TÉCNICO LISBOA

Transactional Memory for Heterogeneous CPU-GPU Systems

Ricardo Manuel Nunes Vieira

Thesis to obtain the Master of Science Degree in

Electrical and Computer Engineering

Supervisors: Dr. Aleksandar Ilic
Dr. Paolo Romano

Examination Committee

Chairperson: Dr. Nuno Horta
Supervisor: Dr. Aleksandar Ilic
Members of the Committee: Dr. João Lourenço

November 2016

Acknowledgments

I would like to thank my supervisors: Professors Aleksandar Ilic and Paolo Romano, for all the patience, guidance and motivation during the development of this work. I would also like to thank Nuno Diegues, for his patience in helping a clueless student understand TM.

For the moral support and encouragement, I would like to thanks my parents and family, and of course my friends, especially those at IST.

Could not have done it without you!

Abstract

Over the course of the last decade, concerns with power consumption and demand for increasing processing power have lead micro-architectures to a highly parallel paradigm, with multiple cores available in each processing unit. Notably the GPU, previously a rigid, highly pipelined unit, now uses a versatile manycore architecture. These architectural changes, along with their raw computation power, led researchers to use GPUs for general purpose computing,

Whilst these parallel architectures allow for great peak performances, they also raise complex programming problems, especially in regards to synchronizing accesses to shared data. As a result, applications that strive to exploit the parallelism potential of heterogeneous systems equipped with both GPU and CPUs are notoriously difficult to develop.

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. Literature on TM has been prolific, and several TM implementations have been proposed targeting either CPUs or, more recently, GPUs.

This dissertation proposes Heterogeneous Transactional Memory (HeterosTM) the first (to the best of the author's knowledge) TM system capable of supporting concurrent execution of applications that exploit both CPUs and GPUs. In order to minimize synchronization overheads, HeterosTM relies on speculative techniques, which aim to amortize the costs of enforcing consistency among transactions executing on the different units. This system's architecture is presented and the tested using both a synthetic benchmark and a real application.

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 de maior poder de processamento levou as microarquitecturas a um paradigma altamente paralelo, com vários cores disponíveis em cada unidade. Especial atenção a GPUs, que de unidades rígidas e altamente pipelined passaram a uma versátil arquitetura manycore. Estas mudanças na arquitetura, bem como a sua capacidade de computação, levaram investigadores a usar Graphics Processing Unit (GPU)s para processamento de âmbito geral.

Apesar destas arquiteturas paralelas permitirem, teoricamente, uma performance bastante alta, em contrapartida também impõe desafios de programação complexos, especialmente no que toca a sincronizar acessos a dados partilhados. Como tal, aplicações que extraem o paralelismo inerente de sistemas heterógenos com GPU e CPU são particularmente difíceis de desenvolver.

Neste contexto, Memória Transacional (TM) foi proposta para facilitar programação paralela. TM é uma abstração, que procura esconder a complexidade de sincronismo de dados partilhados em sistemas com múltiplas threads . Na área de TM, a literatura é prolifica, existindo vários modelos para **cpus!** (**cpus!**)s, e recentemente para GPU.

Esta dissertação propõe Memória Transacional Heterógena (HerosTM), o primeiro (do conhecimento do autor) sistema de Transactional Memory (TM) capaz de escalonar threads simultaneamente no Central Processing Unit (CPU) e no GPU. Para minimizar o impacto na performance, Heterogeneous Transactional Memory (HerosTM) usa um modelo especulativo, com verificação de erros para garantir correção dos resultados, que amortizam os custos de garantir consistência entre unidades. A arquitetura deste sistema é apresentada e o sistema é testado recorrendo a uma benchmark sintética e a uma aplicação real.

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
HeterosTM	Heterogeneous Transactional Memory
CM	Contetion Manager
CR	Conflict Resolution

1

Introduction

Contents

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, a wide body of research has investigated the idea of using 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 very popular 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 for synchronizing access to shared data among multiple threads. The usage of locks is notoriously challenging, as the programmer needs to keep in mind problems 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, on the one hand, 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 of dead/live locks on a much larger scale.

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. Despite these difficulties, as heterogeneous systems are ubiquitous, research on this area is very prolific [7].

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 hard-

ware unit. TM solutions, for both CPU and GPU, already exist. However these solutions do not support sharing of data among threads running on heterogeneous CPU/GPU systems. Thus, they fail to fully explore the computing capability of this class of computing systems.

1.2 Objectives

This dissertation aims to develop the first TM system capable of supporting concurrent execution of applications that exploit both CPUs and GPUs. The intended goal is to facilitate programming of these systems by abstracting the difficulties of data sharing over multiple, physically separated, units via the TM abstraction. Applications running this system are able to share computational power and access to the dataset, transparently, between the CPU and the GPU, through the running of concurrent transactions.

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 the author's knowledge, a still unexplored approach in the literature, which makes it interesting from a research perspective.

Pursuing this goal requires addressing the following key issues:

1. Support the concurrent execution of TM implementations for both CPU and GPU ensuring correctness via adequate conflict detection and resolution techniques, in face of possible inter-unit conflicts.

2. Enable the possibility of partitioning transactional workloads and scheduling them to the various units.

3. Minimize the introduced overhead, incurred by synchronizing execution among transactions executing on heterogeneous processing units, despite the large communication costs that notoriously affect such heterogeneous architectures, so as to achieve a higher throughput than a single unit solution.

4. Expose a simple Application Programming Interface (API), so as to preserve the ease of programming at the basis of the goals of the TM paradigm.

1.3 Contributions

The above goal were pursued by designing and implementing Heterogeneous Transactional Memory (HeterosTM), the first TM system capable of coordinating execution of transactions on both CPUs and GPUs.

HeterosTM fully handles scheduling transactions to the different units, exposing a simple API to programmers that allows them to use the familiar abstraction of atomic transaction, for code that is executed

on both CPU and GPU. In order to maximize scheduling efficiency, HeterosTM allows programmers to specify which processing unit should be preferentially used to execute each transaction.

In order to minimize synchronization overheads, HeterosTM relies on speculative techniques, which allow to effectively amortize the costs of enforcing consistency among transactions executing on the CPU and GPU(s), over (potentially large) batches of transactions.

This system uses a speculative model to run transactions concurrently on different units, were units work separately and see the whole memory as their, which may cause conflicts between units. These conflicts are checked lazily, at certain intervals, by comparing the reads and writes of the various units. When a conflict is detected the Contetion Manager (CM) is called to decide how to proceed, otherwise units exchange results and are ready to execute again.

HeterosTM fully handles scheduling transactions to the different units, and ensuring that results at the end of execution are correct. To use this system the user codes his transactions as well as their respective partitioning algorithms. To run his code concurrently, the user feeds the inputs to his transactions to the system, instead of calling them directly, which will decide when and where to run them.

HeterosTM was evaluated using both a synthetic benchmark, Bank, and a real application, MemcachedGPU [8]. Experimental results for Bank show that HeterosTM offers better performance than Nvidia's transparent transfers with Zero-copy, whilst the results for MemcahedGPU show that it can outperform single unit performance by about 50% .

1.4 Outline

This report is structured as follows: Chapter 2 analyses current research on TM and GPGPU. Chapter 3 presents the work produced to develop HeterosTM whilst Chapter 4 presents its experimental evaluation. Finally Chapter 5 presents some concluding remarks.

2

Related Work

Contents

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 by any of the threads, at any given time. Concurrent accesses to the same memory locations 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 [9].

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 the transactional operations fail, all others follow suit - and serializability - i.e., the results produced by a batch of transactions are can be produced by executing them serially, in a given order [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 he 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 hash table case it would simply require marking a set of 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 [10], 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 [11] 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 [12], TL2 [13], SwissTM [14], TinySTM [15] 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 [12], 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 [13] is a word/object based STM. It was one of the first STM to use the concept of time-based snapshotting. 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 [14] 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 Manager's goals (resolving conflicts) are lazier version of a scheduler's (preventing conflicts), which are analysed further ahead.

TinySTM [15] 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.

TinySTM 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

	Granularity	Conflict Detection	Writes	Synch Strategy
NoRec [12]	Word	Lazy	Deferred	Lock-Based
TL2 [13]	Word/Object	Lazy	Deferred	Lock-Based
SwissTM [14]	Word	Mixed	Deferred	Lock-Based
TinySTM [15]	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.

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 was not 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.

For its configurability, TinySTM was the chosen system for this work. It was configured to use in-place writes with encounter-time locking, and therefore, eager conflict detection. As for the proposed system the logging is a very important feature, it is important to mention that Tiny maintains read and write sets, although only for the lifetime of the transactions.

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 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 Herlihy's [6] 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 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].

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

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., percentage of expected conflicts), 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 bench-

marks, 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 not 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) [33]. 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 [34], 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 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 [35] to real-time video encoding [36]. 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 [37] 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 overhead, 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 these reasons, the following analysis will be focused on CUDA and Nvidia's architecture.

2.2.1 GPU Architecture Overview

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

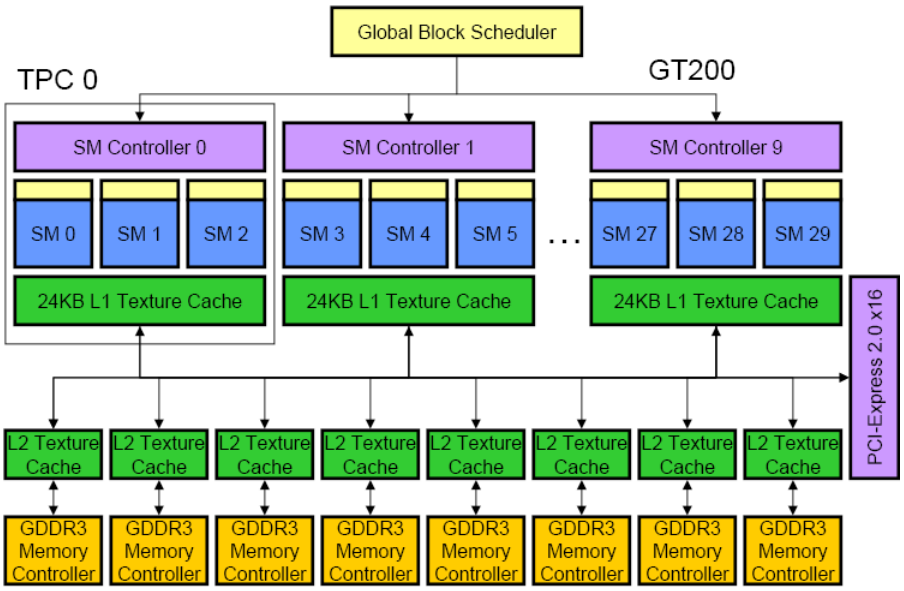


Figure 2.1: Architecture for a second generation Tesla GPU. Adapted from [1].

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.

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.

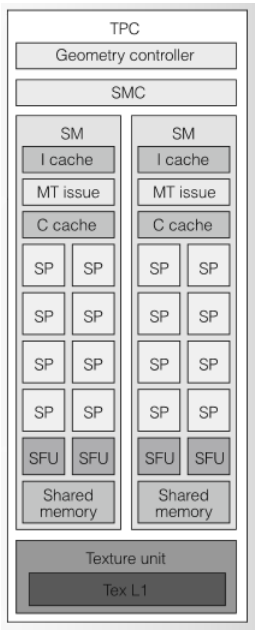


Figure 2.2: First generation Tesla Texture/Processor Cluster (TPC). Adapted from [2].

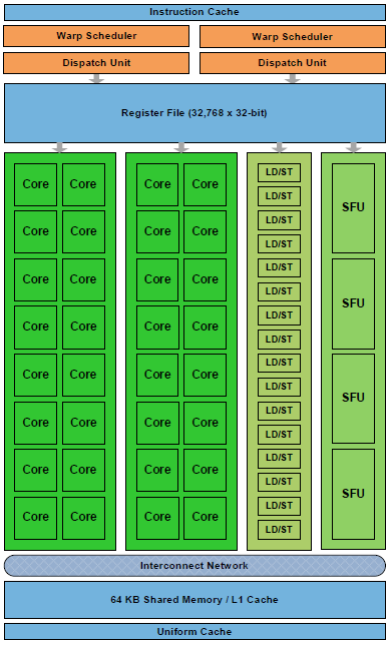


Figure 2.3: Fermi's Streaming Multiprocessor. Adapted from [3].

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 [38].

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 [39].

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 it is 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.

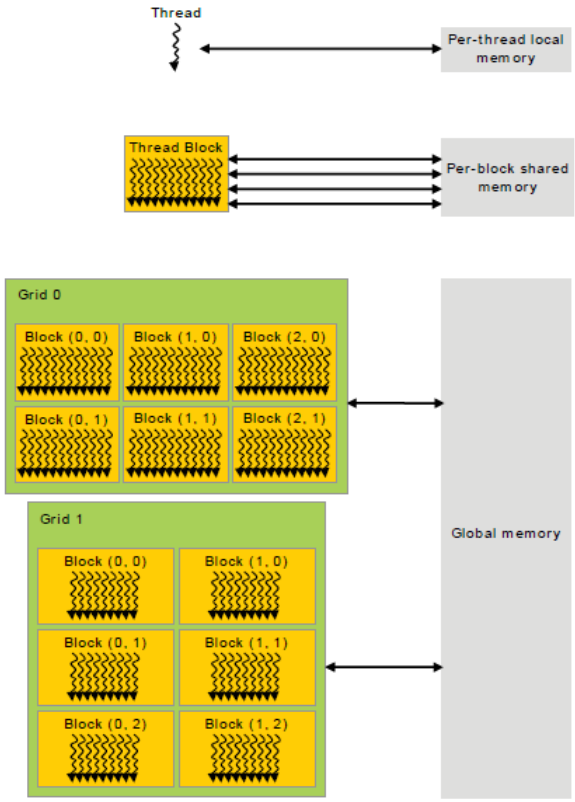


Figure 2.4: CUDA thread and memory organization. Adapted from [4].

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. Whilst this seems to imply that concurrent access is possible on Unified Memory, the reality is the data is moved on the background and cannot be accessed in one of the units while it is being accessed in the other. On the other hand, Zero-copy allows both systems to operate on the data concurrently, in the same way that is intended for the system presented in this work. However, race conditions may still happen between devices when using Zero-copy - 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.

As such, the TM abstraction 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 [40], 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.

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 [41]	Eager	Visible	In-place	Metadata Based	Software
PSTM [41]	Eager	Visible	In-place	Metadata Based	Software
ISTM [41]	Mixed	Invisible	In-place	Lock Based	Software
GPU-STM [42]	Mixed	Invisible	Defered	Lock Based	Software
PR-STM [43]	Mixed	Invisible	Defered	Lock Based	Software
KILO TM [40]	Lazy	Invisible	Defered	Value Based	Hardware

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.

The main difference between the GPU TM solutions and the CPU TM solutions is how they handle validation in a environment with very high thread counts and unordered memory accesses. This makes implementations more reliant on locking, however, the high thread count means that approaches such as encounter time-locking may be very costly. Therefore, most GPU TM rely on detecting conflicts only at validation time, using complex locking schemes to decrease both reliance on the unordered memory accesses, and thread divergence.

2.3.1 GPU-STM

GPU-STM [42] 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 as writes are typically unordered and atomic operation require global synchronization, and therefore have a very high cost. 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 in conflict detection, despite its larger overhead, it 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 an overlapping set of locks, they will do so following the same global order, one will do so and abort the other, which avoids the scenario of both acquiring different locks, and blocking whilst waiting for the other.

Altough GPU-STM provides some isolation, it does not ensure opacity, as transactions don not detect conflicts during execution, only doing so during the validation stage. 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. [41], 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 [43]. 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, as the highest priority conflicting thread gets to ”abort” the lower priority ones.

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 pre-locking each entry in its read and write sets. Each success leads to checking if the local version value matches the global one, for that memory position. When any of these two steps fails, the transactions is aborted. 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.

Otherwise, PR-STM executes transactions in a very similar fashion to CPU STM, with threads lazily detecting conflicts during validation. Like TinySTM, PR-STM keeps reads and write sets only for the lifetime of a transaction.

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 a 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 imple-

mentations 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

Heterogenous Transactional Memory

Contents

Building on the previous chapter's research, a new TM system, named Heterogeneous Transactional Memory (HeterosTM), is presented in this chapter. The goal of this system is to be able to transparently schedule a transactional workload across CPU and GPU, maintaining correctness, and achieving a superior performance to a CPU-only or GPU-only solution. All of this, whilst using a programming interface that abstracts the inter-unit synchronization problems away from the programmer, hence greatly reducing the complexity of development.

Whilst TM systems are already used to abstract the difficulties of synchronising multiple threads within a single processing unit, to the best of the author's knowledge, no solution exists that extends the convenient TM abstraction beyond the boundaries of CPUs or GPUs. This lack of heterogeneous solutions is due to the multitude of new challenges presented by heterogeneity, notably the high latency in inter-unit communication and the different programming models of CPU-based and GPU-based applications.

Nonetheless, with the growing interest in GPGPU research, as well as the emergence of new hardware and software solutions, that minimise these problems, the interest in applications that make effective use both CPU and GPU grows [7], [44], [45].

In this chapter, the architecture of the designed HeterosTM system is detailed, as well as the system's programming interface and expected usage.

3.1 Architecture Overview

This section presents an overview of HeterosTM's architecture and execution model. A large variety of heterogeneous systems exist, however, for the purposes of this work the system's considered were those comprised of a singe CPU and a single dedicated, consumer-grade GPU. Nonetheless, HeterosTM's design was extended to support multiple-GPUs. To start, the basic, single GPU/CPU design is introduced. Its execution model, for a single GPU/CPU system is represented on the flowchart in Fig. 3.1. Furthermore, the single CPU limitation refers to a single instance of the CPU TM, not to a hardware limitation.

When using HeterosTM, the programmer is responsible only for providing the transactional code and inserting inputs into the system. In turn, the system is responsible for scheduling transactions across the different processing units and ensuring correctness. To improve performance, HeterosTM allows programmers to provide hints on which computational unit (CPU vs GPU) should be used to process transactions.

Clearly, the likelihood of succeeding in successfully executing concurrent transactions on heterogeneous units can be strongly affected by the effectiveness of the transactional partitioning scheme. An effective partitioning scheme would strive to schedule transactions so as to minimize inter unit conflicts, whose detection is way more expensive than for transactions executing on the same unit. To allow for partitioning, HeterosTM uses a producer/consumer execution model for its transactions, with the user producing transactions by inserting transactional inputs in to the system.

HeterosTM currently adopts a simple hint-based approach, which allows programmers to exploit ap-

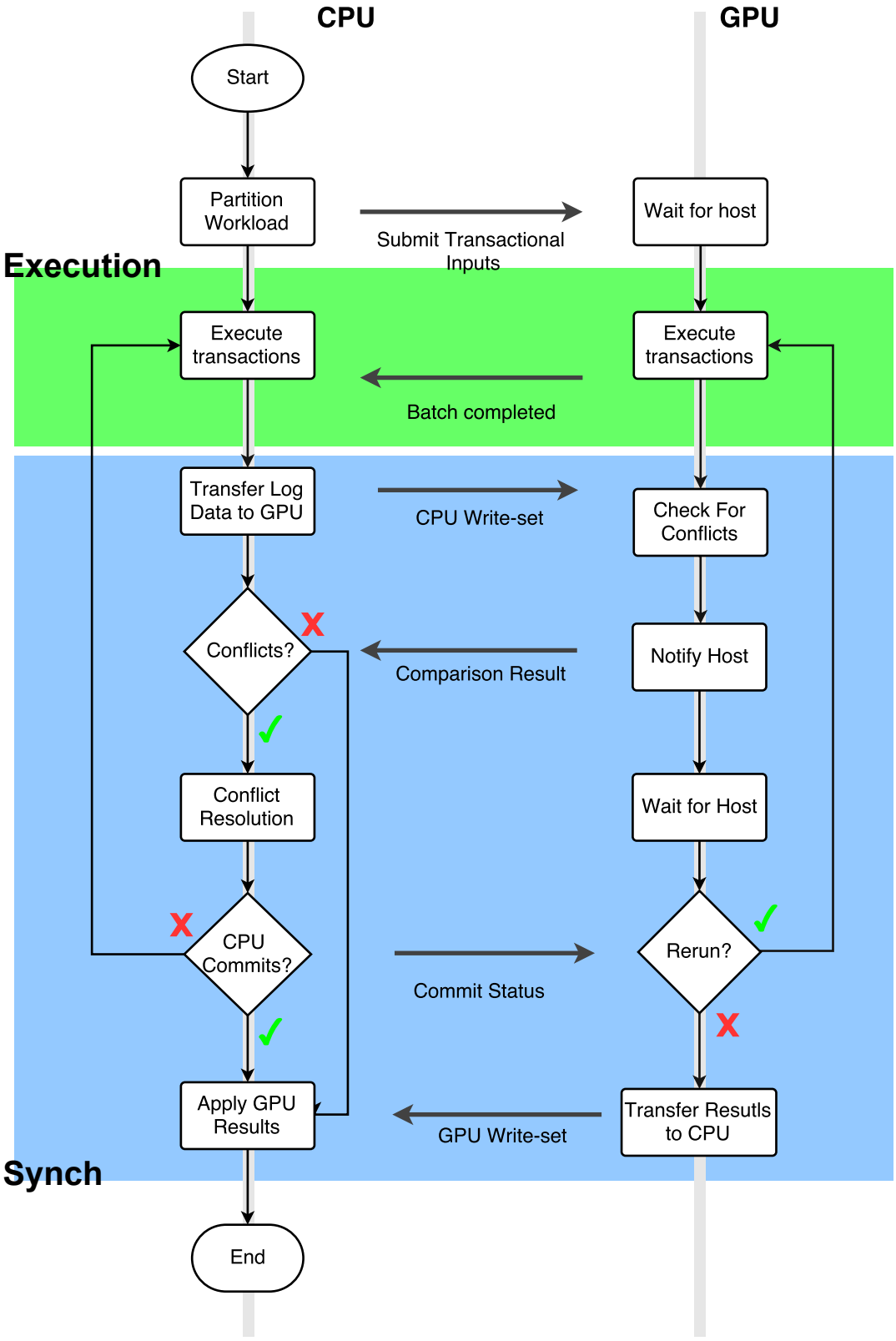


Figure 3.1: Execution flowchart for HeterosTM.

plication/domain knowledge to take informed decisions on which computational unit to use for executing their transaction. Yet, the programming interface of HeterosTM is flexible enough to allow embedding more sophisticated scheduling techniques that may decide to further optimize the programmer sug-

gested initial policy using, e.g., machine-learning technique [29].

HeterosTM's execution model involves switching between two states: Execution and Synchronization. The system alternates between the two, either running a transactional workload in the Execution state, or running system error checking, in the Synchronization state.

Synchronization amongst threads running on the CPU and threads running on the GPU occurs in a lazy fashion, i.e., periodically or when the set of transactions to be executed on the CPU/GPU is completed. Once synchronization triggers the HeterosTM system uses transaction logs, which contain the cumulative CPU write-sets, to compare the memory accesses amongst units to ensure no inter-unit conflict has occurred. Comparison is done by searching for intersections between a unit's write set, and the others read-sets, which serializes the former, which is the CPU in this system, before the latter.

One noteworthy aspect of this design is that it only requires units to exchange the write-sets of the transactions they processed. Read-sets are, conversely, never communicated among processing units. As transactions' read-sets are typically much larger [46], [33] than write-sets in common TM workloads, this allows for a significant reduction in the amount of data to be transferred - which is crucial to maximize the efficiency of the system.

In case of successful synchronization, with no conflicts detected, both units' datasets are updated with each other's produced results. When any conflict is detected among the set of transactions executed on the CPU and the GPU(s), HeterosTM decides which batch of transactions to commit, causing the other unit to rerun, drop its results and overriding its dataset with the other unit's. This synchronization scheme essentially creates a two-step commit phase for every transaction, with the first commit performed locally within the unit and the second commit performed after inter-unit synchronization.

The system's initial design goal was to use the GPU as an accelerator, with the CPU's copy of the data being the authoritative copy and the GPU's copy always serialized after the CPU's. Based on this vision, only the CPU's commits are preserved in case of conflict. This design decision was motivated by the CUDA programming model's inherent Master-Slave architecture, as well as the CPU's capability to expose results to the user/outside world, in which case late aborts due to unit synchronization are undesirable.

However, when transactional workloads are highly parallel, the GPU's performance can dwarf the CPU's. In these scenarios, GPUs can commit a much larger number of transactions than CPUs during a given time window/synchronization interval. Hence, always giving priority to the CPU in case of inter-unit conflicts, may be largely suboptimal performance wise, since it may lead to abort a larger batch of transactions in favour of a smaller one. Further, it may expose the transactions scheduled for processing GPU to the risk of starvation. To cope with this issue, HeterosTM supports different conflict resolution policies, which allow optimizing the system's behaviour to maximize different objective functions (e.g., throughput vs fairness).

As mentioned, the focus of this dissertation is on investigating and evaluating heterogeneous CPU-GPU systems hosting a single GPU. Yet, the base design of HeterosTM has been extended also to support concurrent execution on a set of GPU(s). The algorithm used in HeterosTM to support concurrent execution of multiple GPUs is illustrated in Fig. 3.2.

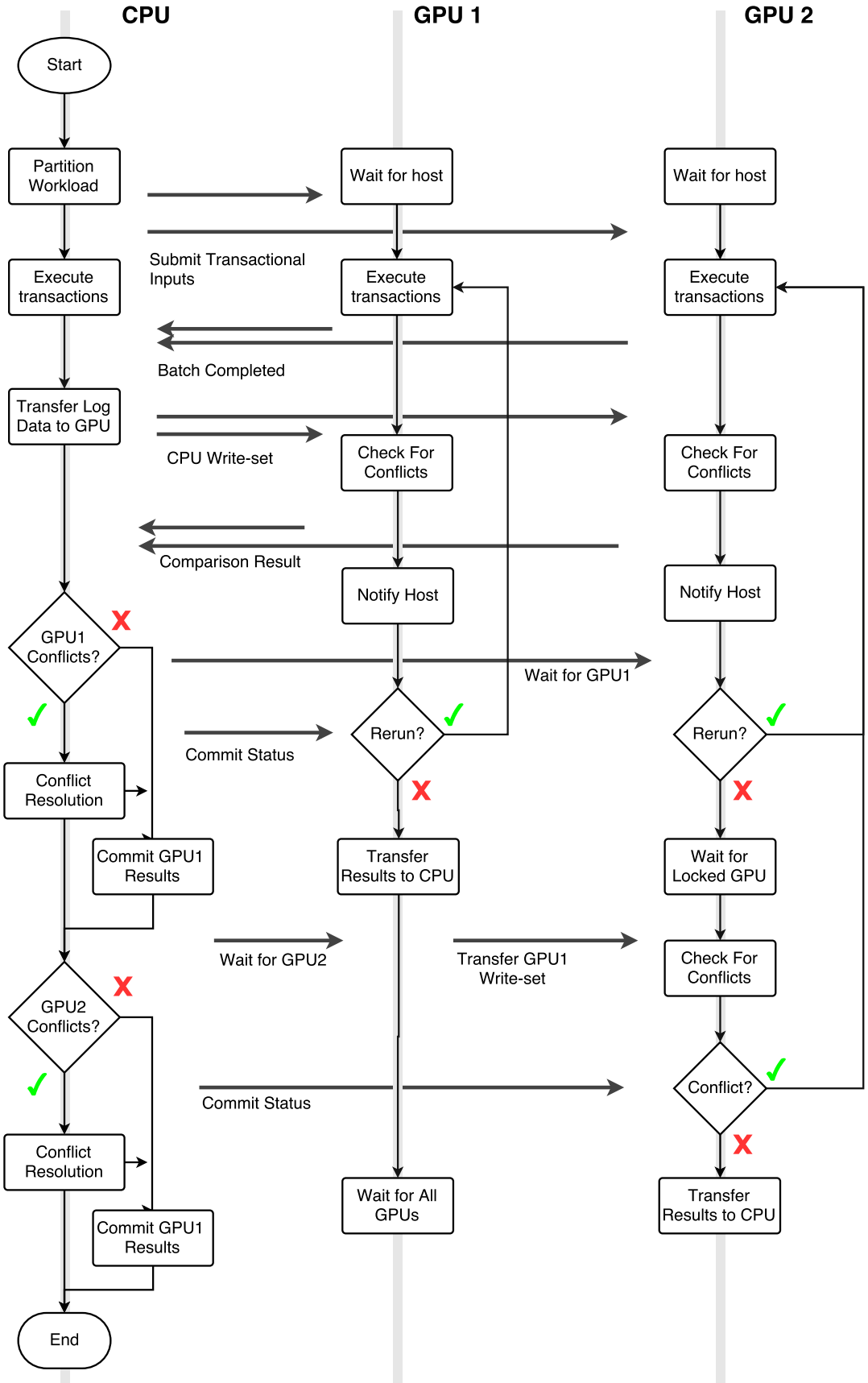


Figure 3.2: Multi-GPU execution flowchart.

When using multiple GPUs,each GPU receives its own batch of transactions, which are launched concurrently. When the units conclude execution of all the transactions in their batches, they notify the CPU, which starts to send its write-set for conflict detection. Each GPU does this comparison, so that conflicts can be detected with unit granularity and the system can preserve specific units' results.

If comparison with CPU's logs succeeds, the units lock access to the CPU temporarily, to commit their write sets in sequence. The GPU commit order is established dynamically, according to which GPU finishes its comparison with the CPU first. For instance in Fig. 3.2 GPU1 finishes the comparison first and is allowed to commit. On the other hand GPU2 needs, to wait to commit its results to GPU, and, as other GPU units have already committed, it must also validate is read set with all of the preceding GPUs' write-sets, which in this case is only GPU1.

As mentioned at the beginning of this section, when using HeterosTM the system is fully in charge of scheduling transactions. When programmers submit transactions for execution in the HeterosTM system, these are dispatched towards a set of queues, from where the processing units fetch their inputs. HeterosTM maintains a set of queues per each processing unit queues, from where it fetches its transactions, as well as shared queue for non-attributed transactions.

For each transaction, the queue stores, the inputs, the results produced, and if they have already been run successfully or not. Ideally the queues' transactions for each unit are partitioned perfectly from the others' queues, however transactions may be moved to another unit's queue for load balancing amongst units.

If the programmer uses multiple types of transactions, the system instead keeps separate, per-unit queues for each of the transaction types, fetching from these in a round-robin manner. Using multiple queues consumes a lot of system resources, however, separate unit queues facilitate batching transactions which is the ideal use case for the GPU. Furthermore, although this is application dependent, separating queues into multiple types of transactions allows the programmer to batch together similar operations, for even lower thread divergence.

The key challenges faced when developing the HeterosTM system are the following:

1. Introducing logging support for the CPU/GPU TMs used as the base for this system, in an efficient fashion, enabling the HeterosTM system to detect conflicts between the different units.

2. Developing comparison algorithms capable of detecting conflicts between the different units, whilst adding the smallest possible overhead.

3. Partitioning and scheduling transactions amongst CPU and GPU, whilst avoiding conflicts with a large degree of confidence.

4. Resolving conflicts amongst units, both ensuring correctness and minimising impact of conflicts on system performance.

In the following sections of this chapter the solutions chosen for these challenges are presented, as well as the usage of the HeterosTM API.

3.2 Programming Interface

To use HeterosTM the programmer needs to first define the transactional inputs, and then the transactional functions and kernels to be ran. HeterosTM's API is composed by 4 core primitives:

int REGISTER_TRANSACTION(int $txtype_{id}$, function * $funcCPU$, function * $funcGPU$, size_t $outputSize$, size_t $inputSize$): This function is used to register a transactional code block in the HeterosTM system. The programmer has to provide two pointers to the functions to be executed, depending on whether the transaction is executed on the CPU or on the GPU. For the case of the GPU, the function pointer is a wrapper for a CUDA Kernel. This primitive also requires a unique id, $txtype_{id}$, as an input to identify each CPU/GPU transaction pair, as well as the size of the inputs and outputs of these transactions. Returns 1 on success, 0 otherwise.

void START(int $memorySize$, int $CPUthreads$, int $GPUthreads$, int $GPUcount$): Called to start the execution of the HeterosTM system. The arguments for this primitive are the the thread counts for the various units, the number of GPUs to use and the maximum size of the memory to be shared by the CPU and GPU. This primitive is also responsible for allocating the datasets' memory, which is why it requires its size as input, as well allocation and initialization of the queues for the registered transactions.

int SUBMIT(int $txtype_{id}$, void * $txInput$, int $unit_{id}\_hint$): Used to request the execution of a transaction of type $txtype_id$, and to specify its input arguments (if any), as well as the user provided hint, of which computational unit to use to execute the transaction. Units are identified numerically, with the CPU always being zero, and the GPU(s) using their CUDA device ID incremented by one. The hint may also be ANY, placing the transaction in the shared queue. This call returns the unique ID of the scheduled transaction, which can be used to retrieve the result produced by its execution.

tx_output GET_RESULT(int tx_{id}): This primitive is used to fetch the produced results, if any exist. As there no time guarantees for when a transaction is ran, a non blocking variant of this primitive also exists, which can be used to check whether the transaction has already been executed without blocking the caller thread.

When using HeterosTM, these primitives are essentially called in order. The programmer inserts his transactions into the system with REGISTER_TRANSACTION() primitive, and then starts execution by calling START(). START() launches the CPU threads, initializes the GPU(s) with the data needed for execution and builds the necessary queues.

When the START() has been successfully called, the user is free to submit transactions via SUBMIT(), however he is no longer capable of registering new transactions. Finally, to retrieve these transactions' return values the user call GET_RESULT().

As mentioned, HeterosTM's execution model involves switching between two states: Execution and Synchronization.

The Synchronization state is triggered when all GPU batches are concluded, as when a kernel is launched there is no efficient way to kill it. Therefore, the conclusion of a GPU kernel is the best time to

check for illegal accesses in both units, which means the size of the GPU batches defines the frequency of synchronization.

Every time a new transaction is submitted, the system dispatches it to the computational unit specified by the programmer-provided hint. This helps mitigating inter-unit contention, but may lead to load unbalancing scenarios, where one unit keeps getting all the inputs while others starve.

To cope with this issue, in case all the queues of a unit are empty, the unit can consume transactions from the shared queue, or from the queues of other units.

Its worth noting that CUDA's limitations present a challenge when using dynamic memory allocation. To implement similar functionality, HeterosTM provides its own memory allocator. This allocator is limited to only allocating inside the memory region established as the dataset when first calling the START() primitive.

3.3 Integrating TinySTM and PR-STM in HeterosTM

To implement TM for each of the units, HeterosTM includes two state-of-the-art STM implementations: TinySTM for the CPU, and PR-STM for the GPU. In order to support HeterosTM speculative execution model, both Tiny and PR STM had to undergo some lightweight and unobtrusive modifications, which aimed at persisting the read-sets and write-sets of transactions beyond their normal lifespan.

Indeed, like most TM systems, both Tiny and PR maintain read and write sets of transaction for commit time validation. However, these logs are only temporary and are discarded after the transaction's completion.

In the proposed system, though, these logs must be kept until a inter-unit synchronization is triggered, due to the need of detecting conflicts developed among transactions executing in different computational units.

Clearly, it is desirable that the logging functionality has minimum impact on the base STM performance. Further, the logging system should be designed so to allow for efficient implementations of the validation phase and efficient usage of system resources.

A key design choice of HeterosTM aimed to maximize validation's efficiency is to require exclusively exchanging the transactions' write-sets, which are up to several order of magnitudes smaller than tx write-sets in popular benchmarks [46]. In fact, HeterosTM attempts to serialize the GPU transaction batch, after the CPU's. To verify whether this is possible, one needs to confirm that none of the memory regions read by any of the transactions processed by a GPU was written by a CPU transaction. This validation is implemented in HeterosTM by having the CPU sending the write-sets of its batch to the GPU, which compares them with the (locally available) read-sets of the transactions it processed.

As the logs are no longer necessary after synchronization, this allows the HeterosTM to free up logs after validation, so as to more efficiently use system memory.

3.3.1 Changes to TinySTM

The CPU is always synchronized before the GPU, therefore, it only needs to save its own write-set. The focus of development of the CPU TM's logging system was minimising the logging system's impact on transactional performance.

A first decision was to maintain distinct per CPU thread logs, as the system only needs to know which memory positions the CPU accessed and therefore its log can be kept out of order and split. This removes the need for inter-thread synchronization upon successful commits, meaning the only overhead incurred is performing a copy of the write-sets from TinySTM's local data structures - which are recycled when the transaction terminates - to HeterosTM's in-memory logs.

In order to reduce the frequency of dynamic memory allocation, as it is very costly in terms of performance, HeterosTM preallocates large memory chunks with a capacity of 64K log entries. Memory chunks are organized into a linked list: whenever a memory chunk is full it is appended to the linked list, a new chunk is allocated. This design also has the advantage of facilitating log transfers between units.

With this base design two variations of a CPU logging systems were developed:

- Address Log:In this case, each entry of the log stores only the memory address of the position accessed. This saves the minimum amount of information possible for a comparison, and therefore has the smallest possible footprint, facilitating its transfer to the GPU. On the downside, it provides no information to the GPU on the values which were produced by the writes issued by transactions committed by the CPU. Hence, as we will discuss in Section 3.4, the usage of this log requires the use of an additional phase to transfer the memory regions updated by the CPU towards the GPU, in case the inter-unit synchronization is successful

- Versioned Log: In this case, beside address, the log also stores the value written and the current value of TynySTM's global timestamps (a scalar clock used by TinySTM to establish the transaction serialization order). Compared to the Address log, the Versioned log records a larger amount of information. However, as we will discuss in Section 3.4 it enables the use of efficient validation mechanisms that allow CPU to keep on processing transactions, in a non-blocking fashion, even while inter-unit validation is in progress.

In Section 4.1.1 these two implementations are profiled to find out the workloads for which they provide the best results.

3.3.2 Changes to PR-STM

The GPU TM also requires modifications, with the goal of storing transactional logs.

An important consideration when developing for the GPU is that dynamic memory allocation inside a kernel has a huge negative impact on performance, and is considered a bad practice. Therefore all GPU logs should be designed using some form of statically allocated log.

On the GPU side, HeterosTM gathers logs for the read-sets and write-sets of committed transactions, supporting two different approaches:

- Explicit log: A per-thread explicit log of all the memory positions accessed. Threads story only the address of the accessed memory positions, in the GPU's global memory. Logs are kept per-thread, and are stored in warp-sized contiguous addresses to allow for coalesced memory accesses.

- Compressed log: A global data structure, functioning as bitmap of the accessed memory positions. This log's structure uses bytes, instead of bits like in regular bitmaps, so that it avoids reading it to update it. An access is always signalled by setting an entry to 1, so to add a new entry the system only needs to write to the correct position with no need for locking. The logs for read-sets and write-sets are layed out in memory as two contiguous memory words, which indicate if that address was read or written, respectively.

For both logging systems, the read-set is added to the log when PR-STM successfully validates a transaction, whilst the write-set is saved when committing the results. The logs for read-sets and write-sets are layed out in memory in an interleaved fashion. This design is motivated by the consideration that some memory locality is expected to exist for these two sets, which will improve performance by using caching and coalesced accesses more efficiently.

Unlike the CPU TM's case, here the two log implementations are quite different. The Explicit Log allows for a very low computational overhead, at the cost of memory space, since all this algorithm does is copying the accessed addresses to global memory. However, with the large number of threads in the GPU this logging implementation's scaling becomes pretty poor as it can require enormous amounts of memory, and an even greater number of operations whilst comparing, as each entry in this log needs to be compared with each entry in the log its comparing to. Further more this design is not very efficient, as it may store several duplicate entries if it accesses the same positions multiple times.

The Compressed log on the other hand, is designed for very fast comparisons, and has great scaling both in terms of memory and number of threads. Whilst ideally each entry in its bitmap equates to a memory position, a decrease in granularity can be beneficial as it allows for a much smaller memory footprint, at the cost of the possibility of false-positives in conflict detection, but maintaining no possibility of false-negatives. In HeterosTM, we chose to use PR-STM's hashing algorithm to map the addresses as relative instead of absolute, which facilitates comparison with the CPU's logs. Using this means that granularity in the bitmap is tied to the one used for the locks, which is a concession to avoid the usage of an extra set of locks.

Despite the Explicit Log presenting a good case for scenarios where the logs are expected to be small - long transactions with few transactional accesses - these sort of workload is not favourable for the GPU, which is why for most of the tests presented in Chapter 4 the Compressed Log is the one used. In Section 4.1.2 these two implementations are tested and compared.

3.4 Synchronization

Synchronization is one of the two systems states for HeterosTM, where the transaction batches processed by CPU and GPU are validated in order to detect whether they developed any conflict or not. The former case requires aborting one of the batches. In the latter case, instead, the write-sets produced by the two computation units must be merged in order to ensure that both units reach a common state. Fig. 3.3 presents the base HeterosTM Synchronization algorithm.

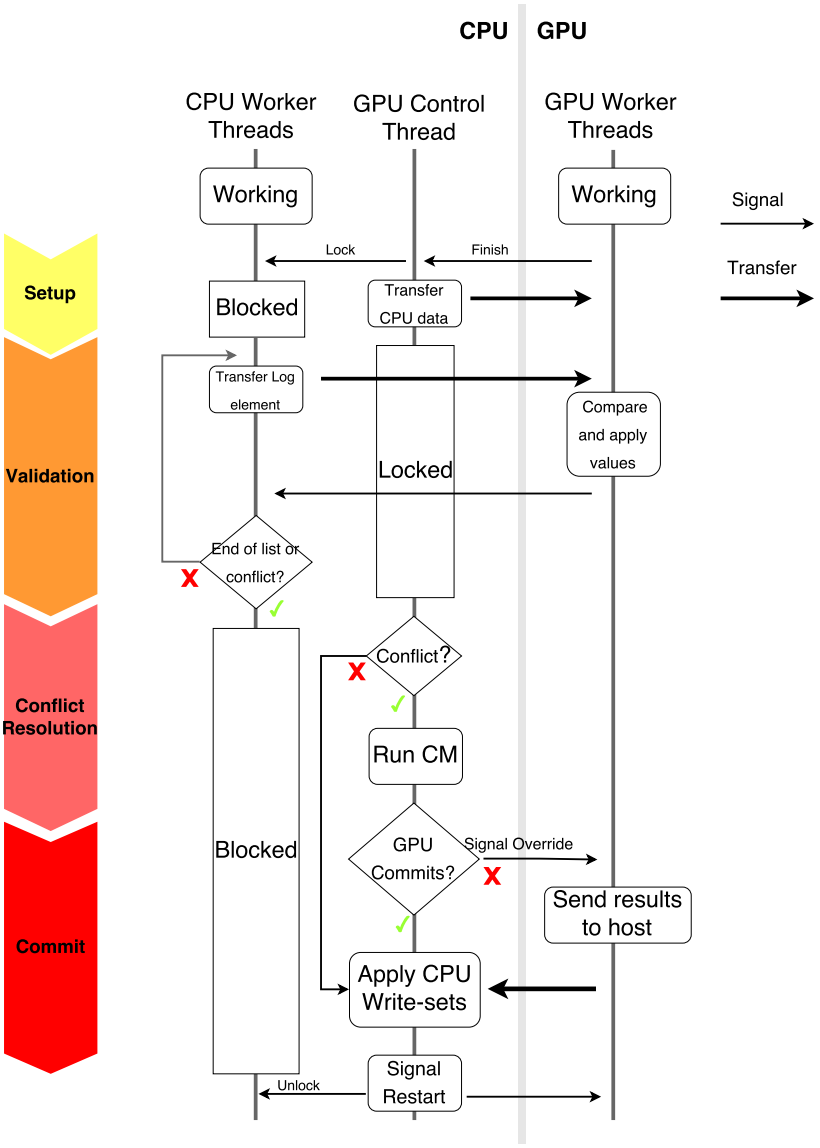


Figure 3.3: Base HeterosTM synchronization fluxogram.

As pictured in Fig. 3.3 this state is split into 4 consecutive phases. Setup, is the first phase and is used to lock the CPU and prepare for the log comparison. The Validation phase happens next. During this phase the CPU sends its logs to the GPU, which compares them with its own logs, to detect if the CPU's writes intersect with its reads, transferring back the outcome of the validations to the CPU. The Validation phase ends when either all comparisons are finished or one of them detects a conflict, and the system enters the Resolution phase. When a conflict is detected this phase is used to decide on

which unit should get priority, otherwise when no conflict is detected both units' transactional batches can be committed, and have to be applied to both units. When the Resolution phase finishes, the units chosen to do inter-unit commits does so during the Commit phase.

Logs are compared on the GPU, as this is a fully parallel workload. Since the comparison is also a workload that does not need to be executed in order, it also allows the usage of CUDA Streams [4], which do not ensure any sort of ordering, to run overlapping comparison kernels.

Since HeterosTM's CPU logs are per-thread, each thread is responsible for launching comparison with its logs on the GPU, when notified by the CPU's control thread. To achieve this HeterosTM uses Streams, allowing CPU threads to launch kernels and transfers in parallel. To guarantee that the system is oversubscribing the GPU and the PCI-Express transfers, each CPU thread queues multiple operations in different Streams. Using this approach, the GPU can run multiple comparisons kernels in parallel and overlap them with the memory transfers. Further overlapping could be achieved on high-performance GPUs which support concurrent bi-directional transfers, however, in the interest of making HeterosTM universal, this is not considered.

To avoid excessive non-log related memory transfers between units, the Validation phase is also used to update the GPU with the CPU's write set. Since the CPU's log only keeps its writes, any address present in the log is an address to be overridden with value of the CPU's dataset. In the ideal scenario (no inter-unit conflicts) this means that at the end of the Validation phase the GPU already has its memory image fully updated with the CPU's results. For the Commit phase, the system simply overrides the now outdated CPU dataset with the more recent CPU plus GPU data, which is present on the GPU.

Since using the same type of fine-grained memory transfers to update the CPU's dataset with the GPU's produced results is not efficient in the CUDA programming model, as it requires dynamically allocated memory, HeterosTM divides its dataset into chunks of 128KB and transfers only the updated chunks. To do so, HeterosTM uses the GPU's write-set to build a map of the regions that were touched by the GPU, and transfers only those regions to CPU during the Commit phase.

When using the Address log, the approach of comparing and applying requires that the CPU's dataset is transferred to the GPU before Validation, during the Setup phase. When comparing the CPU has to stop transaction processing as that would cause the dataset transferred to the GPU to be outdated. Addresses contained in the log are checked with the GPU's log, and then updated in the GPU's dataset with the previously sent CPU copy. The algorithms used to run the log comparison are presented in Section 3.4.1.

Transferring the whole dataset is very costly, and scales poorly with increased dataset size. This can be avoided by using the Versioned Log as it already contains all the necessary information to run comparison and apply the write sets. The Versioned Log does not require the initial memory image transfer, as already mentioned in Section 4.1.1, provide the additional advantage of enabling a, so called, Non-Blocking Synchronization. This alternate Synchronization algorithm only blocks the CPU from producing new results at the end of the Validation phase, as opposed to the start of Setup.

When using Non-Blocking Synchronization, the CPU's threads queue batches of comparisons until all

of their streams are occupied. Threads periodically check their streams to see if they've all concluded, in which case they queue up a new batch of comparisons, otherwise they continue executing transactions. This Synchronization algorithm requires a workload where the comparison of a set of logs is fast enough that the CPU cannot produce a similarly sized set of logs in the same time interval - which has always been the case for all the workloads we tested, as log comparison can be parallelized very efficiently using GPUs. When the number of elements remaining in the log reaches a certain, user defined, minimum the system blocks and enters the final Synchronization Phases.

If a conflict between units is detected, the Conflict Resolution phase is used to decide on which unit should be allowed to commit. HeterosTM implements three different Conflict Resolution techniques, which are presented in Section 3.4.2. The unit which maintains its results after the Conflict Resolution overrides the others dataset with its own during the Commit phase, therefore deleting results produced by the other.

3.4.1 Validation

The validation is the most resource intensive phase of Synchronization, as it is comprised multiple memory transfers, both to and from the GPU. It is also very important for HeterosTM that conflicts are properly detected, so that it can ensure opacity and correctness. Bellow are presented the algorithms used by HeterosTM for the Validation phase, starting with the CPU's.

CPU Side

As previously pictured in Fig. 3.3, HeterosTM uses two types of CPU threads. The Worker Threads are the ones executing the transactional workload. On the other hand, the Control Thread is the one responsible for interfacing between the GPU and CPU Worker Threads. In Listing 3.1 is the pseudo-code for Blocking Synchronization.

The Worker threads also play a key role in the Validation, as they are the ones that queue the transactions. Due to the CPU's logs design as a linked list of arrays, the comparisons are partitioned into these same arrays. As each Worker Thread only has access to small quantity of GPU memory, threads only launch a certain number of comparisons at each time, referred to as comparison streams.

When the Synchronization phase is triggered, the CPU Worker threads block, while the Control Thread is carrying out the memory transfers in the Setup phase. Threads are blocked using barriers, to ensure that no progress happens until all threads lock. After the set-up HeterosTM enters the Validation phase. During this phase the Control Thread is idle, whilst the Worker Threads are queueing comparisons.

Worker Threads queue comparison streams, and then wait until either they all conclude or one of them detects an error. CUDA's callbacks are used to execute a function, which notify the threads when the attached comparison kernel ends. Upon error detection, or full log comparison conclusion, the Worker Threads block, whilst the Controller Thread runs through the remaining phases for synchronization.

Listing 3.1: Blocking Comparison Queuing Algorithm.

```
function comparison_Blocking() {

    barrier_cross();  //Lock for Setup phase

    logPointer = fetch_log();

    barrier_cross();  //Signal start of comparisson

    /*Run till the end of the log or until a failed comparison*/
    while(logAux != NULL && compFlag == 0 ) {
      count = 0;

      /*Launch a batch of comparisons*/
      for( n=0; n<nb_Streams && logPointer!=NULL; n++ ) {

        launch_comparison(logPointer);

        logPointer = logPointer->next;
      }

      /*Wait till the batch has finished processing*/
      while(comparison_finished_counter < n);

      /*Check if this threads comparisons detected a
      conflict and notify other threads of it*/
      if ( error_Comparison() )
        compFlag = 1;
    }

    barrier_cross();        //Signal end of comparisson
  }
}
```

On the other hand Non-Blocking Synchronization, presented in Listing 3.2, only blocks the worker threads at the end of validation phase, i.e., more precisely during the validation of the last comparison stream. In this algorithm after queueing a comparison stream, the Worker Threads return to executing transactions, instead of idling while waiting for the GPU to finish the batch.

When using the Non-Blocking algorithm, the Worker Threads use three different Validation states, so that they can both run transactions and queue comparisons. During the Read state, which is the initial state, the thread fetches a new log, and checks its size. If the size is equal to a threshold (for instance 1 comparison stream) the thread moves to the Lock state, and runs comparison as shown in 3.1. Otherwise, the thread moves into the Stream state, where it queues comparison streams and runs transactions. When the queued batch of comparisons finishes the thread queues another one, unless all the fetched logs have been compared, in which case the thread returns to the Read State.

GPU Side

Log comparison is an ideal workload for the GPU's architecture, since it is comprised of a large amount of tasks, which may be executed in any order. The design of HeterosTM, with logs being inherently partitioned both by being per thread and by being a linked list, also favours the usage of Streams, as comparison can be done in any order, unaffected by the GPU's weak ordering.

Comparison for the Address log is very straight forward. The kernel, whose pseudo-code is pre-

Listing 3.2: Pseudo-code for Non-Blocking Comparison Queuing.

```
function comparison_nonBlocking() {

  /*Fetch elemtents from the log*/
  if(state == READ) {
    logPointer = fetch_log();

    comparison_finished_counter = 0;
    n = 0;

    if(nb_element(logPointer) < LOCK_THRESHOLD)
      state = LOCK
    else
      state = STREAM;
  }

  /*Non-blocking comparison queuing*/
  if(state = STREAM) {
    /*Exit if the last batch has not finished*/
    if(comparison_finished_counter != n)
      return;

    /*Check if the last batch failed*/
    if ( error_Comparison() ) {
      compFlag = 1;
      state = LOCK;

    } else {

      /*If the log is now empty go back to reading
      otherwise queue a new batch of transactions*/
      if(nb_element(logPointer) == 0) {
        state = READ;
      } else {

        /*Launch a batch of comparisons*/
        for( n=0; n<nb_Streams && logPointer!=NULL; n++ ) {

          launch_comparison(logPointer);

          logPointer = logPointer->next;
        }
      }
    }
  }

  /*Lock to finish synchronization*/
  if(state = LOCK) {
    comparison_Blocking();
    state = READ;
  }
}
```

Listing 3.3: Code for Address Log comparison.

```
__global__  void checkAdressLogKernel(
  int * flag_conflict,     /*Flag for error detection*/
  long * host_log,         /*Host log*/
  int * device_Log,        /*GPU read log */
  int size_host_log,       /*Size of the device log*/
  long * device_data,      /*Device Dataset*/
  long * host_data)        /*Host Dataset*/
{
  int id = blockIdx.x*blockDim.x+threadIdx.x;
  int address;

  /*The conflict flag is shared by all kernels in the same batch
  so it can be used to notify newly starting kernels of conflicts*/
  if(*flag_conflict == 0) {

    if(id < size_host_log) {
      address = host_log[id];

      /*Check the same adress in the GPU log,
      if is marked the kernel has detected conflict,
      otherwise update the GPUs dataset*/
      if(device_Log[address] == 1)
        *flag_conflict=1;
      else
        device_data[address] = host_data[address];
    }
  }
}
```

sented in Listing 3.3, is launched with a number of threads equal to the size of the log it is comparing. Each thread reads from one element of the CPU log, and then searches the GPU's log for that very same memory address. If it detects an conflict, the unit sets a flag in global memory to 1. All kernels check the flag in global memory at the start of execution, to detect if one of the other Streams that ran before it have detected an error.

The kernel for the Versioned Log, is fairly similar, with the added version checking, which was added to allow storing the log entries to the GPU dataset. In general, logs can be expected to contain duplicate entries for the same memory position, corresponding to updates serialized in different order by the Tiny. The version checking is what allows the Versioned Log comparison to be executed out-of-order while still ensuring that by the end of the execution we have the most updated values applied to the GPU's dataset.

Since updating a memory position and checking its version are atomic operations, this comparison kernel uses the GPU TM's lock array to lock the memory positions when reading the current version, and when updating both its value and its version. Therefore, the comparison overhead for this specific algorithm is slightly higher than Address log's comparison kernel. In situations where the dataset is small, the overhead increases as the probability of threads fighting for lock increases. However, when the dataset's size increases significantly, fighting for locks becomes improbable, and the initial overhead of transferring the full data set to the GPU means that the Address Log gets worst whilst the Versioned Log gets better.

As each logging solution favours a certain workload, HeterosTM implements both, leaving it to the user to decide which is more appropriate for his application.

3.4.2 Conflict Resolution

When a conflict is detected the system must decide how to proceed, by selecting which units get to keep their commits and which will be overridden. HeterosTM implements three different Conflict Resolution (CR) policies:

- GPU Invalidation: In this configuration the CPU is seen as the master copy, and always keeps its commits no matter what. This may lead to GPU starvation when the workloads cause are prone to inter unit conflicts. However, it allows CPU commits to be externalized immediately (e.g., to external users or systems interacting with the application), without waiting for the completion of the synchronization phase with the GPU, as CPU commits are guaranteed to never be discarded.
- Favour Fastest CR: A throughput oriented CR. With this configuration, the system compares the commits produced be the units since the last Synchronization, and the unit with the most commits wins. This can still lead to starvation for the slowest unit, however for workloads we tested, it can still lead to better performance than than GPU invalidation.
- Balanced CR: Focused on avoiding starvation, this CR ensures that both the GPU and CPU both get to commit a roughly even number of batches of transactions.

Balanced is based on GPU Invalidation, in the sense that the CPU is assumed to be the master copy and that the GPU is always overridden on conflicts. To ensure that both units have a roughly similar amount of inter-unit commits, Balanced CR disables the CPU after a certain, programmer defined, number of failed inter-unit conflicts. The CPU remains disabled until the GPU commits a programmer defined portion of the transactions that lead to to locking the CPU. With the correct configuration Balanced avoids starvation for both units, however this approach leads to, in general, a hit in performance when compared to Favour Fastest CR.

3.5 Summary

This Chapter presents Heterogeneous Transactional Memory (HeterosTM), a new abstraction, based on TM, that facilitates programming for heterogeneous systems by allowing concurrent access to the same dataset by various units. When using HeterosTM, users give it transactional inputs and the system is in charge of scheduling them to run in the correct unit, and then executing it.

To use HeterosTM, the user first defines his transactional code for both CPU and GPU, the and their inputs and outputs. After setting this up and initializing the system, the user submits transaction inputs into the system, along with a hint of what is the bes unit to run them: The system places the transaction on queue taking both this hint and other factors into account, to schedule this transactions in the appropriate processing units.

To achieve concurrent access amongst units HeterosTM uses a speculative model, were certain memory regions are assigned for each unit to work with. Units may occasionally be assigned overlapping regions, or require access to other unit's regions, possibly creating inter-unit conflicts. To guarantee opacity units alternate between running transactions and checking for conflicts.

The inter-unit conflicts are detected lazily when HeterosTM synchronizes units. When no conflict exists, the units exchange their produced results amongst themselves. Otherwise, HeterosTM's Conflict Resolution (CR) decides which unit gets commit, and that unit overrides other's dataset with own, effectively rolling back their transactions.

4

Results

Contents

To evaluate the performance of HeterosTM, a prototype was developed and tested in a variety of scenarios. The results of these tests are presented in this chapter. Due to time constraints, the HeterosTM programmer API was not implemented, although all the underlying synchronization and scheduling algorithms which power this API were the developed, with two exceptions: load balancing between units, and load balancing inside the units, amongst the different queues.

As mentioned in Chapter 2 the chosen solutions for the base unit's TM systems were TinySTM and PR-STM, respectively for the CPU and the GPU. Both these TMs were modified to support logging as described in Chapter 3.

HeterosTM was two tested with two benchmarks:

- Bank: a synthetic benchmark provided by TinySTM, which simulates the bank transactions by transferring money between accounts.
- MemcachedGPU: part of GPU version of Memcached [8], a popular distributed memory object caching system adopted by many popular web-services, such as Facebook or Twitter.

Due to its simple nature as synthetic benchmark, Bank was used to profile the various elements of HeterosTM. Furthermore, Bank's algorithms also allow for a fine-grained control of both intra-unit (inside each unit) and inter-unit (between units) conflict rates, which are used to study the scenarios in which the usage of HeterosTM is beneficial.

On the other hand, Memcached is a real application and tests ran with it serve to evaluate the utility of this system in a more realistic scenario. The benchmark is based on GNoM's [8] GPU algorithms for searching and inserting keys into Memcached's tables, and does not consider the full network stack.

All tests on this chapter, unless specifically mentioned, were run on a machine with an Intel Core i7-5960X processor, using a quad-core configuration, and two Nvidia Geforce GTX980 GPUs (GM204 micro architecture), connected to the CPU with a PCI-Express 3.0 bus.

4.1 Bank

Bank is a synthetic benchmark provided by TinySTM, which simulates transferring money between bank accounts. Each account is a value in an array, and Bank's various transaction operate over several of these variables with atomicity. Bank has 3 types of transactions: Transfer - which transfers money between accounts, Total - which reads the values of all banks accounts and adds them up, and finally Reset, which returns all accounts to 0. For homogeneity purposes the test presented here use only the Transfer transactions.

The Bank benchmark was designed with the Transfer transaction happening between only two accounts. However, to test a larger variety of scenarios this benchmark was extended to support transfer between multiple accounts within a single transaction. Both this decision, and usage of transfer transactions only, mirrors the tests used to evaluate PR-STM.

This benchmark's simplicity allows it to be used to profile the behaviour of both processing unit's TMs separately as well as the various elements combined solution, at a lower level. Additionally, since

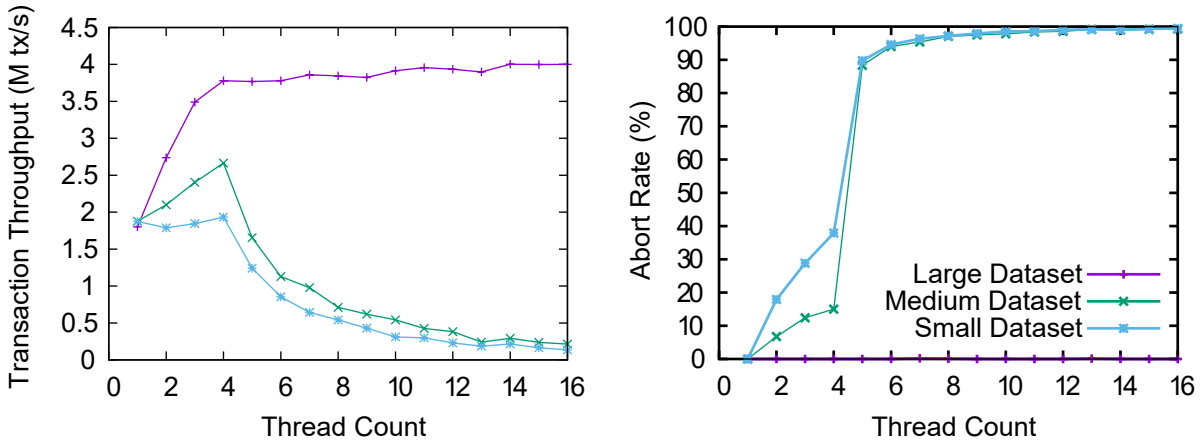
this is an easily partitionable benchmark, it was used extensively to profile the Synchronization phase's functioning with respect to how the different CRs work, and how the different Synchronization algorithms perform.

For the Bank benchmark only the synchronization algorithms were studied, as the transaction queues were not implemented. Instead, as is the design of Bank, each unit generated its own transactional inputs locally.

The approach taken to evaluate the system, was to begin by analysing the CPU/GPU only solutions first, which provide a baseline for further comparisons with the combined system. Furthermore, this allows the system to be correctly tuned to run in an ideal workload for both units. After these tests, the combined system is tested in several configurations, as well as different levels of intra and inter contention.

4.1.1 Evaluating TinySTM

In order to profile TinySTM, the first step is to evaluate its performance for various levels of contention and different thread counts. The Bank benchmarks was tested with three different dataset sizes to observe different levels of intra-contention. For this Tiny was evaluated using 3 datasets, with each being twice as large as the previous, with the Medium one being 2 KB in size. The results of this test are presented on Fig. 4.1.



(a) Throughput Comparison

(b) Contention Comparison

Figure 4.1: TinySTM evaluation test.

From analysing these results, it can be concluded that the ideal number of transactional threads for TinySTM is four. From the contention comparison graph specifically, it is observed that the abort rate is vastly increased for runs with over four threads, except for the scenario with very low contention, due to its very large dataset.

Although the processor used for these tests has only four physical cores, it has support for eight concurrent threads in hardware, via hyper threading. As performance only degrades for thread counts over 4, and only for scenarios with contention, it can be deduced that hyperthreading exacerbates contention's impact on performance. This happens due to hyperthreading causing threads to sleep, which,

in a lock heavy workload (as is the case for TM), causing a thread to sleep may stall another waiting to acquire its locks

Having set TinySTM to use 4 threads, the next step was to test the impact of the developed logging solutions on performance. As is visible on Fig. 4.2 the results of this test match the goals proposed in Section 3.3.1, as the impact of the logging solution in CPU TM performance is negligible.

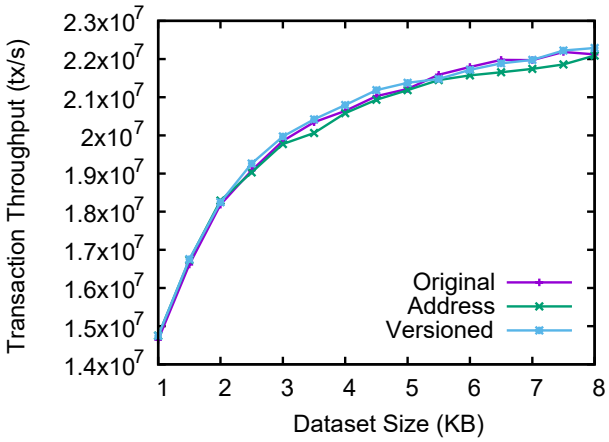


Figure 4.2: CPU log comparison.

4.1.2 Evaluating PR-STM

The tests set for PR-STM, the GPU TM, aim for the same goals as the ones in Section 4.1.1. To obtain the correct thread/block configuration Bank was ran with different warp and block sizes, and a large dataset to avoid contention as a factor.

In Fig. 4.3, it can be observed that whilst performance is pretty close for all block sizes, runs with blocks of 1024 threads are slightly inferior, whilst blocks of 128 threads offer the best overall throughput. The GPU used to obtain this results uses the Nvidia Maxwell micro architecture, whose SMs support only up to four warps (128 threads) issued per clock cycle, which explains why this thread count excels.

From the graph it is also observable that total thread counts over 5000 are enough to fully saturate the GPU, therefore a configuration of 64 blocks of 128 threads (8192 total threads) is defined as being the point of ideal performance for HeterosTM testing, since it comfortably saturates the GPU.

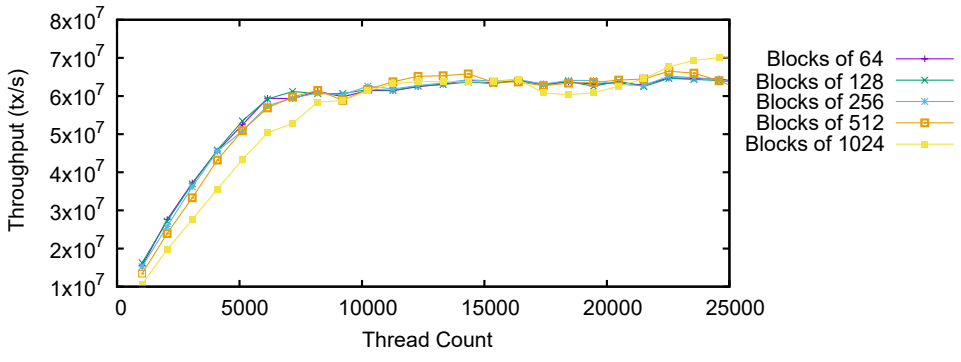
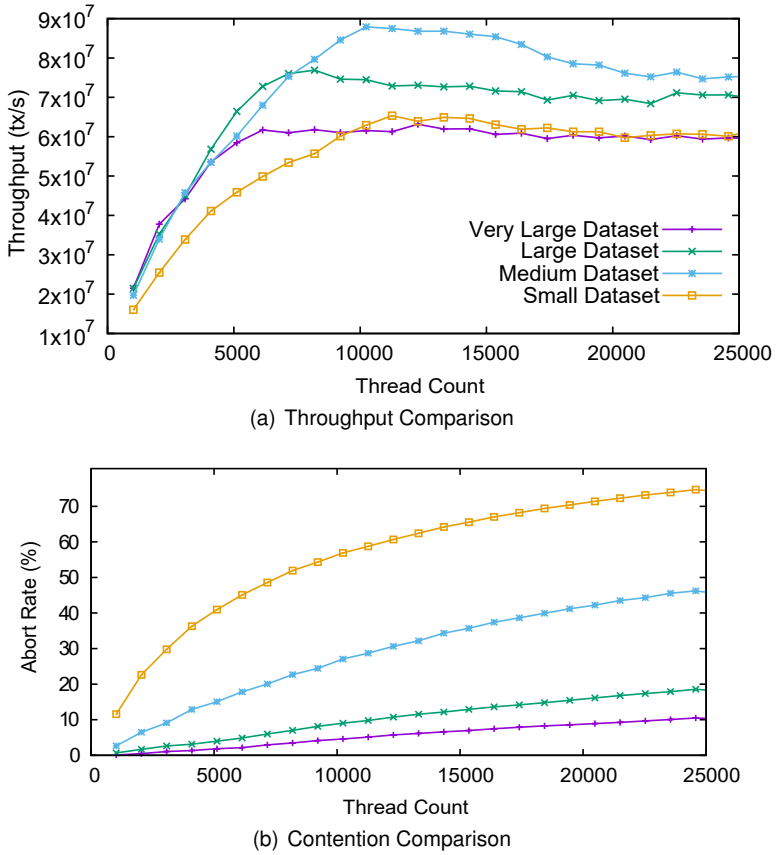


Figure 4.3: PR-STM thread/block evaluation.

Using the ideal configuration of 128 threads per block, contention is analysed. The results are presented in Fig. 4.4. Contrary to what was expected, the Medium sized dataset (2 MB) is the one that provides the highest throughput, despite the having a higher contention than the one on the very large datasets. Using Nvidia's profiler, it was discovered that the Medium dataset is small enough that both it, and PR-STM's meta-data can fully fit into the L2 Cache of the GPU. As this workload is mainly memory-bound, the guaranteed cache hits greatly increase performance for smaller workloads, and offset the overheads caused by contention.



(a) Throughput Comparison

(b) Contention Comparison

Figure 4.4: PR-STM evaluation test.

Finally the performance of the logging solutions presented in Section 3.3.2 is evaluated. As expected, the Compressed log has a higher overhead than the Explicit log, since it requires greater computational effort and has fewer coalesced memory accesses. Furthermore, when comparing with the CPU's logs both these solutions have a higher overhead than when compared to the base PR-STM. This is because in the GPU's architecture accesses to global-memory, where the logs are kept, are by nature not very efficient, and represent a big bottleneck on system performance.

4.1.3 Evaluating HeTM

HeterosTM' Bank tests analysed the following parameters: size of the dataset, transaction size and synchronization frequency. These tests were run with both types of CPU logs and the Favour Faster CR. For both thr Address log and Versioned log, the Blocking synchronization algorithm was used.

For these tests, the units were configured using the information gathered in the previous tests: the

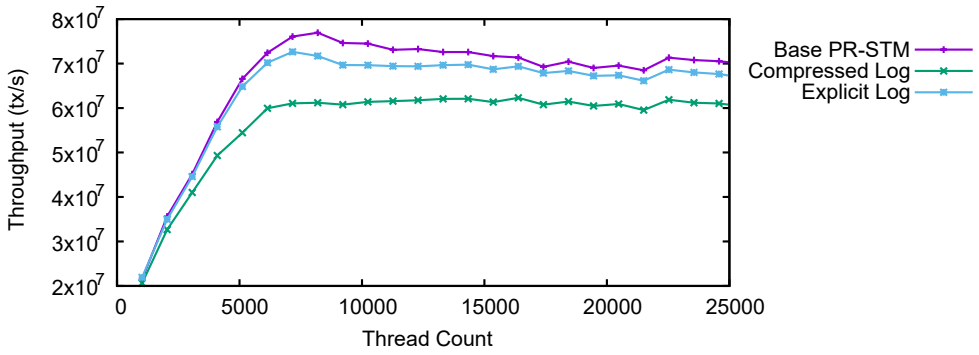


Figure 4.5: GPU log comparison.

CPU uses 4 threads, and the GPU uses 64 blocks of 128 threads. To obtain different levels of intra-contention each unit's partition is set to the sizes used in their respective evaluations.

Dataset Size

The dataset size influences the intra contention and the cost of synchronization when using the Address Log. To evaluate how performance is affected by this factor, a scenario with no inter contention was used. Fig 4.6 presents the results of this test.

Peak performance for the combined system is hit when the GPU hits its peak performance point, as the CPU's performance is mostly independent of dataset size, and this dataset is large enough that contention is not influential for the CPU.

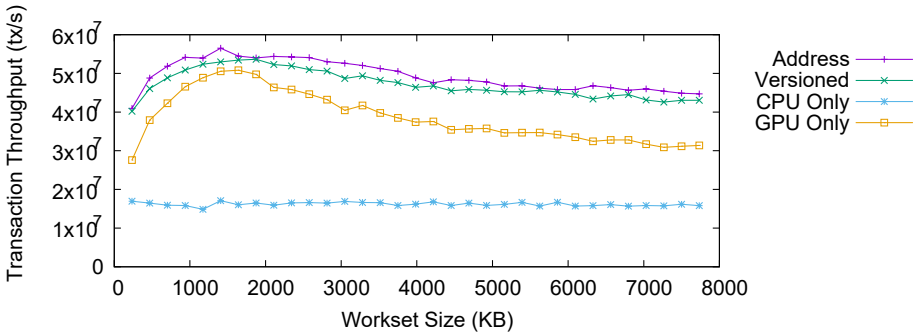
As expected for small datasets, were transferring the whole memory image is not costly, the Address log outperforms the heavier memory transfers of the Versioned Log. On the other, for the large datasets, the Address log's performance decay is noticeably higher and the Versioned proves to be the better solution.

Overall, despite the weaker performance for small datasets, the Versioned log is the most robust solution as it is only slightly worse than the Address Log for small datasets, whilst still scaling for larger ones. Furthermore, for this test, the chosen workload heavily favours the Address log's design, as it is write dominated. In scenarios were writes represent a smaller percentage of the workload, transferring the full dataset is even more inefficient.

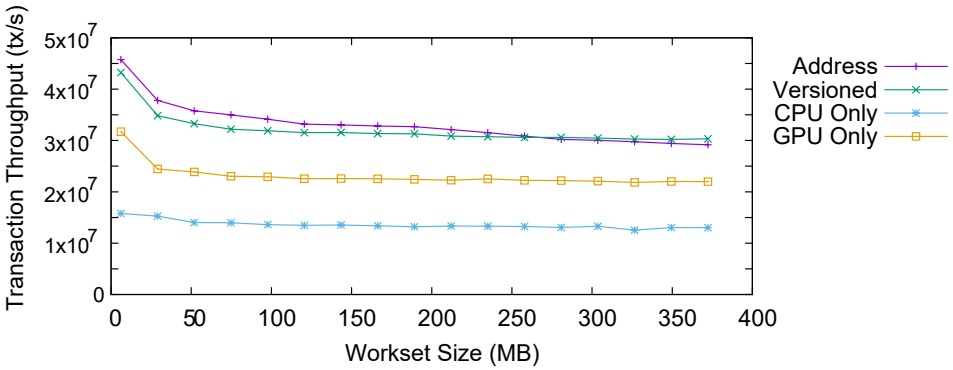
As each of the logging solutions excels in one area, HeterosTM provides both, leaving to the programmer the choice of which is the best for his application.

Transaction Size

The transaction size test highlights some of the weaknesses of the GPU. As is pictured in Fig. 4.7, increasing the transaction size greatly decreases the GPU's performance, whilst the CPU remains mostly stable. Here the overhead incurred by PR-STM's two-step validation, which requires parsing the full readset twice, when combined with the inherent increased contention incurred by increasing the size of the transactions, causes the GPU's performance to decay greatly.

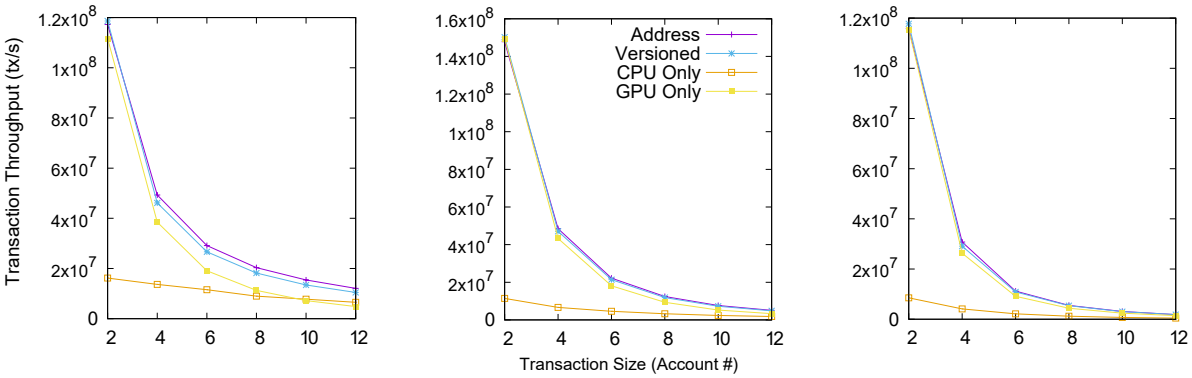


(a) Small Dataset



(b) Large Dataset

Figure 4.6: HeterosTM performance comparison for varying dataset sizes.



(a) Low Contention (b) Medium Contention (c) High Contention

Figure 4.7: HeterosTM performance for different transaction sizes.

When running longer transactions, the full system performance decays to that of a singular unit, as the slowdown incurred by running larger transactions causes the overhead of synchronization to nullify the performance gains of running on two units simultaneously. The increased size of the transactions increases the size of the CPU's logs, whilst synchronization frequency decreases, due to being linked with the GPU's throughput.

Sill, even for the more unfavourable scenario, HeterosTM is capable of matching the fastest (GPU) single unit performance.

Synchronization Frequency

When using a speculative model, the frequency of error checking is a key tuning parameter. If synchronization is too frequent, the overhead incurred with the constant stopping becomes too great and performance decays. On the other hand, synchronizing too late, may lead to a lot of wasted work, when one of the units is eventually doomed to fail.

To establish an ideal frequency of synchronization, Bank was tested with varying GPU work batch sizes as well different levels of overlapping between the two unit's partitions. This test was ran using the Versioned Log, and the Favour Faster CR. Fig. 4.8 presents the results of the test, and Fig. 4.9 presents the success rates for the various levels of contention.

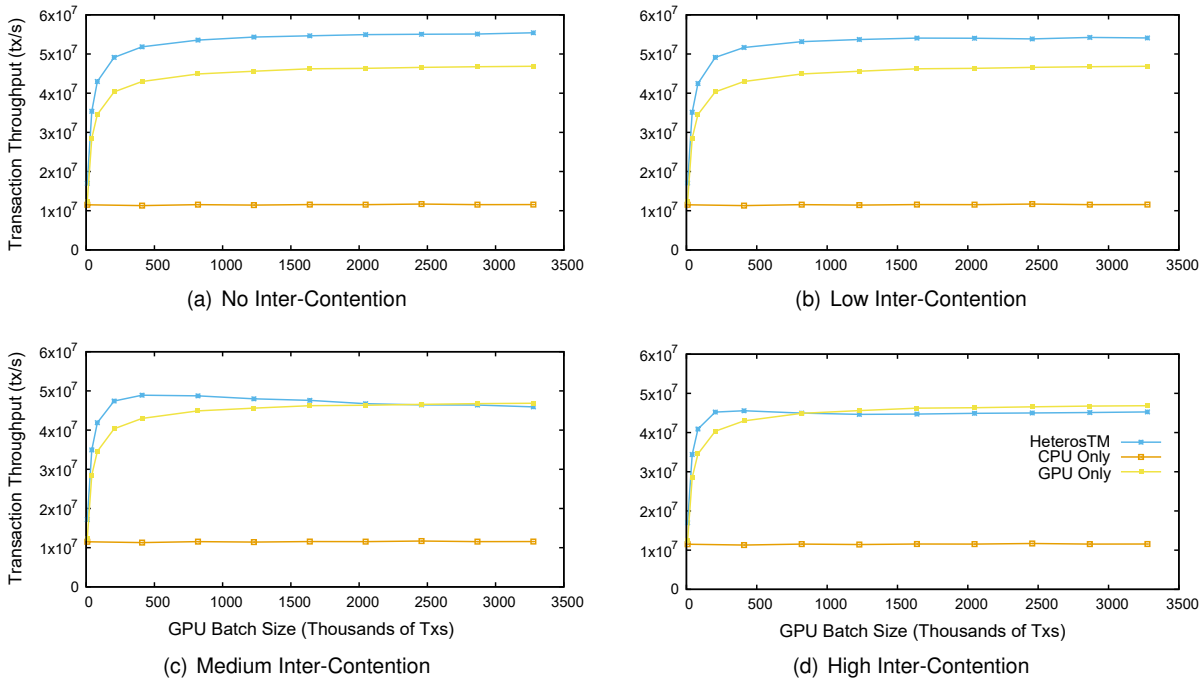


Figure 4.8: HeterosTM performance evaluation for different inter-contention values, using high intra-contention.

Analysing these results, it is observable that HeterosTM outperforms CPU/GPU only solutions for any inter-contention levels, as long as the batch sizes are bellow 500 thousand transactions. As expected, when batches start to get larger both the probability and the cost of conflicts increases as the units produce more results between each synchronization. Since for this test the CR of choice was the Favour Faster CR, in the worst scenarios were synchronization fails repeatedly, performance degrades to a level slightly inferior than the one obtained with the GPU only solution, as this is the unit with the highest throughput.

On the other hand, in the low inter-contention scenarios, HeterosTM's peak performance is close to the combined performances of the two involved units. This means that the developed system is competitive even with fine-grained solutions with perfect partitioning, despite its generic design.

From these graphs, it can be concluded that HeterosTM's synchronization algorithm meets the goals set for the system, by allowing collaborative access to the same memory region in different units, so as long as it is tuned correctly and/or partitioned with a high degree of confidence. However this assumes

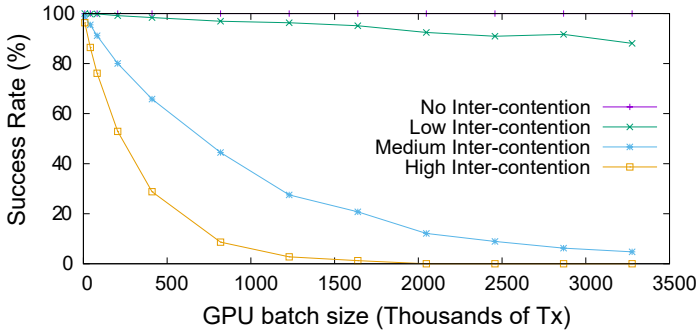
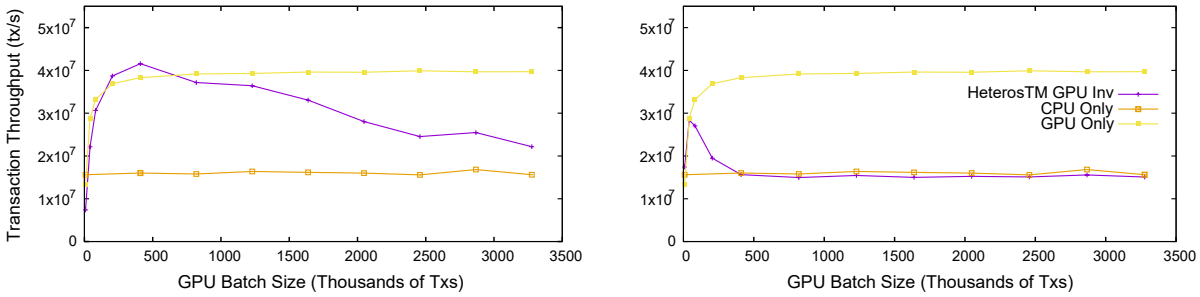


Figure 4.9: Synchronization success rate.

CPU invalidation which may not be desirable for some workloads. Running the same test using the static GPU invalidation the results are those present in Fig. 4.10.



(a) Medium Inter-Contention

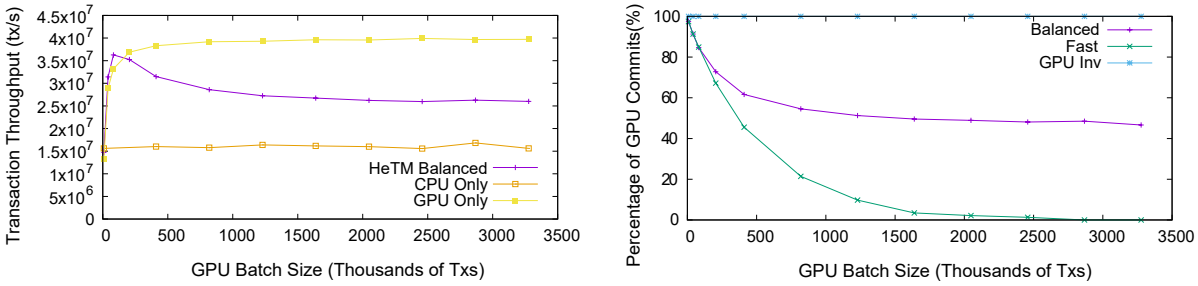
(b) High Inter-Contention

Figure 4.10: GPU Invalidation performance evaluation.

In this scenario HeterosTM's performance instead degrades to that of the CPU, which for this scenario significantly impacts the throughput of the system. The system's performance is heavily impacted in the high inter-contention scenario, where it only out performs the GPU for really high frequencies of synchronization, and even so, only marginally. However, for the medium inter-contention scenario, the system still outperforms single unit, at the same range of frequencies as the Favour Faster CR. Furthermore, the goal for this CR was that synchronization did not degrade the CPU's performance, which can be considered to be achieved, due to the throughput difference between the CPU and HeterosTM being inferior to 5%.

Both these CR algorithms may lead to starvation of one of the units, either by design on the GPU Invalidation algorithm, or due to workload related factors with the Favour Faster CR. To counter this HeterosTM presents the Balanced CR, which uses tries to balance the commits between the two units according to factors set by the programmer. For testing purposes, Balanced CR is configured to block CPU commits for every failed GPU commit. ¡repeating the test for the Medium Inter-Contention scenario the results in Fig. 4.11 are obtained.

Observing these results, its clear that performance for this configuration of Balanced CR falls between Favour Faster and GPU Invalidation, failing to achieve the same performance degradation of Favour Faster. However, looking at Fig. 4.11b), it can be seen that the goal of balancing commits between both units is achieved, which may be more important than throughput in certain applications.



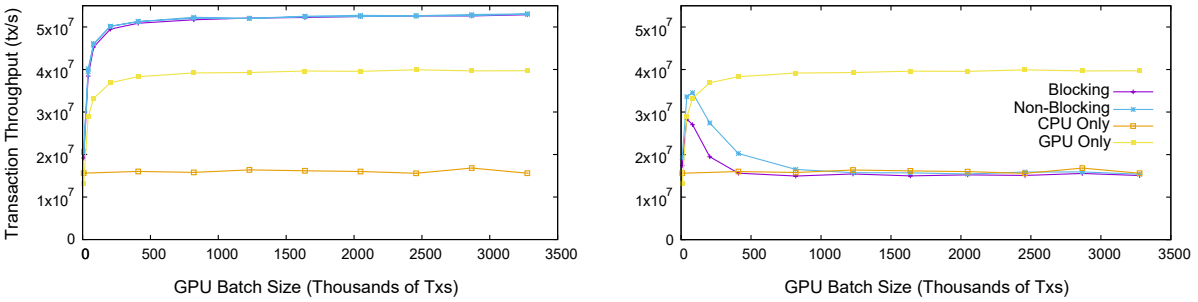
(a) Throughput Comparison (b) Synchronization Success Rate

Figure 4.11: Performance evaluation for Balanced CR

It should also be noted that the Balanced CR allows for other configuration options, allowing the user to fine tune it to his performance needs.

Non-Blocking

So far, all tests were ran using the Blocking algorithm. In Fig. 4.12 this algorithm is compared with the Non-blocking algorithm, for the two extreme inter-contention scenarios. The GPU Invalidation algorithm is used, as it facilitates the observation of the differences between the algorithms.



(a) No Inter-Contention (b) High Inter-Contention

Figure 4.12: Comparison Between blocking and non-blocking algorithms.

When Inter-contention is low, the performance difference between the two algorithms is negligible. On the other hand, in high contention scenarios, the difference is much starker, with the Non-Blocking algorithm being considerably better. In these high inter-contention scenarios, much of the time spent synchronizing is wasted, as the comparisons are doomed to fail, and the CPU might have been producing results instead of locking, which is exactly what the Non-blocking algorithm allows, and why its performance is superior.

Comparison with Zero-copy

In Chapter 2, Zero-copy was presented as having similar functionality to the system herein proposed. Zero-copy functions in a very different way to HeterosTM. When the CPU accesses memory which is currently being accessed on the GPU, the Nvidia API generates a page-fault and transfers the value. This imposes a much higher overhead than HeterosTM's speculative scheme, as it constantly saturates the PCI-Express connection between the CPU and GPU with small, inefficient, memory transfers. To

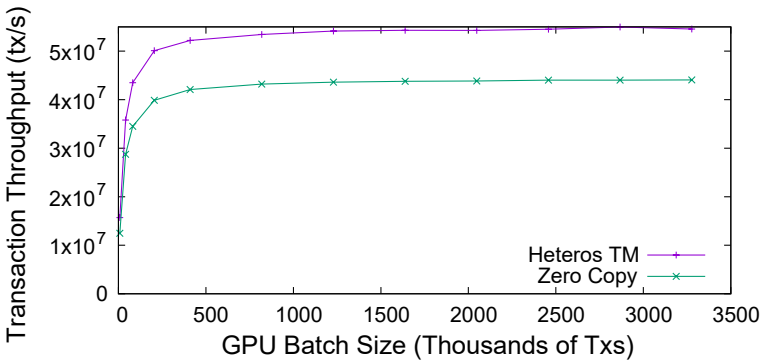


Figure 4.13: Comparison with Nvidia's Zero-copy.

compare them HeterosTM was converted to use Zero-copy, for Synchronization only, where transfers are bigger and more efficient.

As pictured in Fig 4.13, Zero-copy presents poor results, as its lazy approach to memory transfers to the CPU is slower than HeterosTM's in-place memory overrides, as these transfers allow for efficient batching during the transfer, which ends up being much more efficient, in terms of latency, than only transferring when necessary.

4.2 MemcachedGPU

The second benchmark ran to profile HeterosTM is an adaptation of the MemcachedGPU algorithm, by Hetherington et al [8]. This algorithm is part of a GPU conversion of Memcached, a distributed, in-memory, object caching system, used in several web applications to reduce database loads. Memcached has also been converted to use TM [34], making it an ideal application to benchmark HeterosTM.

The full conversion of Memcahed, considers every aspect of processing requests, from receiving packets, to building and sending reply packets. For the purposes of benchmarking HeterosTM, the networking is discarded, and only the request processing is considered. MemcachedGPU supports two types of requests:

- GET: Requests for an in-memory object.
- SET: Request to add a new object to memory.

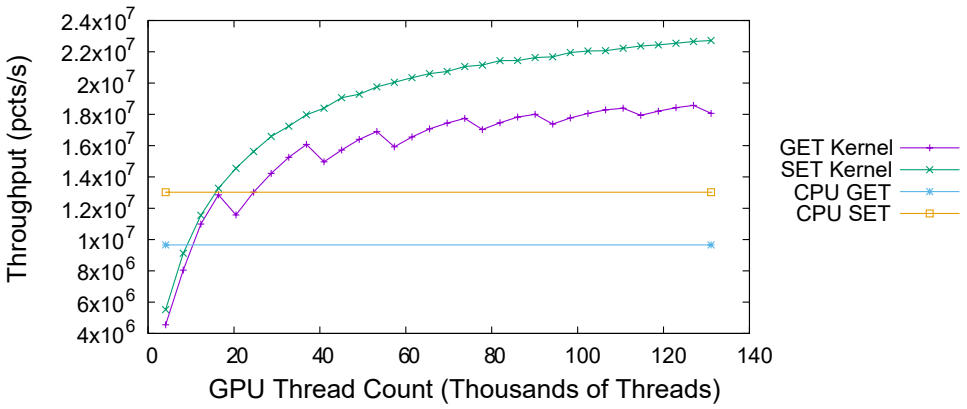
To refer the objects it is caching, Memcached uses a key-value store schema. In the original CPU algorithm key-value pairs are stored using an hash map used to lookup the file pointer to the corresponding key. This approach requires dynamic memory allocation, therefore to adapt it to GPU it was necessary to make some compromises. Instead of an hash map, MemcahedGPU uses a structure similar to an hash table, with a fixed number of entries per hash value, functioning in a model that is more akin to that of a fully-associative cache with multiple ways, with evictions used when an hash table entry has all its ways full.

For the purposes of testing HeterosTM, the algorithms concerning both types of requests were implemented for both units, each using their respective TMs. Requests were partitioned between the two units using hash algorithms.

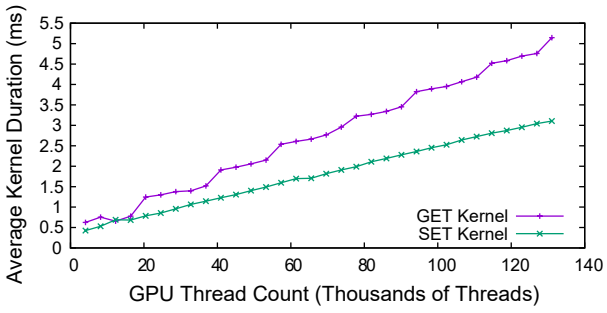
For these tests, a more complete version, when compared to Bank, of HeterosTM was implemented, making use of unit queues and transactional inputs. As the network components of Memcached were considered to be out of scope, the system is initialized with a large number of already queued transactional inputs. Deciding between the different transaction types is done by a con-flip at every queue fetch.

4.2.1 Evaluating GPU/CPU only

As with Bank, the first step to this evaluation is analysing the separate units performance. Furthermore, since we have multiple types of transactions, these must also be analysed in isolation. Fig 4.14 presents both the transactional throughput, and the kernel duration, as for an application with time requirements like Memcached, this might be a more important metric than raw throughput. Blocks were set to 128 threads, from previous results with the Bank benchmark.



(a) Througput



(b) Latency

Figure 4.14: Evaluation of Memcached unit implementations.

From the observation of Fig. 4.14a) the behaviour of the GET kernel stands out as being the most unexpected result. As read-only transactions on PR-STM, always succeed, the main bottleneck for their execution is global memory accesses. When threads are waiting for the global memory, CUDA's dynamic parallelism while issue instructions to another warp, to improve performance. For the GET kernel this means that the scheduler will keep changing blocks untill it reaches a point were the register file is no longer enough for all the threads, forcing register spill, which causes register evictions to the cache. As more and more registers get evicted, they start replacing each other in the cache, leading to

a point where they override each other and are force to move to global memory, causing the ”ladder” effect. This problem is not present in the SET kernel, as this kernel uses more cache, stopping evictions caused by register spilling.

From analysing these graphs, it can also be seen that for this workload CPU and GPU are much closer in terms of performance, when compared to Bank, as these are very small batches, and the CPU does not require memory transfers for its work.

4.2.2 Evaluating HeTM

To test HeterosTM, the GPU was configured to use roughly 15 thousand threads (128 block of 128 threads each), in order to hit a processing latency of around 1 ms. The workload was configured using the data from Facebook [47] as the basis, with 0.1% of SET requests, and the rest of the requests as GET requests. To simulate Load Balancing or errors in partitioning, units were also made to do a certain percentage of fetches from the shared queue. As this is a workload were writes are very sparse, and the dataset very large the Versioned Log was used, along with the Non-blocking algorithm.

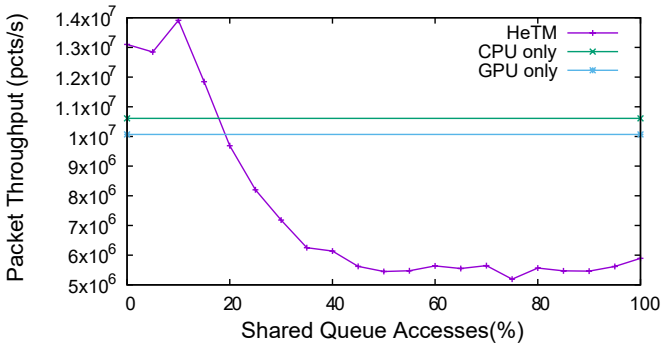


Figure 4.15: Memcached performance using HeterosTM with 0.1% SET requests.

The results of these test, observable in Fig. 4.15, show that HeterosTM is able to outperform single unit solutions in scenarios were the partitions do not overlap significantly. However, in this scenario performance does not degrade so gracefully, and high contention scenarios have roughly half of the throughput of the single-unit performance. This due to two factors: necessity to transfer results, and unfavourable workload configuration for the GPU.

Unlike Bank, Memcached requires transactions to have a return value: the response packet. This increases the number of memory transfers significantly, which in turn increases the overhead of synchronization. Furthermore, whilst transactional inputs are transferred during the execution phase, meaning they do not impact CPU performance, they still impose a large overhead to GPU execution, especially so in this scenario where the kernels are very short.

Adding to this, Memcached’s workload, despite being highly parallel, is not very well catered to run on GPUs, due to the time requirements forcing packets to be processed in small batches were the overheads of the transfers are rather large. To obtain some better data on the system, Fig. 4.16 presents a scenario were these time requirements are ignored, and the previous tests are repeated using different thread counts, and therefore different batch sizes.

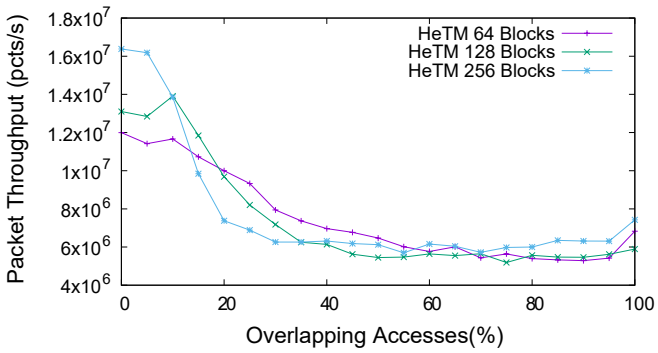


Figure 4.16: Memcached performance with varying batch sizes.

The results of this test highlight that as the block count decreases, and therefore the workload per synchronization, the overall system performance decreases considerably, however it is more resistant to conflicts. When using a higher thread count the results look more positive, with a good performance gain, however the cost of inter-unit conflicts increases significantly. This highlights how dependent on tuning HeterosTM is in order to obtain good performances in scenarios with overlapping.

4.2.3 Multi GPU

Following the tests with a single GPU, HeterosTM was tested using the extended multi-GPU design presented in Fig. 3.2. This test was ran using the same configurations as the previous Memcached HeterosTM tests: 128 blocks of 128 threads and 0.1% of SET requests, using two GPUs. The results of this test are presented in Fig. 4.17.

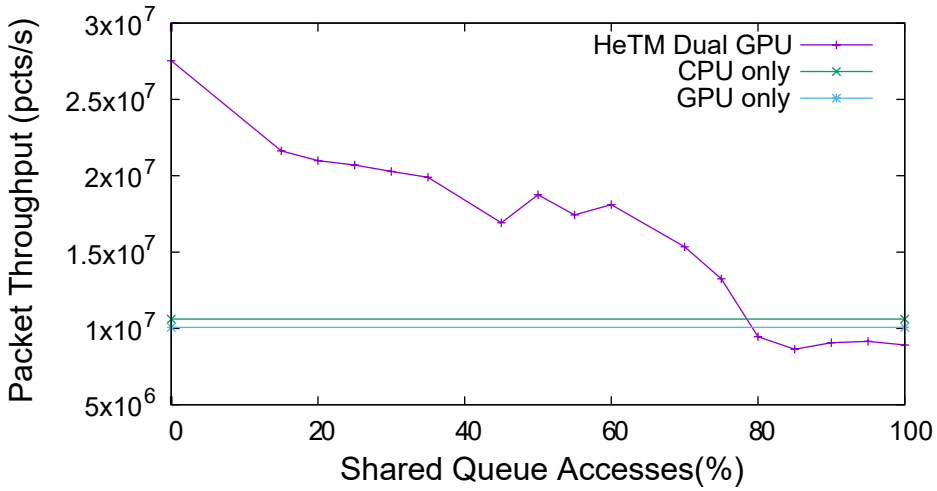


Figure 4.17: Memcached performance using HeterosTM dual-GPU.

These results point to HeterosTM being more favourable when using multiple GPU instead of a single one. The reality is that, the multi-GPU algorithm causes much longer synchronizations, and the CPU spends much more time locked. This essentially causes faster per-unit comparisons despite the overall comparison time being longer. The brunt of the results are then produced by GPUs, which are rarely aborted by the CPU, and compete only amongst themselves.

Whist it is intuitive to expect that inter-GPU conflicts should have a higher impact on performance,

the use of GPU direct to transfer logs from unit to unit, along with the smaller size of these logs, makes GPU comparison more efficient, with conflicts being more readily tossed aside instead of necessitating to parse several logs as is the case with the CPU comparison.

4.3 Summary

In this Chapter, HeterosTM's performance was evaluated be running several tests to its various elements. For these tests, two benchmarks were used.

The first benchmark used to evaluate HeterosTM, Bank, is a synthetic benchmark. Tests ran with this benchmark highlighted the performance of the various parts of the synchronization algorithm. Results show that HeterosTM meets the performance goals set for it: outperforms single unit performance significantly and still function with some inter-unit contention, so as long the synchronization frequency is high enough and the inter-contention not very high. These tests also serve to highlight the difference in performance of both logging solutions, as well as the various CR implementations.

Having benchmarked synchronization, the next step is to evaluate the complete system. For this the MemcachedGPU algorithm is used. Memcached is a popular in-memory object caching system, that as been ported to GPU. Using this algorithm, HeterosTM is benchmark for a realistic workload achieving positive results for low contention, but the performance gains are as significant as those of Bank due to the added transfer overhead. MemcachedGPU is also used to test multi-GPU implementation of HeterosTM, which yields great performances, due synchronization between GPU's requiring few transfers.

5

Conclusions

Contents

This work presents Heterogeneous Transactional Memory (HeterosTM), a system which simplifies programming for heterogeneous by allowing concurrent accesses to same memory, by multiple, physically split, units. To achieve this the system applies a speculative model, were units work separately on the data, and periodically check to see if they've committed any overlapping accesses.

The guarantee correctness, the system runs on a two state model, either running transactions or checking for inter-unit conflicts. During this stage units exchange logs and check for conflicts. Upon conflict detection, HeterosTM uses Conflict Resolution (CR) policies to decide which unit gets to keeps its produced results and which gets aborted and overridden with the other's results. This means that HeterosTM uses a two-stage commit for its transactions, were produced results are only valid after unit synchronization.

Users of the system, code for each of the units using transactional code, and establish inputs and outputs for these functions. The user registers these functions into HeterosTM, and then starts it with the desired characteristics. Having started the system, the user inserts transactional inputs into the HeterosTM system, along with the an ID of the unit where it expected to run. The system uses this ID, along with internal considerations, such as load balancing, to schedule and run the transaction in one of the available units.

The results obtained when testing the system, prove that the proposed design achieves the goals set for it, after some application specific tuning. This means that the goal of facilitating programming was not fully achieved, however, abstracting the difficulties of synchronizing the different units is still a significant gain. In contrast, the performance and correctness goal were entirely achieved, proving that this system can succeed.

5.1 Future Work

As this is, to the best of the author's knowledge, the only work in the field of heterogeneous transactional memory, development was faced with several challenges, some of which were not completely solved.

First among these challenges is the system's usability. Due to time constraints, it was not possible to fully develop the API. Even so, requiring the user to write separate transactional code for both units is not very user friendly. This is a concession, as both CPU and GPU use different programming model and a transactions cannot be written to work on both units within the used programming models. However, other models, such as OpenCL allow the user to write device agnostic code, and porting the system to this model could provide a better API. Unfortunately for current research, OpenCL lacks many of the functionalities of CUDA such as host pinned-memory for faster transfers. However, as development on OpenCL continues it is likely it will become the system of choice for future development of this work.

In order to develop a more flexible system, the specific algorithms for partitioning are left to the programmer, as a generic solution is very difficult to design, and was considered to be out of scope. Once gain, this harms the system's usability by requiring more effort from the programmer. Research on automatic partitioning of transactional code already exists [48], [29], which would be an interesting

solution to avoid user input on partitioning.

Finally, for the sake of facilitating the development of this work, heterogeneous systems were considered only as CPU plus dedicated GPU, inter-linked through a PCIe connection. This a reductive view, and HeterosTM could be expanded to cover more types of heterogeneous system, such as:

- Multi-GPU systems: whilst some brief work was done on this front, HeterosTM's design is dual-unit oriented and could be expanded to more efficiently synchronize in multi-GPU scenarios, by for instance not requiring all units to synchronize at the same time.
- Integrated GPUs: integrated GPUs are historically worse performers when compared to dedicated offerings, however, as they become ever more prevalent in consumer technology, this trend has is starting to be reversed. As integrated GPUs share their memory with the CPU, they present all new challenges and opportunities for collaborative work involving the CPU, which would be compelling research.

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] S. Mittal and J. S. Vetter, “A survey of cpu-gpu heterogeneous computing techniques,” ACM Comput. Surv., vol. 47, no. 4, pp. 69:1–69:35, Jul. 2015. [Online]. Available: http://doi.acm.org/10.1145/2788396

[8] 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

[9] 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

[10] 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

[11] 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

[12] 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.

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

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

[15] 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.

[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] 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.

[34] 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

[35] 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.

[36] 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.

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

[38] 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.

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

[40] 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.

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

[42] 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.

[43] 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.

[44] Z. Zhong, V. Rychkov, and A. Lastovetsky, “Data partitioning on multicore and multi-gpu platforms using functional performance models,” IEEE Transactions on Computers, vol. 64, no. 9, pp. 2506–2518, 2015.

[45] 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, Jan 2014.

[46] R. Palmieri, F. Quaglia, P. Romano, and N. Carvalho, “Evaluating database-oriented replication schemes in software transactional memory systems,” in Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on. IEEE, 2010, pp. 1–8.

[47] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab et al., “Scaling memcache at facebook,” in Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13), 2013, pp. 385–398.

[48] 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.