invisible	Defered	Lock Based	Software
PR-STM [42]	Mixed	Invisible	Defered	Lock Based	Software
KILO TM [39]	Lazy	Invisible	Defered	Value Based	Hardware

As such, the TM abstractions has gained added importance in the scope of these challenges. Several implementations already exist and Table 2.3 presents a comparison between state-of-the-art solutions. In the existing literature, the attempts at hardware based solutions on GPU, such as KILO-TM [39], are rare. Much like HTM on CPU, these solutions present many difficulties for research purposes, as producing custom hardware is extremely expensive and simulating it is very time consuming and can produce unreliable results.

For these reasons, the rest of this chapter focuses on analysing software solutions for GPU TM. Although a lot of CPU implementations were previously analysed in this work, their flexibility, along side with the limitations for GPU based designs, causes the latter to better fit as inspiration for the heterogeneous implementation herein proposed, as it is easier to adapt a CPU based TM for our purposes than a GPU based one.

2.3.1 GPU-STM

GPU-STM [41] is word and lock-based STM. At its core GPU-STM uses the same time-based validation scheme as TinySTM. However, in a SIMT architecture the number of commits is very high and version counters, especially when they map to a set of memory positions, are unreliable. To compensate for this, when time-based validation fails, a NORec-like value-based validation scheme is triggered to ensure it is not a false positive. This method, called hierarchical validation ensures a much smaller rate of false positives and, despite its larger overhead, which is more desirable on SIMT architectures.

Unlike NORec, GPU-STM uses encounter-time lock sorting. Lock sorting entails saving the lock's address in a local ordered data structure, and not acquiring them. Locks are then acquired at commit time by the system, following the order in which they were sorted. This means that all locks are acquired in certain order, the same in all threads. Unlike unordered acquisition, the ordered system avoids circular locking, as if two threads need to acquire a set of locks, they will acquire them in the same order.

Altough GPU-STM provides some isolation, it does not ensure opacity, as the transactions that are aborted continue to execute till they fail validation. This makes it very undesirable as the base for the proposed research, as opacity is the main correctness criteria for TM.

2.3.2 Lightweight Software Transactions on GPUs

Holey et all. [40], proposed 3 variations of their implementation: ESTM, PSTM and ISTM. ESTM is the reference implementation. Each of these implementations is targeted at different workloads, so that

the coder can choose the one that best fits his situation. All three solutions use speculative in-place writes and thus must save the previous value of positions written to an undo log. To avoid livelocks exponential backoff on aborts is used.

ESTM uses a complex shared metadata structure, with one to one mapping to the shared memory being transactionalized. This structure, called shadow-memory, mimics other STM's metadata such as lock arrays or version counter. It contains: a read clock, a thread id of the last access, a bit indicating a write, another bit indicating a read and one for shared access. It also includes a lock that is acquired to modify a entry in the metadata in the structure. Read after write conflicts are detected eagerly using this data structure as, if the write bit is set, the transaction aborts immediately. All other metadata is thread-private and consists of a read log and a undo log.

To validate, the transaction first checks its reads' shadow entries to see if they been modified, then it checks if its writes have either been written to and/or speculatively read. If any of these steps fails, the transaction aborts. In case of an abort writes are reverted back to their original value using its undo log. Otherwise, the writes' shadow entries are updated by changing the modified bit back to zero and incrementing the clock value.

PSTM uses a very similar system, however since it was designed for workloads that frequently write to the same positions that they read, it does not distinguish between reads an writes. Instead of using bits to express if a position as been read or written to, PSTM uses the thread id of the last access. If it is the same or zero (reserved id, not attributed to any thread) the transaction continues, otherwise it aborts. When either committing or aborting the thread releases all the shadow entries it has modified by changing their value back to zero. This operations uses an atomic Compare-and-swap and it essentially equates to using fine-grained locks. To avoid livelocks, threads abort on failed lock acquisition, and try to avoid further conflicts by using an exponential backoff .

ISTM implements versioned locks with the intent of allowing invisible reads, which closely resemble the ones used by GPU-STM. For this implementation the shadow memory contains versioned locks, which consist of a version counter, with the least significant bit being used as a lock. This type of locks is very similar to other time-based STMs, like TinySTM. If a thread attempts to read a locked position, it aborts unless it owns that lock (which means it has already written to that position in this transaction). If the lock is free, the thread reads the value and saves its address and version to the local read log. The lock is only acquired for writes.

On commit-time ISTM validates its reads by checking if the local version is consistent with the one in the Global Memory. If this fails the thread releases the acquired locks and reverts its changes, otherwise it both frees the locks and increments them, to notify other threads that it has successfully committed.

2.3.3 PR-STM

The last STM presented is PR-STM [42]. It is a lock based STM, using versioned locks for both validation and conflict detection. It works in a similar way to the already presented ISTM, as reads are invisible, but it uses defered writes. The global lock table is also different from ISTM's, in that its mapping is done using hashing and does not need to be one to one, i.e., several memory locations can use a

single lock. While this may increase the rate of conflicts, it also decreases the memory overhead of the lock metadata. This flexibility is very important as the GPUs memory is quite limited.

The most important innovation presented by this TM is the static contention manager. In PR-STM each thread is assigned a unique priority, and locking is a two stage process, involving acquiring two different locks: the write-lock and the pre-lock. Threads with higher priorities can ”steal” pre-locks, from lower priority threads. This means that, on conflict detection when acquiring a lock, at least one thread will continue, avoiding livelocks and deadlocks.

When compared to GPU-STM, PRSTM's lock-stealing allows it to have a much smaller overhead. Sorting the locks is a non-trivial operation, if it is done at commit time the complexity is $n * log(n)$, otherwise, if done as GPU-STM instead, then the instrumentation overhead for each operation in a transaction increases non-trivially.

In PR-STM, threads only detect conflicts when they fail to acquire a lock, or when they try to read a write locked position. Validation starts with checking if the local version value matches the global one, for each entry in its read and write sets. Each success leads to pre-locking that memory position. If all pre-locks are acquired with success validation continues, by checking if any pre-lock was stolen. If this fails the transaction aborts and releases all its locks, otherwise it is free to commit.

PR-STM's relatively low memory overhead allows a greater degree of modifications to its design. Additionally the researchers were kind enough to provide access to source code for inspection and modification. These factors, along with it being the recent publication in this area, make it the implementation of choice for the work proposed here.

2.4 Summary

In this chapter both TM and GPGPU were discussed, ending with the concept of GPU TM. TM is an abstraction that moves the complexity of locking shared-data to a software/hardware library, whilst retaining performance similar to that of fine-grained locking.

In this abstraction critical sections of code are marked as transactions. From the coders perspective these are executed as if they were atomic operations, but in reality transactions are speculatively executed in parallel, and use mechanism to detect and roll-back illegal operations. Several TM implementations are analysed and compared from the perspective of performance and energy consumption. Another important topic for the work proposed in this paper, scheduling, was also introduced. Finally an analysis on the most popular benchmarks for TM was done.

GPGPU is a relatively recent field of research, which focuses on leveraging GPU's processing power for non-graphics related tasks. Due to the high market demand for processing power on the GPU market, these have evolved into highly performant multicore processors, with a theoretical operation throughput higher than that of CPUs.

Despite their high theoretical performance, GPU programming is a difficult task. One of the biggest offenders in this area is synchronization, as the GPU memory model imposes lots of restrictions on coding. This motivates the usage TM solutions on GPU, which were discussed in the last section.

3

Work Proposal

Building on the previous chapter's research, a work proposal is herein presented. The proposed goal is to be able to schedule a transaction workload across CPU and GPU, maintaining correctness, with a competitive performance to a CPU/GPU only solution.

Since explicit synchronization between these units is very costly, the idea is to rely on speculative execution to minimize the risk of conflicts between threads running on different units. Speculation can be made through the use of partitioning techniques to split the workloads amongst units. Research in regards to partitioning TM workloads already exists 43, which supports this approach.

The system is planned to function as follows: at any point in time the host (CPU) is the authoritative copy of an application's address space. Yet, it can assign leases on memory regions to a set of threads that are executing on the GPU. Synchronization among threads running on the CPU and threads running on the GPU occurs in a lazy fashion, e.g., periodically or when the set of transactions to be executed on the CPU/GPU is completed. However, this approach raises several challenges, such as how and when to detect conflicts between threads running in different units and what to do in case of a conflict.

The steps taken to achieve this work are the following:

1. Study the state-of-the-art on both TM and GPGPU.
2. Modify PR-STM [42] to support logging, enabling the host to check if any illegal accesses were made.
3. Develop and implement a scheduler capable of partitioning transactions amongst CPU and GPU, whilstavoiding conflicts with a large degree of confidence.
4. Using the previous system, develop a conflict resolution system to enable running transactions on both units concurrently, assuring correctness.

5. Test the final solution, in several scenarios, analysing both raw throughput and conflicts, as well as considering energy consumption.

The first step is presented in Chapter 2 of this work. In regards to the second step, some early work is presented in Section 3.1. The presence of logging is a necessity when using lazy synchronization, as errors can not be detected as they occur.

The central point to this work is the correct partitioning of transactions amongst the two units. Research [43] already exist in memory partitioning for the STAMP benchmark suite, however it is not clear if this solution would work for this system. Further research and experimentations is required for this step.

Following this, the final system will be built by integrating the scheduler with the CPU side STM, likely to be TinySTM, and the modified PR-STM. Finally, the produced solution will be tested in several scenarios, with regards to both performance and energy consumption. A comparison between it and CPU/GPU only systems will also be produced.

3.1 Experimental Results

To validate the work proposal presented in this chapter, PR-STM was modified to include logging. As previously pointed out, the presence of some form of logging is necessary to allow speculative execution. As such two solutions were developed - by modifying PR-STM.

The first solution is an explicit log, which records to memory every address read and written upon commit time. Whilst this results in a small overhead in terms of instrumentation, it does come at the cost of a rather large memory overhead. Additionally, if the transaction size is not fixed, we run into memory allocation problems, as this is not a capability of device GPU code.

To address these issues another solution was proposed, this time using Bloom filters to record addresses. Bloom filters are probabilistic data structures used to represent whether or not an element is part of a set. It is important to keep in mind that Bloom filters require a larger instrumentation overhead.

Both solutions were tested. The test consisted of running TinySTM's Bank benchmark, which defines each memory address of its ”bank” array as bank account and uses transactions to ”transfer” money between them. The test used 50000 accounts and a small number of operations, which was slowly incremented by increasing the thread counts and thus resulted in increased contention. For this test a Nvidia GeForce GTX 680 GPU was used.

The obtained results show that both solutions - explicit logging and Bloom filter - introduce only a small overhead in performance, validating their usage. In fact, as pictured in Fig. 3.1, the performance difference is only relevant for either low threads counts, a situations that is not favourable for GPUs or a Bloom filter using a large number of hash functions.

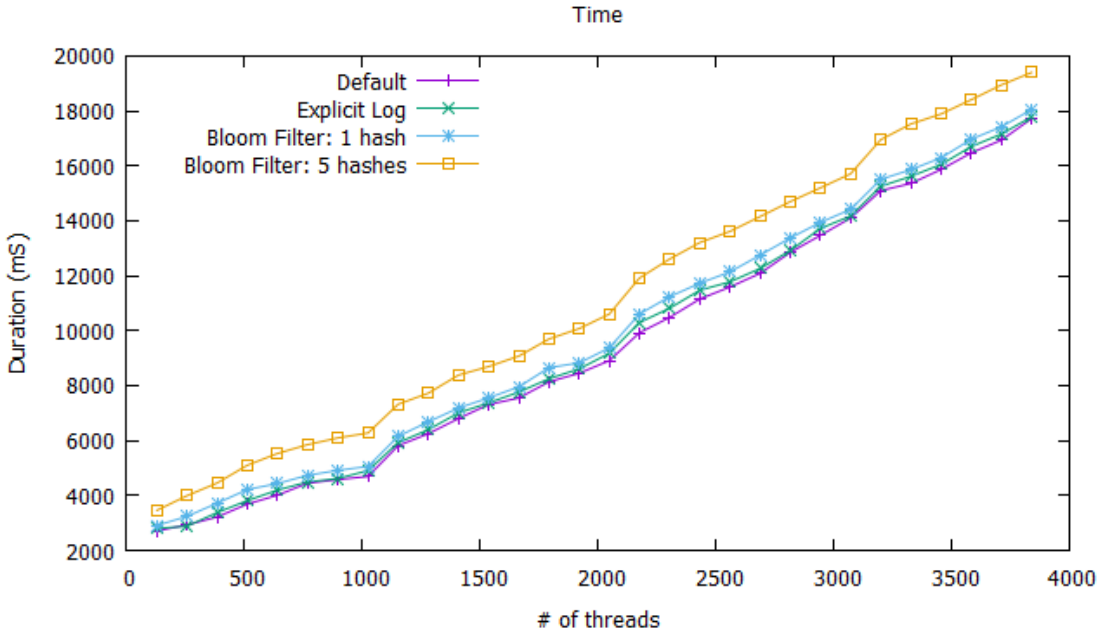

Figure 3.1: Comparison of execution time for the various solutions.

3.2 Work Plan

For the work proposed earlier in this Chapter the following calendar structure depicted on 3.2 is planned. The delivery goal is set for July 2016, however, since no other works on this area exist, these dates are purely speculative and delays may occur, which is why the planning stretches until September.

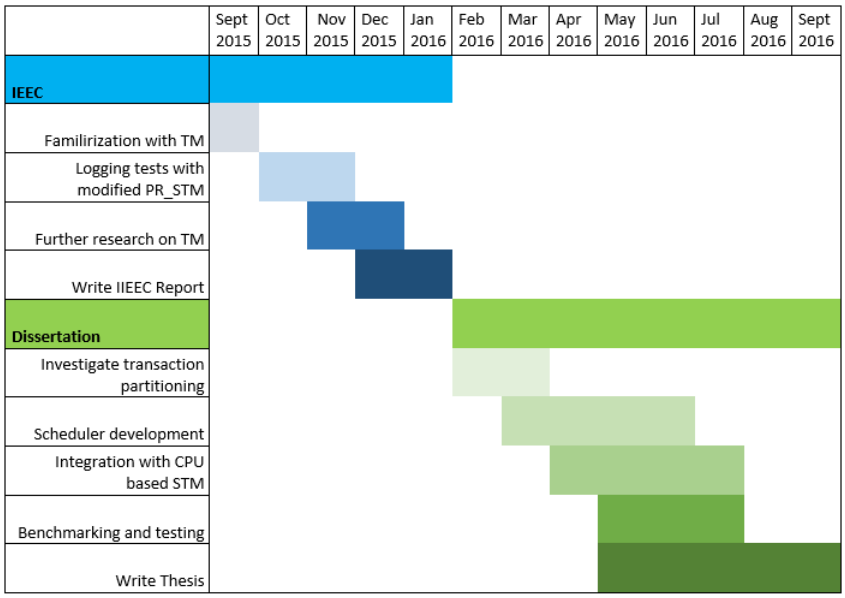

Figure 3.2: Calendar structure for the proposed work.

4

Conclusions

In this work, an analysis on the current state-of-the-art on TM and GPGPU was presented. The first area recently gained relevance with several hardware producers including it in their systems, whilst the latter has been a hot topic amongst researchers ever since the release of the first general programming models.

TM is an abstraction that moves the burden of shared memory access synchronization to the underlying system. This facilitates coding whilst still maintaining a competitive performance. GPGPU is the usage of GPUs for non-graphics tasks. This unit's peak computational capacity is vastly superior CPU's, which makes using it for other tasks an interesting proposition.

Despite their performance, programming for GPUs is an arduous task, - especially when synchronism with the CPU is required. To remove some of the complexity in programming for GPUs, TM implementations are proposed for use in GPGPU.

In order to remove some part of this complexity, and in the light of the analysis of the state of the art in the areas of TM and GPU, this dissertation aims to propose the first TM implementation that allows for sharing data among threads running concurrently on GPU and GPU. The proposed approach to this challenge is to speculatively attribute workloads to both units, log all the memory accesses during execution and then check for conflicts using this log.

Finally a calendar structure was proposed. Whilst it is expected that this work concludes in the presented time frame, working in area with no previous research implies a large degree of uncertainty, especially in regards to length of the steps taken.

References

[1] D. Kanter, “Nvidia’s gt200: Inside a parallel processor,” http://www.realworldtech.com/gt200, 2008.

[2] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, “Nvidia tesla: A unified graphics and computing architecture,” IEEE micro, no. 2, pp. 39–55, 2008.

[3] N. Corporation, “Whitepaper - nvidia’s fermi - the first complete gpu computing architecture,” http://www.nvidia.com/content/pdf/fermi_white_papers/p.glaskowsky_nvidia’s_fermi-the_first_complete_gpu_architecture.pdf, 2010.

[4] ——, “Cuda c programming guide,” https://docs.nvidia.com/cuda/cuda-c-programming-guide/, 2015.

[5] T. Nakaike, R. Odaira, M. Gaudet, M. Michael, and H. Tomari, “Quantitative comparison of hardware transactional memory for blue gene/q, zenterprise ec12, intel core, and power8,” in Computer Architecture (ISCA), 2015 ACM/IEEE 42nd Annual International Symposium on, June 2015, pp. 144–157.

[6] M. Herlihy and J. E. B. Moss, “Transactional memory: Architectural support for lock-free data structures,” SIGARCH Comput. Archit. News, vol. 21, no. 2, pp. 289–300, May 1993. [Online]. Available: http://doi.acm.org/10.1145/173682.165164

[7] T. H. Hetherington, M. O’Connor, and T. M. Aamodt, “Memcachedgpu: Scaling-up scale-out key-value stores,” in Proceedings of the Sixth ACM Symposium on Cloud Computing, ser. SoCC ’15. New York, NY, USA: ACM, 2015, pp. 43–57. [Online]. Available: http://doi.acm.org/10.1145/2806777.2806836

[8] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun, “Stamp: Stanford transactional applications for multi-processing,” in Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on. IEEE, 2008, pp. 35–46.

[9] P. Felber, C. Fetzer, and T. Riegel, “Dynamic performance tuning of word-based software transactional memory,” in PPoPP ’08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. New York, NY, USA: ACM, 2008, pp. 237–246.

[10] R. M. Yoo, C. J. Hughes, K. Lai, and R. Rajwar, “Performance evaluation of intel® transactional synchronization extensions for high-performance computing,” in Proceedings of the International Conference on High Performance Computing, Networking, Storage and

Analysis, ser. SC '13. New York, NY, USA: ACM, 2013, pp. 19:1–19:11. [Online]. Available: http://doi.acm.org/10.1145/2503210.2503232

[11] C. H. Papadimitriou, "The serializability of concurrent database updates," J. ACM, vol. 26, no. 4, pp. 631–653, Oct. 1979. [Online]. Available: http://doi.acm.org/10.1145/322154.322158

[12] R. Guerraoui and M. Kapalka, "On the correctness of transactional memory," in Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, ser. PPoPP '08. New York, NY, USA: ACM, 2008, pp. 175–184. [Online]. Available: http://doi.acm.org/10.1145/1345206.1345233

[13] L. Dalessandro, M. F. Spear, and M. L. Scott, "Norec: streamlining stm by abolishing ownership records," in ACM Sigplan Notices, vol. 45, no. 5. ACM, 2010, pp. 67–78.

[14] D. Dice, O. Shalev, and N. Shavit, "Transactional locking ii," in Distributed Computing. Springer, 2006, pp. 194–208.

[15] A. Dragojević, R. Guerraoui, and M. Kapalka, "Stretching transactional memory," in ACM Sigplan Notices, vol. 44, no. 6. ACM, 2009, pp. 155–165.

[16] S. M. Fernandes and J. Cachopo, "Lock-free and scalable multi-version software transactional memory," in ACM SIGPLAN Notices, vol. 46, no. 8. ACM, 2011, pp. 179–188.

[17] J. a. Cachopo and A. Rito-Silva, "Versioned boxes as the basis for memory transactions," Sci. Comput. Program., vol. 63, no. 2, pp. 172–185, Dec. 2006. [Online]. Available: http://dx.doi.org/10.1016/j.scico.2006.05.009

[18] N. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," in Computer Architecture, 1990. Proceedings., 17th Annual International Symposium on, May 1990, pp. 364–373.

[19] J. L. Hennessy and D. A. Patterson, Computer Architecture, Fifth Edition: A Quantitative Approach, 5th ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011.

[20] N. Diegues and P. Romano, "Self-tuning intel transactional synchronization extensions," in 11th International Conference on Autonomic Computing (ICAC 14). Philadelphia, PA: USENIX Association, Jun. 2014, pp. 209–219. [Online]. Available: https://www.usenix.org/conference/icac14/technical-sessions/presentation/diegues

[21] B. Goel, R. Titos-Gil, A. Negi, S. Mckee, and P. Stenstrom, "Performance and energy analysis of the restricted transactional memory implementation on haswell," in Parallel and Distributed Processing Symposium, 2014 IEEE 28th International, May 2014, pp. 615–624.

[22] N. Diegues, P. Romano, and L. Rodrigues, "Virtues and limitations of commodity hardware transactional memory," in Proceedings of the 23rd International Conference on Parallel Architectures and Compilation, ser. PACT '14. New York, NY, USA: ACM, 2014, pp. 3–14. [Online]. Available: http://doi.acm.org/10.1145/2628071.2628080

[23] D. Rughetti, P. Romano, F. Quaglia, and B. Ciciani, “Automatic tuning of the parallelism degree in hardware transactional memory,” in Euro-Par 2014 Parallel Processing, ser. Lecture Notes in Computer Science, F. Silva, I. Dutra, and V. Santos Costa, Eds. Springer International Publishing, 2014, vol. 8632, pp. 475–486. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-09873-9_40

[24] D. Dice, Y. Lev, M. Moir, D. Nussbaum, and M. Olszewski, “Early experience with a commercial hardware transactional memory implementation,” Mountain View, CA, USA, Tech. Rep., 2009.

[25] L. Dalessandro, F. Carouge, S. White, Y. Lev, M. Moir, M. L. Scott, and M. F. Spear, “Hybrid norec: A case study in the effectiveness of best effort hardware transactional memory,” ACM SIGARCH Computer Architecture News, vol. 39, no. 1, pp. 39–52, 2011.

[26] I. Calciu, J. Gottschlich, T. Shpeisman, G. Pokam, and M. Herlihy, “Invyswell: a hybrid transactional memory for haswell’s restricted transactional memory,” in Proceedings of the 23rd international conference on Parallel architectures and compilation. ACM, 2014, pp. 187–200.

[27] M. Ansari, M. LujÃ¡n, C. Kotselidis, K. Jarvis, C. Kirkham, and I. Watson, “Steal-on-abort: Improving transactional memory performance through dynamic transaction reordering,” in High Performance Embedded Architectures and Compilers, ser. Lecture Notes in Computer Science, A. Seznec, J. Emer, M. OBoyle, M. Martonosi, and T. Ungerer, Eds. Springer Berlin Heidelberg, 2009, vol. 5409, pp. 4–18. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-92990-1_3

[28] R. M. Yoo and H.-H. S. Lee, “Adaptive transaction scheduling for transactional memory systems,” in Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures. ACM, 2008, pp. 169–178.

[29] N. Diegues, P. Romano, and S. Garbatov, “Seer: Probabilistic scheduling for hardware transactional memory,” in Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures, ser. SPAA ’15. New York, NY, USA: ACM, 2015, pp. 224–233. [Online]. Available: http://doi.acm.org/10.1145/2755573.2755578

[30] R. Guerraoui, M. Kapalka, and J. Vitek, “Stmbench7: a benchmark for software transactional memory,” Tech. Rep., 2006.

[31] M. J. Carey, D. J. DeWitt, and J. F. Naughton, “The 007 benchmark,” in Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, ser. SIGMOD ’93. New York, NY, USA: ACM, 1993, pp. 12–21. [Online]. Available: http://doi.acm.org/10.1145/170035.170041

[32] M. Ansari, C. Kotselidis, I. Watson, C. Kirkham, M. Luján, and K. Jarvis, “Lee-tm: A non-trivial benchmark suite for transactional memory,” in Algorithms and Architectures for Parallel Processing. Springer, 2008, pp. 196–207.

[33] W. Ruan, T. Vyas, Y. Liu, and M. Spear, “Transactionalizing legacy code: An experience report using gcc and memcached,” SIGPLAN Not., vol. 49, no. 4, pp. 399–412, Feb. 2014. [Online]. Available: http://doi.acm.org/10.1145/2644865.2541960

[34] D. Clarke, A. Ilic, A. Lastovetsky, L. Sousa, and Z. Zhong, “Design and Optimization of Scientific Applications for Highly Heterogeneous and Hierarchical HPC Platforms Using Functional Computation Performance Models,” in High-Performance Computing in Complex Environments, ser. Wiley Series on Parallel and Distributed Computing, E. Jeannot and J. Zilinskas, Eds. John Wiley & Sons, Inc., April 2014, ch. 13, pp. 237–260.

[35] S. Momcilovic, A. Ilic, N. Roma, and L. Sousa, “Dynamic Load Balancing for Real-Time Video Encoding on Heterogeneous CPU+GPU Systems,” IEEE Transactions on Multimedia, vol. 16, no. 1, pp. 108–121, January 2014.

[36] K. O. W. Group et al., “The opencl specification,” version, vol. 1, no. 29, p. 8, 2008.

[37] N. Corporation, “Whitepaper - nvidia’s next generation compute architecture: Kepler gk110,” https://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf, 2012.

[38] ——, “Whitepaper - nvidia geforce gtx 980,” http://international.download.nvidia.com/geforce-com/international/pdfs/GeForce_GTX_980_Whitepaper_FINAL.PDF, 2014.

[39] W. W. Fung, I. Singh, A. Brownsword, and T. M. Aamodt, “Hardware transactional memory for gpu architectures,” in Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture. ACM, 2011, pp. 296–307.

[40] A. Holey and A. Zhai, “Lightweight software transactions on gpus,” in Parallel Processing (ICPP), 2014 43rd International Conference on. IEEE, 2014, pp. 461–470.

[41] Y. Xu, R. Wang, N. Goswami, T. Li, L. Gao, and D. Qian, “Software transactional memory for gpu architectures,” in Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization. ACM, 2014, p. 1.

[42] Q. Shen, C. Sharp, W. Blewitt, G. Ushaw, and G. Morgan, “Pr-stm: Priority rule based software transactions for the gpu,” in Euro-Par 2015: Parallel Processing. Springer, 2015, pp. 361–372.

[43] T. Riegel, C. Fetzer, and P. Felber, “Automatic data partitioning in software transactional memories,” in Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures. ACM, 2008, pp. 152–159.