TÉCNICO
LISBOA

Debunking the CPU+GPU Processing with Transactional Memory

Diogo Miguel Rainho do Nascimento Nunes

Thesis to obtain the Master of Science Degree in

Integrated Master in Electrical and Computer Engineering

Supervisor: Prof. Paolo Romano

November, 2021

Declaration

Resumo

Os GPUs são tradicionalmente utilizadas em aplicações com acessos regulares facilmente paralelizáveis. No entanto, ao longo dos últimos anos, os GPUs foram também utilizados com sucesso para acelerar aplicações de acessos irregulares em vários domínios usando esquemas de sincronização de baixa granularidade.

Infelizmente, estratégias de sincronização de baixa granularidade são notoriamente complexas e sujeitas a erros. Tal tem motivado a busca por paradigmas alternativos que visam simplificar a programação concorrente e, entre estes, a Memória Transacional (TM) é provavelmente uma das propostas mais proeminentes.

Esta tese lança luz sobre as limitações de escalabilidade que surgem ao portar algoritmos Multi-versioned STM inicialmente projetados para CPU para o domínio do GPU e ainda introduz CSMV (Client Server Multi-versioned). CSMV é uma STM multi-versioned para GPUs que adota um design cliente-servidor inovador. Ao desvincular a execução de transações do seu processo de commit, CSMV oferece dois benefícios principais: (i) permite o uso de memória de baixa latência no chip para aceder a metadados globais usados para sincronizar as transações (ii) permite a implementação de colaboração altamente eficiente para procedimentos de commit, adaptados para tirar o máximo proveito das características estruturais do GPU.

Por meios de um extenso estudo experimental, esta tese mostra que o CSMV atinge ganhos de até 3 ordens de magnitude em relação a STMs de última geração para GPUs e que pode acelerar em até 20$\times$ aplicações irregulares que executam STMs MV de última geração para CPUs.

Palavras-chave: Memória Transacional, Unidade the Processamento Gráfico, Memória Transacional em Software, Multi-version Concurrency Control, Sincronização, Arquitetura Cliente-Servidor

Abstract

Graphic Processing Units (GPUs) have traditionally focused on streaming applications with regular parallelism. Over the last years, though, GPUs have also been successfully used to accelerate irregular applications in a number of application domains by using fine-grained synchronization schemes.

Unfortunately, fine-grained synchronization strategies are notoriously complex and error-prone. This has motivated the search for alternative paradigms aimed to simplify concurrent programming and, among these, Transactional Memory (TM) is probably one of the most prominent proposals.

This thesis sheds light on the scalability limitations that arise when porting multi-versioned Software Transactional Memory (STM) algorithms designed for the Central Processing Unit (CPU) to GPUs and introduces CSMV (Client Server Multi-versioned TM). CSMV is a multi-versioned STM for GPUs that adopts an innovative client-server design. By decoupling the execution of transactions from their commit process, CSMV provides two main benefits: (i) it enables the use of fast on-chip memory to access the global metadata used to synchronize transactions (ii) it allows for implementing highly efficient collaborative commit procedures, tailored to take full advantage of the architectural characteristics of the GPU.

Via an extensive experimental study, this thesis shows that CSMV achieves throughput increases of up to 3 orders of magnitude with respect to state of the art STMs for GPUs and that it can accelerate by up to 20$\times$ irregular applications running on state of the art MV STMs for CPUs.

Keywords: Transactional Memory, Graphic Processing Unit, Software Transactional Memory, Multi-version Concurrency Control, Synchronization, Client-Server Architecture

Contents

List of Tables

List of Figures

Acronyms

ATR Active Transaction Record. x, 25–28, 30–32, 34–36, 38–40, 43, 44, 46, 47, 49–51, 53, 55, 59

CAS Compare-and-Swap. 3, 32, 34, 35, 40

CPU Central Processing Unit. vi, 2–4, 11, 14–16, 18–20, 24, 25, 53, 58

GPGPU General Purpose Computing on Graphic Processing Units. 4, 18

GPU Graphic Processing Unit. vi, 2–4, 11–26, 44, 47, 53, 58, 60

GTS Global Time Stamp. 25–27, 31, 32, 34, 35

HTM Hardware Transactional Memory. 8, 23

MV Multi-versioned. 3, 4, 9, 25, 44, 45, 53, 54, 56, 57

MVCC Multiversion concurrency control. 6, 25

ROT Read-Only Transaction. ix, x, 9, 35, 36, 43–57

SM Streaming Multiprocessor. 11–13, 15, 29, 30, 59

STM Software Transactional Memory. vi, 3, 4, 8, 18–26, 44, 45, 53, 58

TM Transactional Memory. vi, 3–10, 19, 22–24, 45

Vbox Versioned Box. 25–27, 34–36, 41

Chapter 1

Introduction

Over the last years we have witnessed a growing interest in using GPUs to accelerate applications with irregular access patterns, such as graph applications [1], machine learning [2], indexes [3, 4] and other concurrent data structures [5]. These applications generate data-dependent access patterns over shared mutable data, which poses a notoriously complex and error-prone problem: how to efficiently synchronize concurrent memory accesses issued by the 1000s of threads supported by modern GPUs. In fact, traditional synchronization approaches, based either on fine-grained locking or lock-free schemes, are known to be very hard to reason about and verify, being prone to a number of subtle concurrency bugs like deadlocks, livelocks, and data-races.

Transactional Memory (TM) [6] has emerged as an attractive alternative synchronization paradigm that can dramatically simplify the development of concurrent applications via the intuitive and familiar abstraction of atomic transactions [7]. When using TM, programmers only need to demarcate which code blocks need to be executed atomically, delegating to the TM implementation the responsibility of how to achieve atomicity. Over the last two decades, a large number of TM implementations, in hardware, software and combinations there of, have been proposed in the literature, initially targeting multi-core CPUs [8–12] and, more recently, GPUs [13–17].

1.1 Motivation

As more programmers make use of GPUs in more general applications, synchronization approaches to handle concurrent memory accesses are needed. In fact, GPU is traditionally used to extract data-level parallelism from embarrassingly parallel applications, where memory access patterns are regular, well defined and in many cases can be statically inferred by the compiler [18]. However, many applications do require dynamic accesses, and in those cases concurrent threads may conflict when accessing the same shared resources.

This thesis focuses in this important class of applications where it is possible to extract data-level parallelism, even though the access pattern is inherently irregular, for which the GPU can be used and still obtain significant speed up gains over a CPU implementation. Such applications however, require synchronization mechanisms to function correctly and, as seen in [19], using synchronization primitives on the GPU can some times lead to better results than a well-engineered solutions tailored specifically for the GPU, which avoid locking primitives.

Coarse-grain locking is a synchronization solution that is very easy to implement. In this locking scheme, only a single lock exists that controls all the accesses to the shared objects. But, since it essentially serializes all accesses to the shared memory, the purpose of using the thousands of threads of the GPU is defeated, therefore this locking solution incurs a huge performance degradation. Fine-grained locking, on the other hand, utilizes a greater number of locks and therefore maximises performance of the application. Yet, this solution can easily run into deadlocks or livelocks, and debugging these problems is often a very time consuming process. The problem is only worse for the GPU, since it exhibits weaker memory consistency than the CPU along with thousands of concurrent threads running on a SIMT model that exacerbate the problem.

TM, as mentioned before, can alleviate the problem by abstracting these concerns away from the programmer. It can provide performance that is comparable to the use of fine-grain locking but with an ease of use at the level of coarse-grain locking. Many implementations of TM have been made for the CPU [20–24].

As for the GPU, a number of STM implementations have been proposed [25–28] that ensure correctness, however, they are still heavily influenced by the CPU implementations and do not fully exploit the architecture of modern GPUs, namely the different levels of the memory hierarchy, defaulting to always work in the slow global memory, which incurs high overheads in the management of the transactions.

Furthermore, none of the current GPU STMs explore the concept of a Multi-versioned (MV) scheme, which is well-known to provide significant performance improvements for workloads with long read-only transactions [29–31]. Instead, current implementations employ single-version schemes that are prone to starvation for these types of workloads, due to a high likelihood of conflicts between concurrent transactions. As such, there is also a gap in the literature with respect to the implementation of MV scheme in the context of GPU STMs.

1.2 Contributions

This thesis presents a novel STM algorithm named CSMV (Client-Server Multi-Versioned). When compared to existing (S)TM solutions, CSMV has a number of innovative features, which I enumerate in the following:

1. CSMV tackles what I argue to be the most critical source of inefficiency of existing TM designs for GPUs: their reliance on expensive atomic operations, e.g., Compare-and-Swap (CAS), that operate on off-chip memory (global memory in CUDA). Such a design, inherited by TM algorithms originally conceived for CPUs, generates prohibitive overheads when adopted in GPUs supporting thousands of concurrent threads, as will be shown experimentally. CSMV departs from this conventional design by adopting, for the first time in the TM literature, a client-server architecture. In more detail, CSMV delegates the most expensive phases of the commit logic to a dedicated kernel

that executes on a single Streaming Multiprocessor (using CUDA terminology) and that acts like a server responsible for determining the outcome (commit/abort) of the transactions. The remaining threads, which act like clients, are instead in charge of generating transactions and executing their logic. This design provides a number of key benefits. Not only it avoids atomic operations targeting global memory, allowing them to operate instead on the fast scratchpad memory (shared memory in CUDA) available on GPUs. It also enables the design of a novel cooperative validation scheme that leverages the efficient communication mechanisms available in GPUs to coordinate the activities of threads within the same warp.

2. In order to enhance the efficiency of its client-server design, CSMV introduces innovative client-side pre-validation and write-back mechanisms, which bring two key advantages: (i) alleviating the server load, by shifting to the client side part of the commit logic and (ii) enabling the server to "batch" the commit of the transactions generated by the same client warp, which reduces the frequency with which server threads need to resort to using atomic operations.

3. CSMV is the first software implementation of a MV concurrency control algorithm for GPUs. The advantages of MV approaches have been long studied both in the database and TM communities [29, 32, 33]: read-only transactions can be executed extremely efficiently, namely being spared from any instrumentation and validation overhead, while also avoiding to ever block or abort concurrent update transactions. As a result, MV schemes shine in workloads, frequently found in realistic applications [29], which include frequent long-running read-only transactions that, with single versioned TMs, would be doomed to suffer from (or generate) frequent aborts, or to block update transactions for unacceptably long periods of time.

1.3 Outline

In the following, I present the structure of this document: Chapter 2 provides a background on the general TM and General Purpose Computing on Graphic Processing Units (GPGPU) concepts while also delving in the state-of-the-art implementations of TM on the GPU. Chapter 3 presents a "conventional" multi-versioned implementation. By means of porting JVSTM [29], a CPU implementation of a MV STM, I gained knowledge about how a naive porting could compromise performance. In the same chapter I consider several research paths for the rest of dissertation. This initial system is named JVSTM-GPU. In the next chapter, Chapter 4, I present CSMV, which overcomes many of the limitations of the system presented in the previous chapter. Finally, in Chapter 5, I evaluated CSMV using both synthetic workloads and a realistic application, MemcachedGPU [34], comparing it to a state-of-the-art single versioned STM (PR-STM [16, 17]) and to JVSTM-GPU. The experiments highlight that CSMV can achieve up to 20$\times$ speed-ups when compared to other state-of-the-art STMs for CPU in irregular applications and up to 1000$\times$ when compared to state-of-the-art of single versioned STMs for GPUs.

Chapter 2

Background and State of the Art

Over the recent years, there has been a shift from single core architectures to multicore ones. These multicore architectures have potential to achieve a much higher raw computational power that their single core counterparts, but writing efficient programs on parallel systems is a much harder task.

There are several new challenges that the programmer must tackle in order to achieve a correct functionality of the developed applications, of which the biggest one is the existence of data races to memory locations shared among the several cores of the parallel system. Therefore, accesses to shared variables must be synchronized across all threads.

One of the most common methods of synchronizing accesses is by applying locking schemes, i.e., when accessing a shared memory region, that thread issues a locking operation, alters the memory region and finally unlocks it.

As locking schemes go, these can either be coarse-grained, which are easy to implement but end up serializing accesses to the shared regions, and thus degrading the performance, or they can be fine-grained, which provide very good performance, but make the application significantly harder to debug, due to being prone to deadlocks or livelocks.

2.1 Transactional Memory

TM is a synchronization mechanism alternative to locking that was first proposed by Herlihy and Moss [35]. TM takes an optimistic approach to synchronization, where the accesses to memory made by the threads are not mutually exclusive, unlike lock-based algorithms. By allowing multiple threads to access the same address location, TM enables more concurrency to the application. However, having threads accessing memory regions without any control mechanism would end up in data races, which in turn leads to incorrect results.

TM is inspired in the concept of transactions that already exists for databases, where running transactions never see changes made by other concurrent transactions. One can simply explain transactions as an atomic sequence of operations, i.e., either all operation execute, or none of them do.

While the transaction is executing, alterations to memory are *transient*, i.e., the effects of the operations executed within the context of that transaction are only visible after it commits. On the other hand, if, for example, a conflict occurred and the transaction must be aborted, then all the changes made by the transaction must be discarded, thus never corrupting the memory space shared by every thread. In case this last event occurs, then the transaction has to retry from the beginning.

The main idea behind an implementation of TM is to move the complexity of dealing with low level lock primitives away from the programmer, providing instead a useful abstraction to coordinating the accesses of concurrent transactions. This abstraction gives the programmer an API with functions for starting, committing, reading and writing ($StartTx, CommitTx, ReadTx, WriteTx$). All the programmer needs to do is mark the regions of code that he wishes to execute atomically, by inserting a $StartTx$ at the beginning of this section and $CommitTx$ at the end, and use the provided access functions, $ReadTx$ and $WriteTx$ when accessing a resource that is shared among transactions.

The programmer does not need to insert any manual locking mechanism, or specify which operations are allowed to run concurrently. This task becomes the responsibility of the TM implementation, releasing the programmer from such burden.

To function correctly, a TM implementation must manage the tentative work that each transaction does, i.e., it must track all the reads and writes that it does to shared objects. This is typically done in one of two ways, either all the updates are made directly to the memory location or each transaction has a private buffer where these values are stored and then pushes to memory at commit time. A TM must also be able to ensure isolation between concurrent transactions: it needs to detect conflicts that eventually may occur and resolve them, typically by aborting and retrying one of the transactions, such that it appears that these transactions happened in a single point in the program's execution timeline, as in a serial fashion.

2.1.1 TM Design Choices

There are concurrently many different ways to implement a TM, that consist in different design decisions that have been made for that implementation. Each of these different techniques have their own merits and tradeoffs, with none being clearly superior to their counterpart. In this section, we will go over the main design choices when building a TM system [36].

Eager vs Lazy Version Management (Single Versioned)

Version Management refers to how the different versions of the memory are handled by the underlying TM system. Each thread when updating to a certain location in the memory creates a new (unstable) version of that position. Is worth noting that the TM only has to maintain: $1)$ a single "shared" version; and possibly, $2)$ one "extra" version per transaction, which is essentially different from the Multiversion concurrency control (MVCC) methodology presented below.

There are two methodologies to manage updates: $i)$ the transaction writes directly to the shared memory; or, $ii)$ updates are kept in a private buffer, local to the transaction. The former is called *eager version management* (or *in-place writes*) and the latter is called *lazy version management* (or *deferred* *writes*).

Both methods require the transaction to keep some logs about the updates that it makes. If *deferred writes* are being used, the transaction needs to keep this write-log that stores every update that the transaction has made. In case it updates a value that it has already written to, then all the transaction needs to do is replace the previous entry in the write-log.

If *in-place writes* are being used instead, there is still a need to maintain a log, but in this case it needs to store the old values that were on the position before the transaction has made the current changes. Whenever a thread updates a location, it must first read the old value, store it locally in a private undo-log, and only then make the update.

When using *deferred writes*, and when the transaction has reached the commit stage, it copies all the entries in their write-log to the corresponding memory locations. However if the transaction needs to abort, all the transaction needs to do is discard the write-log, since their updates never reached the main memory.

For eager version management, the situation reverses, when the transaction commits, it does not need to copy the values to the memory since these are already there, requiring no extra work. Aborting the transaction, on the other hand, requires the transaction to go over their undo-log, and replacing the transaction's tentative values with the older version.

In summary, with eager version management, commits are cheap but aborts are expensive, while the situation is the opposite for lazy version management. This makes eager version management a better suit for workloads that are not very contentious in nature, where transactions are likely to commit without any conflict. Lazy version management, on the other hand, makes more sense being used on workloads where transactions are more likely to abort, since rolling back the transaction is a much more lightweight operation than on the alternative version management method.

Conflict Detection Time

Conflict detection refers to when the conflicts are first detected by the underlying TM system after they occur. Again, like on version management, this can be done either lazily or eagerly.

With eager conflict detection, conflicts are detected and further resolved immediately after they happen. This can be done by having an access by a transaction to cause a modification in the global metadata, that can therefore be seen by every other transaction in execution. This way, whenever a thread tries to access some location, it first checks the global metadata to know if some other thread has also made an access to that location, knowing immediately if a conflict has occurred.

With lazy conflict detection, the transaction in question does not check for possible conflicts upon access, but rather at commit time. The transaction proceeds as normal, adding entries to their respective logs, and then at commit time, it will go through a validation process where it compares the read log with the current values in memory, to ensure that no other transaction has changed these since they were read. At commit time, the transaction also tries to acquire the rights to write its write-log back to the memory, for instance, in a lock-based TM system, the transaction would acquire ownership of all the locks. If either the process of validation of the read set or lock acquisition failed then a conflict has occurred, and must be resolved.

Furthermore, TM implementation might combine eager and lazy conflict detection based on the conflict type. Let us take for instance a lock-based TM implementation that detects write-write conflicts eagerly, i.e., via attempting to acquire a lock as soon as the write operation is called. If it finds the lock busy then the transaction is said to have conflicted with another concurrent transaction. However, write-read conflicts can be detected lazily, i.e., via testing the freshness of the value previously read. If the value changed since it was read, the transaction must abort.

Conflict Detection Scheme

Another design alternative when implementing a TM is to decide how exactly is the TM system going to know that a conflict occurred. These are usually done in one of two different manners: value-based validation or timestamp-based validation.

In value-based validation, the transaction stores the actual that are read from the memory in the respective log, along with the location's address. When it is time to perform validation of the data sets, the transaction compares the value in the transaction's log with the value that is stored in the shared memory. If any of the values mismatch, then a conflict took place and must be resolved.

In timestamp-based validation, there is a global clock that marks the number of committed transactions so far. Associated with each memory position, there is a timestamp that marks the value of that clock when it was last updated. Transactions also take note of this global clock when they begin their execution, making it their own timestamp. Transactions can only read values from the memory that have a timestamp lower than their own, i.e., it cannot read values that were updated after the transaction has begun its execution. A transaction is invalidated when it finds that one of the entries of the read set have a higher timestamp value than the transaction's timestamp, which indicates that that value has since been updated.

Another type of time-based validation is with local version numbers, rather than a global clock. In this scheme, each location has a version number that is incremented only when a transaction updates that memory location. As with the global clock implementation, this TM detects a conflict when performing validation, by finding that the version numbers in transaction's local log are different from the one in the shared memory. This type of implementation reduces the contention on updating the global clock, but it requires that a validation operation is performed after every read to ensure a consistent view of the memory.

Conflict Granularity

Granularity refers to the level at which conflicts are detected. In Hardware Transactional Memory (HTM) systems, these are typically detected at the level of a cache line, whereas in STM systems, this granularity can vary from a whole object to that of a single word (4-8 bytes).

This implies that TM systems have inherently some amount of *false conflicts*, where transactions may conflict even though they have made accessed to different memory locations, albeit very close in the address space.

Techniques such as value-based validation can be used to correctly identify these *false conflicts*, sometimes being used as a fallback mechanism to time-based ones [26].

Read Visibility

Reads that are made by the transaction can be visible or invisible. A visible read is one that by performing the read, the transaction will alter some value on the global metadata, and so every other transaction in the system can see that there was a read by some thread in that memory location.

On the opposite situation, an invisible read is one where no change happens in the global metadata, and the changes made by that read are only visible to the transaction that performed it.

Visible reads enable an earlier conflict detection, since there is some information that a reading happened for that location, the conflicting transaction can immediately tell that its access will be causing a conflict, and so the conflict can be resolved immediately.

However, read visibility may cause an increase in memory traffic and higher memory usage to accommodate for the extra metadata needed to track reads. In cases where that is undesirable, invisible reads can be used since they make reading a more lightweighted operation.

Multi-versioning concurrency control

MV schemes have been long investigated in the literature both in the database [37–39] and TM community [32, 40, 41]. As the name indicates, MV schemes keep multiple versions of a transactional data item, rather than just the most recent version, i.e. what occurs on single-versioned schemes. When a data item is updated, a new version is created and stored along previous versions, instead of simply overwriting the version that was already there.

The main goal of this approach is to completely isolate read-only transactions from update transactions, by allowing transactions to observe a constant snapshot of the shared memory space that is the same as when the transaction began execution. This is achieved by having read operations only retrieve the most recent version of a given data item that was already present in memory at the time the transaction started.

From a theoretical standpoint, the key benefit of MV approaches are: $i)$ ROTs can execute without ever aborting, as they are guaranteed to always observe a consistent snapshot by resorting to “older” versions; $ii)$ for analogous reasons, update transactions are never blocked or aborted by concurrent ROTs. From a pragmatical perspective, MV schemes are well-known to provide superior performance in workloads characterized by long-running read-only workloads [29, 30], which are prone to suffer from starvation in single-versioned approaches (due to the high likelihood to conflict with concurrent update transactions).

2.1.2 Correctness Criteria for Transactional Memory

From a user's perspective, a TM should provide the same semantics as critical sections: transactions should appear as if they were executed sequentially. However, a TM implementation would be inefficient if it never allowed different transactions to run concurrently.

Reasoning about the correctness of a TM implementation goes through defining a way to state precisely whether a given execution in which a number of transactions execute steps in parallel "looks like" an execution in which these transactions proceed one after the other. The role of a correctness criterion in this context is precisely to capture what that notion actually entails.

Linearizability [42] is safety property that describes shared objects. In the context of a TM system, linearizability says that every transaction should appear has it happen in a single and unique point of the application's execution timeline. However, linearizability only cares about the finished result of the transaction, but transactions are not a black box operation but rather an intrinsic part of the program, where every operation that is made inside the transaction is important and accessible. A correctness criteria should also take into account these small inner operations inside the transactions, and not only the end result of the transaction as a whole.

Serializability [43] is a very common correctness criteria for database transactions, on which the TM paradigm is inspired on. Serializability expands on the previous criterion by stating that for a history of transactions, all the committed transactions must make the same operations and receive the same responses as if the were executed in some sequential order. But, serializability, even when considered to account for real time ordering, does not say anything about the state of the memory that is accessed by live transactions, even those that may abort eventually.

A transaction that accesses inconsistent states in memory can potentially lead to serious problems, even if that transaction is later aborted due to its inconsistent accesses. Guerraoui and Kapalka illustrate such an example of these problems. Consider two shared variables, x and y, that are governed by the rules $y = x^2$ and $x \geq 2$, and the initial values of x and y are 4 and 16, respectively. Assume now that a transaction T_1 alters the values of x and y to 2 and 4, and then commits its changes. A concurrent transaction T_2 reads the old value of x (4) and the new value of y (4), and now, in the context of the T_2 the rules are not being followed.

The transaction T_2 can eventually be aborted, since the error will be caught when read validation is performed. But what if T_2 tries to compute some operation like $1/(x - y)$ before any validation is performed? A "divide by zero" exception is thrown and that could potentially crash the whole application. This small and simple example illustrates the dangers of dealing with inconsistent memory states, even if the consequences end up not being as dramatic as a full application crash, they are nonetheless unacceptable, which is why a new correctness criterion is necessary for implementations of TM.

Guerraoui and Kapalka propose the concept of opacity [44] for TM systems. Opacity captures the intuitive requirements that $i)$ all transactions that commit must appear as if they happened in a single, indivisible point during the application's lifespan, $ii)$ No operation that is performed by an aborted transaction is ever visible to any other transaction, and $iii)$ every transaction must observe a consistent view of the memory. Opacity is the correctness criterion over which most current TM implementations are built around.

2.2 Graphic Processing Units

GPUs are hardware accelerators that were first developed by NVIDIA in 1999. These showed up due to a high demand in the market for fast real-time 3D graphic processing, mainly for the gaming industry. The demand for high-definition 3D graphics has only rose over the years which has made the GPU evolve into a device with an extreme amount of raw processing power and bandwidth that is much higher than most of nowadays CPUs.

The first GPUs had very specific architectures that while it made them very optimized accelerators for graphic processing, they could not be used for other types of workloads. However, the release of the Tesla microarchitecture [45], which featured a unified architecture where different stages of the rendering pipeline had more or less the same capabilities, allowed for a more flexible use of the hardware.

This new microarchitecture, along with the release of CUDA in 2007 and OpenCL in 2009, both programming models for the GPU, have made it a much more generalized computing device which has since been used in diverse fields such as image processing, general signal processing, computational biology, machine learning and even computational finance.

CUDA programming model [46] can only be used for CUDA-enabled devices, which are mainly NVIDIA's GPUs. It is made such that is it possible to work with languages such as C, C++ or Fortran, which make CUDA significantly easier to work with for specialists, unlike other previously existing APIs such as OpenGL.

Note that as mentioned before, coding parallel programs for multi-core processors is a more complex task that writing a serial one. The problem is exacerbated in GPU, because now the 4, 8, 16 cores of a CPU processor are turned into thousands of GPU cores. Furthermore, the architecture of the GPU is intrinsically different from the one on the CPU, which is why many algorithms that were originally written for the CPU require a complete re-engineering when ported to the GPU.

2.2.1 NVIDIA GPU Architecture

The NVIDIA GPU architecture is built around an array of multithreaded Streaming Multiprocessors (SMs). Each of these SMs contains several CUDA cores and it is this core that represent one thread of the application. When the host dispatches a program to the GPU, the blocks of the kernel invocation are enumerated and distributed to the SMs available. There can be more than one block associated with a given SM, but the same block will be always executed on the same SM, until the end of their execution.

A SM is designed to execute hundreds of threads concurrently. To manage such a large amount of threads, it employs a unique architecture called Single-Instruction, Multiple-Thread (SIMT). Unlike CPU cores, instructions are issued in order and there is no branch prediction or speculative execution.

The multiprocessor creates, manages, schedules, and executes threads in groups of 32 parallel threads called *warps*. Threads composing a *warp* start together at the same instruction of the program, but they have their own instruction address counter and register state and are therefore free to branch and execute independently.

The SM partitions the thread blocks that were distributed to itself into *warps*, and each *warp* gets scheduled by a *warp* scheduler for execution. The *warps* are always formed by consecutive threads of the block, for example, a possible *warp* contains the threads with the IDs from 0 to 31, but never having the first 16 threads with the IDs 0 to 15 and the remaining from 32 to 45.

Every *warp* executes the same instruction at a time, also called a lock-step execution. Threads that that may be on a different conditional branch, and should not execute the instruction that the *warp* is performing can be masked off. Since there is only one instruction executed at a time per *warp*, maximum efficiency is achieved when the thread divergence within the *warp* is null, i.e., when all the 32 threads of the *warp* share the same execution path.

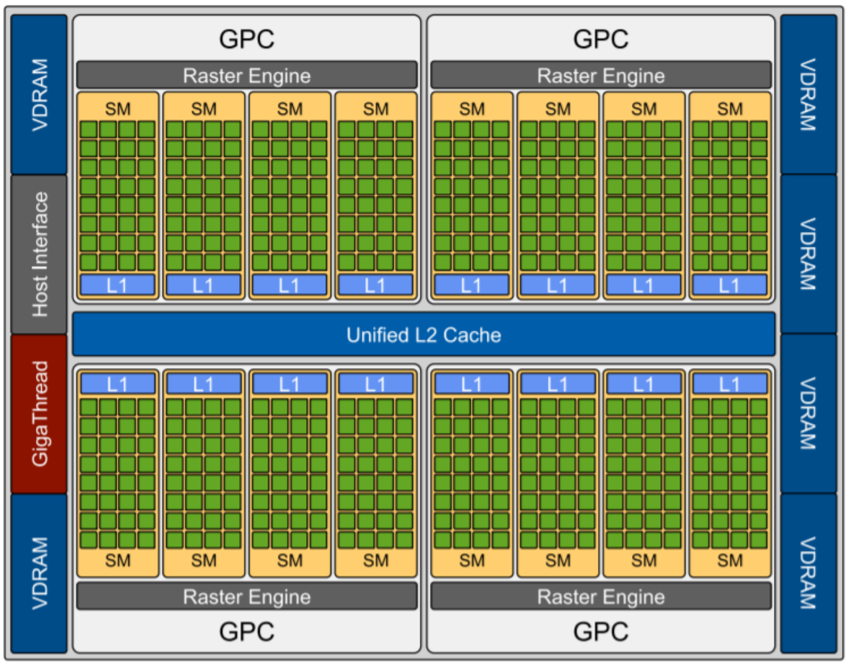


Figure 2.1: Architecture overview of the Fermi microarchitecture. Adapted from [47].

Figure 2.1 shows an overview of the Fermi microarchitecture. Despite having been replaced by newer generations, this microarchitecture is a good example to illustrate the several components that still make up the GPU today. In it, it is possible to see each SM, represented in yellow blocks. Inside a SM there are CUDA cores, represented in green squares, each will execute a single thread of the block that is assigned to that SM. Furthermore, each SM has access to a private L1 cache, a read-only texture cache, a register file and even a very fast scratchpad memory, which is also denominated as shared memory.

SMs share a unified L2 cache, as well as a global memory address space, the VDRAM, that makes it possible for threads in different thread blocks to communicate. Blocks are assigned to the SM by the GigaThread, and inside the SM there is a *warp* scheduler that determines which is the next *warp* of a block to be executed.

The architecture of the GPU has gone through a number of developments along the years. This thesis presents a brief introduction to the most significant changes of the recent microarchitectures released in the recent years, starting from the Pascal microarchitecture, which is the one used for the development of the work of this thesis.

- **Pascal**

  The Pascal microarchitecture [48] is the successor to the Maxwell class, and it was first released in 2016, with the Tesla GP100 GPU from Nvidia.

  The SMs in this class are partitioned in two processing blocks, each possessing 32 CUDA cores with single-precision, an instruction buffer, a *warp* scheduler and two dispatch units. Each *warp* scheduler (one per processing block) is capable of dispatching two *warp* instructions per clock.

  Despite having a fewer amount of CUDA cores than the predecessor architecture, it has maintained the size of the register file and the occupancy of the warps per SM.

  The whole GPU has far more SMs that previous generations which means threads across the GPU have access to more registers, and more warps and blocks can be active at the same time, when compared to previous generations.

- **Volta**

  The Volta microarchitecture [49] is the successor to the Pascal one. It was first released in 2017. It follows the same idea of partitioning the SM into processing blocks as the Pascal micro-architecture, but this time into 4 different processing blocks.

  The resources per SM remain mostly the same, with 64 FP32 cores and 32 FP64 cores, but now with the instruction buffer that was used per SM partition now being replaced with a L0 instruction cache that provides higher efficiency.

  Furthermore, there was a merger of the L1 cache with the shared memory of each SM which enables an increase in shared memory capacity to 96 KB per Volta SM, compared to 64 KB of the Pascal SM.

  Arguably the biggest change with the Volta microarchitecture comes with the introduction of the Independent Thread Scheduling, which essentially allow the GPU to keep the state on a per thread basis. This allows the GPU to yield execution of a single or more threads, either to save resources or wait for the data to be produced by another. A schedule optimizer allows threads to diverge and reconverge at a *sub-warp* granularity, since this optimizer determines how it can group active threads of the same *warp* into SIMT units.

- **Turing**

  The Turing microarchitecture [50] was first released in 2018 as the successor to Volta. The new SM incorporated many of the Volta GPUs features, mainly the partitioning into 4 processing blocks, each with 16 FP32 Cores, 16 INT32 Cores, two Tensor Cores, one *warp* scheduler, one dispatch unit and a 64K register file.

Turing's core execution of the datapaths were revamped to add a second parallel execution unit next to every CUDA core that executes simple integer operations that can used for addressing or fetching data, in parallel to the already existing FP pipeline that previously sat idle when these operations were executed.

Furthermore, Turing improves the main memory, cache memory, and compression architectures to increase memory bandwidth and reduce access latency, with the new GDDR6 memory subsystem. It also doubles the size of the L2 cache from 3 MB to 6 MB, whilst also improving its bandwidth.

- **Ampere**

  The Ampere microarchitecture [51] is the Nvidia's most recent deployment, being released in 2020. This architecture takes the ideas already implemented in previous generations and improves upon then, allowing it to deliver performance of up to 1.7$\times$ faster in traditional raster graphics workloads and up to 2$\times$ faster in ray tracing when compared to the Turing Architecture.

  The second parallel datapath added in Turing was limited to integer operations. Ampere includes FP32 processing on both datapaths, doubling the peak processing rate for FP32 operations. Ampere also continues to expand on the memory subsystem by utilizing a new high-speed GDDR6X memory, which is the next big advance in high-bandwidth GDDR DRAM memory design. GDDR6X is the next big advance in high-bandwidth GDDR DRAM memory design. GDDR6X preserves the same data access granularity and memory module size as the GDDR6 memory standard, but improves data rate and transfer efficiency in many ways.

  Finally, Ampere adds new asynchronous copy, asynchronous barrier, and task graph acceleration. Async copy improves memory bandwidth efficiency and reduces register file bandwidth, and can be done in the background while an SM is performing other work. Hardware-accelerated barriers provide more flexibility and performance for CUDA developers, and the task graph acceleration helps optimize work submissions to the GPU.

2.2.2 CUDA Programming Model

Application's code can be typically divided into two distinct structures, an irregular one, which represents a large fraction of the code, such as memory initialization, IO interactions or extracting results from the processed data; and an regular code structure that is usually a small fraction of the code, typically only a few lines in length but it is where the program spends the bulk of the execution time.

It is this regular code section, that typically features a great level of latent data level parallelism, that can be efficiently exploited by the GPU, while the rest of the irregular code often continues to be executed by the CPU.

Nvidia's CUDA [46] is a scalable programming model that allows programmers to offload these type of regular code structure to the GPU device, and the retrieve back the results from the global memory of the GPU. CUDA is based on the C language and works for the any CUDA-enabled device, i.e. any of microarchitectures of the Nvidia's GPU.

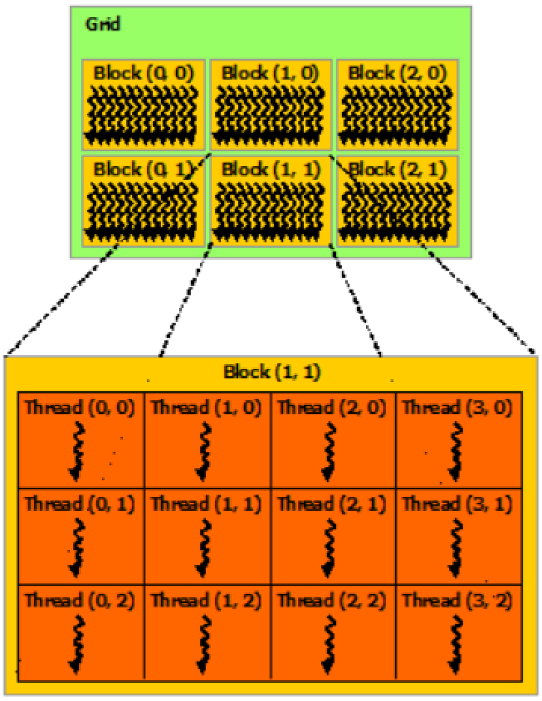


Figure 2.2: Organization of Blocks and Threads invoked by a 2D kernel. Adapted from [46].

The basic unit of execution in CUDA is the kernel, which represents the code to be run in the GPU. The programmer has to build a function that will be launched by the kernel into the GPU, and therefore he has to keep in mind, to a certain level, how the architecture of the GPU is organized, or at least what are the common practices that one could use when programming in CPU that do not necessarily translate to a GPU execution model.

When launching the kernel, the programmer can dictate how many thread blocks will be launched by the kernel, by specifying the value of the block grid size. This value is a 3-component vector, for convenience, in case the programmer wishes to use up to a 3D grid of blocks when using the GPU to tackle the problem. Likewise, the thread block is composed of single threads that will run in the same SM, and the programmer can specify the number of threads per block, again with the possibility to represent this number in a 3-component vector. This dimensional organization of the set of blocks and threads is illustrated in Figure 2.2, where both blocks and threads are organized in a 2D manner. (The third component of each vector is always 1, in this case). A thread block has a limit of 1024 concurrent threads, but the number of blocks in a grid is not limited.

Thread blocks are required to execute independently: It must be possible to execute them in any order, in parallel or in series. This independence requirement allows thread blocks to be scheduled in any order across any number of cores, enabling programmers to write code that scales with the number of cores. If one wishes to maintain a specific ordering among different blocks, then this is only possible by having separate kernel launches, as the threads blocks of a second kernel are only dispatched to available SMs after all the thread blocks of the first kernel have been dispatched as well.

The primitives $threadIdx$ and $blockIdx$ are values available from inside the device code, that help organize the threads inside the blocks and as a whole across the GPU. The $threadIdx$ is the thread identifier and is unique within a single block, while the $blockIdx$ is the block identifier, and it is unique across all blocks. With these two values, it is possible to synchronize accesses between all threads in the GPU.

All the data needs to be explicitly transferred between devices, and the communication between these, whether for issuing a simple launch command or a transfer of thousands of entries of data, is done over the PCI Express bus, which can introduce non-trivial communication overheads to applications that use the GPU to accelerate small data problems.

2.2.3 Memory Model

CUDA GPUs have several different memory spaces, each with their own advantage and disadvantage, which makes it critical to know these memory subsystems to leverage the resources that the CUDA interface exposes to the programmer. The types of memory spaces are:

- Register file, which like registers on the CPU, allow the data stored in it to be fetched immediately, and therefore make up for the fastest memory space. However, registers are typically only accessible to a single thread (with the exception of shuffle operations), and their space is limiting, which may cause spilling to the local memory.

- Shared memory, which is an on-chip memory, has a very low latency and high bandwidth, making it a very useful memory for communication between threads. However, shared memory is only accessible to threads within the same thread block.

- Global memory, which contains a lot more available space than other types of memory. It is accessible to all threads in the system, which makes it the only means of communication between threads that are not in the same thread block. Since it resides in the device memory, it is a high-latency and low-bandwidth memory, which makes accesses to it a slow process.

- Local memory, is a reserved space in the global memory and therefore have the same high latency and low bandwidth as global memory accesses. This type of memory is private to a single thread, and it is typically used by the compiler for large private data structures that can not be accommodated in the register file.

This memory hierarchy within the GPU is illustrated in Figure 2.3, where is possible to see what are the accesses and communication that each level allows. The global memory can even be used for inter-grid communication, unlike the other memory subsystems.

When a warp executes an instruction that accesses global memory, it coalesces the memory accesses of the threads within the warp into one or more of these memory transactions depending on the size of the word accessed by each thread and the distribution of the memory addresses across the threads. This coalescing of memory transactions can therefore be used to mask away some of the high-latency that these fetches typically entail.

Besides these memories subsystems, there are still the texture and constant memories, which are typically faster memories than the global memory, but since they are read-only their benefits are very application dependent.

The GPU has a weak memory consistency, which means that different threads can observe a different order of updates to a memory location. This sort of weakly-ordered access can happen at any level of the memory, be it shared or global. If the order of accesses is important for the application, then the programmer must use memory fences to ensure a consistent ordering for every thread.

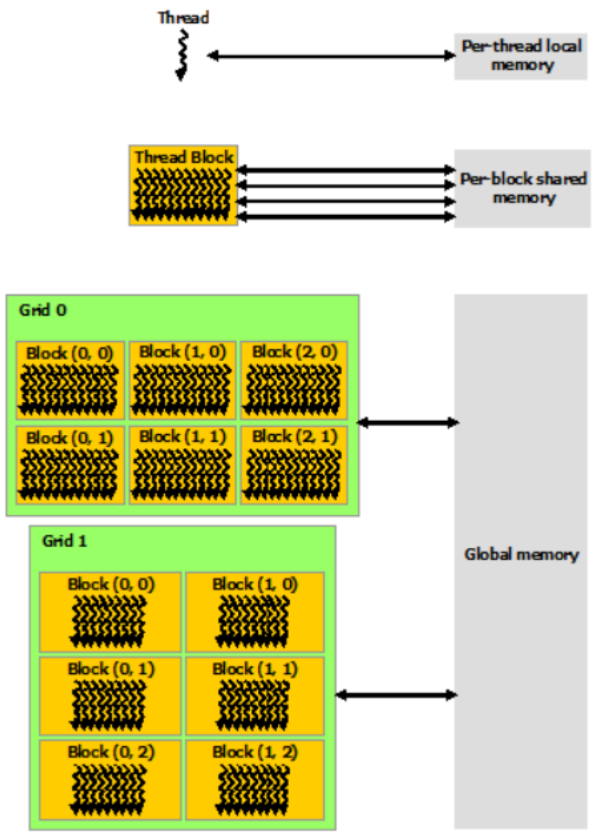


Figure 2.3: Memory Hierarchy on the GPU. Adapted from [46].

Finally, CUDA also provides support for solving data race conditions, through the use of Atomic operations, which ensure that accesses to memory locations through these Atomics are only performed successfully by one thread at a time. This sort of operation is invaluable when building applications that require synchronization, such as lock-based ones, since these require a guarantee of mutual exclusion when accessing their locks.

2.3 Concurrency in the GPU: lock-based

While GPUs have traditionally focused on applications with regular parallelism, irregular applications in range of domains (including machine learning [2], graph manipulations [1], dynamic programming [52], data structures [3–5]) have been accelerated using fine grained locking schemes. Simplifying and optimizing the execution of fine-grained synchronization are some of the key motivations at the basis of the independent thread scheduling introduced by NVIDIA since the Volta architecture [53].

In the literature on optimizing fine-grained locking, the idea of exploiting scratchpad memory is not new [52, 54]. The work developed by Wang et al. [54] proposed the use of a software client-server architecture that that delegates the execution of critical sections on GPUs to dedicated server thread blocks, which can then synchronize the access to the critical sections they are responsible for via scratchpad memory. The remaining (i.e., synchronization-free) code of the application is executed by client thread blocks that communicate (i.e., submit requests for executing critical sections) via a communication library that makes efficient use of off-chip/global memory by employing optimizations that reduce the overhead of message passing operations and that promote coalesced memory accesses. The server thread blocks make use of a dedicated warp that functions as the middleman between client and server nodes by extracting messages from the global message channel and distributing the work to the remaining warps via scratchpad/shared memory.

CSMV applies and specializes this client-server execution model in order to accelerate the commit process of a multi-versioned STM scheme. This raises a number of new challenges, such as how to alleviate the server load by shifting to the client-side part of the commit logic and how to effectively parallelize the server-side commit procedure.

2.4 Transactional Memory on GPUs

GPGPU research typically focus on very data regular programs, where every thread has to deal with a limited amount of data whose location is already predetermined when the program launches. In these types of applications there is very little or non-existent sharing of data between threads, and therefore there is no need to synchronize accesses through the use of a synchronization mechanism like locking or transactional memory.

GPU have been shown to produce incredible speed ups for programs like this, but what about data irregular applications? A common strategy to tackle applications with data or flow dependencies on the GPU is to try to completely re-engineer the CPU algorithm, as it is commonly believed that speed-up can not be achieved without this significant effort.

Nelson and Palmieri [19] shows that an in depth re-engineering of the algorithm may not always be the best performing solution, and that a simple synchronization based on locks can achieve very significant speed ups versus well-engineered and parallel solutions on both synthetic and real datasets.

However, the problems with locking that were mentioned earlier in the chapter are now even more complex in the GPU. Fine-grained locking on the GPU is more prone to deadlocks and livelocks due to the higher number of concurrent threads, as well as a weaker memory consistency than on the CPU. Take for example, two threads T_A and T_B, and two locks L_1 and L_2. Assume that both threads try to acquire the locks, but in different order: T_A acquires L_1 and T_B acquires L_2. So far, no problems have arisen, but when T_A tries to acquire L_2 or T_B goes for the L_1 lock, they will find them already taken, and they either wait for the lock to become free, which will never happen because the other thread will also be waiting, causing a deadlock, or they could abort, but restart the execution immediately leading them to a state where both threads are doing work, but none are actually progressing, which is called a livelock.

Another alternative locking mechanism could be coarse-grain locking: all the shared memory region are protected by the same single lock. Coarse-grain locking is usually easy to implement, since all that is needed is to toggle the same lock bit when entering the critical region, and toggle it again when exiting. No deadlocks or livelocks would come from this, but the downside would be the serialization of all the shared objects accesses which on a SIMT architecture like the GPU would kill any performance improvement.

Furthermore, these locks can only exist on the lowest level of the memory system, the global memory, and could never make use of any of the intermediate cache levels, since the global memory is the only address space that can be accesses by all the threads of the GPU.

The TM abstraction here could move all these concerns with memory consistency and deadlocks away from the programmer, and making them the responsibility of the TM implementation. That is why several implementations of TM on the GPU already exist in the literature, some in software and also in hardware. Over the next sections, we will go over the state-of-the-art of TM implementations for the GPU, with added focus on STM implementations, since the focus of the work is to create a new STM for the GPU.

2.4.1 Block-STMs

Cederman et al. [25] proposed 2 different STM for the GPU in 2010. These were the first proposals of STMs for this device, and have since inspired several other implementations, whether in software or in hardware. In regards to the progress guarantees of each of these, they are called Blocking STM and Non-Blocking STM, respectively.

The first STM proposed offers no guarantees of overall progress for the application, even in the absence of any conflicts between transactions. The authors argue that selecting this kind of progress guarantees allowed for a simpler design and possibly an even more efficient implementation. The downside of this approach is that it makes the STM much more dependent on the warp scheduler of the GPU. This scheduler could potentially swap the lock owners, and repeatedly schedule transactions that were waiting for the lock to release. However the authors claim that they did not experience such problems during their implementation.

The second STM is based on a previous STM for the CPU, and the progress guarantees offered by this implementation are obstruction-free, which means that a transaction will always succeed if it encountered no conflicts with other transactions. This STM tries to acquire all the locks at commit time, while at the same time announcing the values that it is going to write. Conflicting transactions at this point can either steal the locks and use the announced values instead, or abort the original transaction entirely.

Besides these differences at commit time, both STMs are similar in the remaining aspects: both use version locks with invisible reads, they use a local log to store the writes before committing them and the conflicts are detected at commit-time. It is also important to mention that it is a whole thread block that represents a transaction, rather than a single thread as is the case of the next STM implementations of the section. This may lead to believe that there is cooperation between threads within the thread block to realize the transaction, but further inspection leads to the conclusion the authors have used a single thread per thread block, which in the end hampers significantly the scalability of these solutions.

2.4.2 GPU-STM

Xu et al. [26] introduced GPU-STM in 2014. It implements three novelty ideas, that take into consideration the characteristics of the GPU. The first of these ideas is the combination of two different validation schemes that already exist in the STM for CPU literature, time-based validation similar to the one used in TinySTM [21] with a fallback mechanism that uses value-based validation as with STM like NORec [23]. Time-based validation is used at the core of the STM, and when this scheme fails, it checks it again but using value-based validation. This allows the TM implementation to avoid false conflicts that can arise from the granularity of the conflict detection not being fine enough.

GPU-STM also does not acquire locks immediately as needed, instead storing the addresses of the locks that it will need in a already-sorted lock log, a technique which the authors call encounter-time lock sorting. These addresses are inserted in a specific order that will be the same for every transaction. When it is time for the transaction to commit, the locks are acquired in that specific order, which may not have the same ordering on which these locks were stored in the log. This allows GPU-STM to be deadlock and livelock free, even with no backoff mechanism in use, since there is a global order for lock acquisition.

The third implementation idea is to merge the write and read logs of the transactions of the same warp in the global memory. They will still be individual logs for each transaction, but they will be stored in a close range of each other on the address space. This implementation looks to exploit the coalescing mechanism when retrieving data from the global memory.

2.4.3 ESTM, PSTM and ISTM

Holey and Zhai [27] proposed three different implementations of a STM in their paper. They all follow the same baseline idea, while changing some aspects and then evaluating their performance. The implementations are named ESTM, PSTM and ISTM, with ESTM being the baseline STM upon which the other two will improve.

Of the latter two implementations, PSTM and ISTM, achieved the best results, neither being always the top performer, which indicates that these implementations are targeted at different types of workloads. The baseline STM, ESTM, never outperformed either of the other two for the several benchmarks used in this paper. The three STMs use in-place speculative writes, or eager version management, as well as a eager conflict detection for write-write conflicts. Furthermore, all the transactions use an exponential backoff method as to avoid livelocks.

ESTM makes use of a metadata structure, called the shadow memory, which tracks, for each memory location that is shared amongst transactions, all the access information. Each of the entries in the shadow memory store information that allow any thread that accesses it to know the current state of that location, from the number of threads that have read the location, to the control bits that mark an uncommitted transaction is reading, modifying or locking the respective location, that information is all accessible through the shadow entry for the desired location. This allows the underlying STM to detect every type conflict eagerly, whether these are read-write or write-write in nature, and abort the conflicting transaction immediately.

At commit time, the transaction has to clear all the data that it entered in each of the shadow memory entries that it either read or write, since both of these access modes cause a modification in this global structure. In case the transaction is aborted, then it further has to restore the original values to the locations in global shared memory, since this STM is using in-place writes.

PSTM takes a pessimistic approach to conflict detection, where it does not make any distinction in the accesses to the shadow memory: writes and reads are treated in the same manner. It pessimistically assumes that if reads are conflicting, then writes will also conflict.

This approach will lead to a much higher ratio of false conflicts among transactions, however the authors argue that this conflict detection scheme has its benefits: the first and most broad one is the fact that the conflict detection has much simpler mechanism, since now the shadow memory entry needs only to store which transaction is accessing the location, without need to allocate extra parameters to differentiate between reads and write. The second benefit is a lot more workload dependent, and it is an earlier conflict detection in workloads where the locations that are read will be typically written to in the future by the same transaction, reducing wasted work for these types of workload.

The third STM implementation, ISTM uses invisible reads, unlike the previous implementations of this paper. To be able to use invisible reads, ISTM makes use of versioned locks as the entries of the shadow memory. In this scheme, the reads that transactions speculatively make remain hidden to other transactions, i.e., reads do not modify any kind of globally available metadata, but writes still do, by changing the lock bit of the versioned locks, which means that writes are still detected eagerly.

At commit time, ISTM needs to validate its read set, where the transaction compares the version numbers that were stored on the read set with the values that are currently present in the version lock of the actual location in memory. A change in the version number, or a lock bit set to positive in this locks, signifies that a concurrent write operation took place or is taking place at that moment and therefore the transaction must abort, undo the changes that it made and retry the transaction again.

2.4.4 PR-STM

PR-STM [28] is the latest STM developed for the GPU. It was proposed in 2015, at around the same period of the previous 3 STMs. Like with many of the STMs for the GPU seen so far, PR-STM is lock-based, making use of versioned locks both for validation or conflict detection. In essence, PR-STM is similar to the ISTM from the previous paper, with the exception that version management is done lazily, i.e., the writes are made to a private and local write buffer to the transaction rather than writing them directly in memory.

PR-STM is also more prone to false conflicts than ISTM, since it uses an hash bucket to map from locks to words in the shared memory regions, which signifies that a single lock can represent more than a single word in the memory. However, such lock scheme also reduces the overall memory usage by the STMs's metadata, which might be preferred in cases where the memory is limited.

The novelty of the work presented in PR-STM is the new form of contention management. Every other implementation in the GPU, with the exception of the second variation of the Block-STMs, used a contention management scheme where the transaction that found the existing conflict would be the one that would be aborted. This type of contention management is called timid.

PR-STM however, implements a contention manager where transactions are attributed a varying level of priority. This priority attribution is the responsibility of the programmer, and remains the same throughout the course of transaction's lifetime, hence a static priority.

In PR-STM locking is done in two stages. The first, happens as soon as the transaction writes to a location, the transaction acquires a pre-lock, in a similar way as ISTM eager lock acquisition. If a different transaction then also tries to acquire the same pre-lock, then one of two of possible outcomes happen based on the priority of the transactions involved in the conflict, either the *attacker* transaction has a lower priority, in which case the transaction should abort and retry as normal, or the *attacker* transaction has a higher priority than the current owner of the pre-lock, in which case the transaction will steal the pre-lock for itself.

Afterwards, at commit-time, the transaction will have to perform validation on the read-set, same as with ISTM, but will also need to check if it is still the owner of all the pre-locks that it has acquired during its execution. So, a transaction only really knows whether a pre-lock was stolen or not at commit-time. On the second stage of the locking process, the transaction tries to convert all its pre-locks into the actual locks of the memory location. Should it be successful, then it proceeds to copy its write log to the respective positions in memory. If this lock conversion is unsuccessful, then it means that some other transaction stole one of the pre-locks, and so the transaction should free all the metadata accumulated so far and restart from the beginning.

It is important to note that, this lock stealing mechanism only occurs for the pre-locks, and once the actual lock is acquired by the transaction, its ownership belongs to this thread until it willingly releases the lock, by either successfully committing or because it had to abort due a failure to acquire a subsequent lock.

2.4.5 Analysis of GPU TM implementations

Table 2.1 shows a revision of the STM implementations for the GPU shown in the previous section. The table tries to organize the STMs according to the most common design decisions when building a TM, however, many implementation specific decisions can not be correctly compiled into such a table due to the singularity of the decision made.

Table 2.1: Revision on existing TM implementations for the GPU.

	Confl. Det.	Writes	Read Visibility	Granularity	Synch scheme
Blocking STM	Lazy	Deferred	Invisible	Object	Lock-based
Non-Blocking STM	Lazy	Deferred	Invisible	Object	Lock-based
GPU-STM	Mixed	Deferred	Invisible	Word	Lock-based
ESTM	Eager	In-place	Visible	Word	Lock-based
PSTM	Eager	In-place	Visible	Word	Lock-based
ISTM	Mixed	In-place	Invisible	Word	Lock-based
PR-STM	Mixed	Deferred	Invisible	Multiword	Lock-based

The STM research presented showcases some implementations of TM in the GPU that demonstrate that it is feasible to provide support to transactional execution in a system like the GPU, a system that is very sensible to overheads due to its inherently light weighted threads. A common trend on the STMs analysed seems to be in creating lightweight transactions as to minimize the impact that the bookkeeping of said transactions can have on the GPU.

There are some interesting and unique design decisions among the studied STMs. For instance, GPU-STM merges the write and read set of the same transactions within a warp to try and coalesce the memory transactions from the memory. Although the sets are still stored in the slow global memory, this mechanism can mask some of the latency that these fetches may have. On the other hand, PR-STM implements a unique conflict management method, unlike most of the other implementations that use a *timid* contention management approach. This method allows PR-STM to have a much smaller overhead when compared to the lock-sorting method of the GPU-STM, which induces non-trivial overhead to each operation within a transaction.

However, none of the implementations make an efficient use the hierarchical memory system of the GPU, and conduct all their operations in the global memory. From the global metadata, such as the locks, to the transaction's private logs, they are all stored in the global memory which is a slow subsystem when compared to other memory subsystem that are present in the GPU, such as shared memory.

For the GPU, there are also a few implementations of HTM. KiloTM [55] employs a value-based conflict detection and deferred writes. WarpTM [56] builds on top of KiloTM, aggregating protocol messages to reduce communication overhead. Chen and Peng have also expanded the work made on the previous implementations by first adding early conflict resolution schemes [57] and later Snapshot Isolation [58]. However, these implementations require significant hardware modifications, and are therefore not possible to use on commodity GPU architectures.

2.5 Summary

This chapter presented the general concepts of TM, in Section 2.1 and the GPU, in Section 2.2, we finished up with the different STM implementations, in Section 2.4, that constitute the state-of-the-art of STM on the GPU.

TM is a synchronization method that abstracts the complexity of dealing with lower-level lock primitives away from the programmer. This abstraction simplifies the development parallel programs, given that the programmer only needs to deal with functions that constitute an API of the TM, when he wants to access the shared memory resources of the application. We made a compilation of the most common design decision when building a TM implementation, in Section 2.1.1, highlighting the virtues and limitations of each decision. In Section 2.1.2, we go over the concept of correctness within the scope of a TM, where the opacity criterion is detailed.

In Section 2.2, we first go over the architecture of the GPU, in Section 2.2.1. The unification of its architecture has made it a very popular accelerator for general purpose computation, with a scope that goes beyond just graphic processing applications, with a raw computational power that outperforms CPU architectures. The most recent modifications to the microarchitecture of the GPU are also briefly described, followed by the programming model, CUDA, in Section 2.2.2, that can be used to launch kernels that will execute their code on the GPU. Lastly, in Section 2.2.3, we elaborate on the different memory systems that are present in the GPU, highlighting the advantages and disadvantages of each, in terms of latency of accesses and communication between threads that they allow.

Next, in Section 2.4, we briefly go over the implementations of STM that have been made on the GPU, discussing their most important features and decisions. In Section 2.4.5, we perform an overall analysis of the implementations elaborated on the Sections 2.4.1, 2.4.2, 2.4.3 and 2.4.4.

Chapter 3

Dissecting a MV STM for CPUs

Following on the research and insights presented on Chapter 2, this chapter does a more in-depth overview on the algorithm used by JVSTM, a state of the art multi-versioned STM for CPUs [29, 40]. From this algorithm, a direct porting to the GPU is created, named JVSTM-GPU. This chapter ends with a reflection on the inefficiencies that arise when straightforwardly adopting a MV STM designed for CPUs on GPUs.

3.1 JVSTM

This section presents a description of the algorithm employed by JVTSM. This STM is a state of the art MVCC implementation for the CPU that can be considered the archetype for this kind of system. Therefore, based on the design of this system for the CPU, a solution for the GPU is devised, at first a direct porting to the GPU in Section 3.2, and then a more carefully designed solution in Chapter 4.

Data structures. In JVSTM each shared object is encapsulated within a *Versioned Box (Vbox)*, which stores the the list of existing versions for that object along with the timestamps of the transaction that generated them.

JVSTM relies on two globally shared data structures:

1. a (logical) *Global Time Stamp (GTS)*, which is read by transactions upon their start to establishes which *snapshot* they should observe and that is incremented whenever an update transaction commits;
2. a list, called the *ATR*, that stores, for each (recently) committed update transaction:
    (a) the timestamp they obtained upon commit, and;
    (b) the set of VBoxes they updated.

Transaction execution. When a transaction T starts it establishes which *snapshot* it should observe by reading the GTS (which keeps track of how many update transactions committed so far). For T to read an object, T must lookup the most recent version of the Vbox that was created before T started, i.e., the version tagged with the largest timestamp smaller than T's timestamp. Transactions that, upon their start, are declared to be read-only do not need to keep track of the objects they read, since the above mechanism guarantees that they will always observe a consistent snapshot.

Update transactions, conversely, store a reference to each Vbox they read in a thread-local list, called read-set, for later validation. Analogously, write operations are tracked in a different thread-local list, called write-set. As such, both read and write operations are invisible in JVSTM, i.e., not detectable by other concurrent transactions.

Commit process. Read-only transactions are, as mentioned, guaranteed to read a consistent snapshot and simply skip the commit phase.

The commit process of an update transaction T entails the following phases:

1. *Validation*: T determines whether any of the transactions in the ATR that committed after T started updated any of the items read by T — in such case, T aborts.

2. *Insertion in ATR*: T attempts to insert its own record as the next entry in the ATR via a CAS operation. If the CAS succeeds, T add its own entry to the ATR moves on to the write-back phase. If the CAS fails, two scenarios are plausible:

   (a) one or more transactions finalized their commit during T's validation — in which case T validates against them and tries again to insert itself in the ATR;

   (b) some transaction T' inserted its entry in the ATR and is still in its write-back phase — in which case T waits for T' to complete its write-back phase and tries again to insert its own entry in the ATR.

3. *Write-back*: T adds a new version to all the VBoxes that it updated. Next, it increases the GTS, making its writes visible to freshly starting transactions, and flags its entry in the ATR to signal its write-back phase as completed — which is equivalent conceptually to releasing a lock on the ATR.

3.2 Porting to the GPU

As a first approach to this thesis, the design of JVSTM is directly ported to a GPU implementation, which I have named JVSTM-GPU. This STM maintains all the data structures of the original algorithm (VBoxes, ATR and GTS) completely in global memory, which signifies that all the expensive synchronization mechanisms are performed exclusively on using this memory subsystem, in a similar fashion to existing GPU STM implementations. JVSTM-GPU does features some minor changes from the original algorithm, due the inherently different natures between the programming models, which shall be elaborated on below.

One of the challenges when adapting the JVSTM algorithm to GPU code is that it does not allocate memory dynamically, that task is usually reserved to the host CPU, i.e. all the memory that will be used by the kernel needs to be allocated before the launch of the kernel on the CPU host side. This makes the use of linked lists, on which the original CPU implementation is heavy reliant on, unfeasible, for data structures such as the ATR and the VBoxes, since these data structures are not efficient in GPUs. To solve this problem, a circular buffer is used to support these data structures, which continuously stores new entries to the respective structure by overwriting the oldest entry present in the structure.

This however, introduces a new problem since the information that a given transaction needed, either to access an entry of a Vbox or to compare itself to an entry of the ATR during validation, might have already been overwritten by a new update to the data structure. In such cases, to ensure correctness, the transaction has to abort, since it is not possible to guarantee that the missing entry would change the outcome of the transaction. Thus, in this new GPU version of JVSTM there are a new type of spurious aborts that are not present on the CPU implementation.

Furthermore, the CPU implementation applies the write sets to the globally shared objected serially, i.e. only one transaction is on the write-back stage of the commit phase at any given moment, creating a bottleneck. In my implementation however, many transactions can apply their changes at the same time, with the only criteria being that concurrent transactions on the write-back stage must not be written to the same location of the shared data, i.e. to the same Vbox. To ensure this, the transaction will also compare its write set against the write set of transaction that are already in the record but that have not yet finish applying their changes to the VBoxes.

Finally, on the last stage of the commit, when the transaction makes its updates public by incrementing the GTS, the transactions need to follow the order of the commit that was determined previously when they were added to the ATR. A transaction T that finished their write back stage before a predecessor transaction T' will have to stand by while it waits for T' to publicize its results before T can make them visible as well, finalizing the commit stage and thus ending the transaction.

3.2.1 Shortcomings of a direct approach

A straightforward porting of the CPU algorithm to the GPU brings with it certain shortcomings in the sense that system fails to leverage some of the resources made available by the GPU or the different phases of the algorithm's execution are not appropriate to the SIMT architecture of the GPU.

Inefficient access to global data structures

Storing the GTS and ATR in global memory would be the most straightforward approach to enable device-wide access to these data structures. However, such an approach would also introduce several major sources of inefficiency. First, both these data structures are frequently modified (i.e., whenever an update transaction successfully commits). Further, during validation, transactions generate a large number of read accesses the ATR (to extract the write-sets of concurrently committed transactions) that are unlikely to be coalesced for two reasons:

1. threads in the same warp are prone to diverge, since they process independent transactions;
2. even if they do not diverge, they are expected to be concurrently accessing the same ATR entries and not contiguous ones, e.g., non-diverging threads in the pre-validation phase need to validate

against the same set of concurrent update transactions and will access, in lock-step, the same entry of the ATR.

Finally, contention on the lock protecting the ATR is strongly exacerbated in GPUs, due to the massive parallelism that they support compared to CPUs: this has a detrimental impact on the efficiency of atomic operations (CAS) needed to acquire the lock and further amplifies the overhead associated with accessing it via global memory.

Limited parallelism

Some phases of the commit process, namely the insertion in the ATR and the final part of the write-back, are executed sequentially, i.e. after acquiring a global lock. While this might be acceptable in CPUs [1], in massively parallel GPUs the negative impact on performance due to these sequential phases is strongly amplified — as by Amhdal's law, the larger the potential for parallelism, the larger the impact on performance due to executing the same sequential code.

[1]This problem has been at least partially addressed in a later versions of JVSTM [40], which uses a helping scheme to accelerate the execution in the critical section. However, this solution was only to be effective if (update) transactions read or write a large number of objects — otherwise, the overhead imposed by the helping mechanism outweighs the benefits it provides.

Chapter 4

Multi-Versioned Client-Server TM

This chapter is dedicated to presenting the design of CSMV, first by overviewing the novel mechanisms employed in order to address the shortcomings of JVSTM-GPU pointed out in Section 3.2.1, and then by providing a detailed description of the several stages that a transaction goes through from its beginning to the end of its commit phase.

4.1 CSMV Overview

CSMV builds on the JVSTM algorithm and extends it with the following mechanisms that operate in synergy to address the challenges identified in Section 3.2.1. I overview each of these mechanisms next:

- 1) Client-Server Architecture
- 2) Client-side write-back
- 3) Client-side pre-validation
- 4) Batched ATR Insert
- 5) Collaborative Validation

4.1.1 Client-server architecture

As already mentioned, one of the key factors hindering performance is the need to frequently access global metadata maintained in off chip memory. In order to enable the manipulation of global metadata via fast scratchpad memory, CSMV adopts a logical client-server architecture. Specifically, an high level overview of the architecture employed in this thesis can be observed in Figure 4.1, which includes the different steps a transaction commit phase entails. One of the available SMs executes a specialized kernel that handles solely the commit phase; the remaining SMs run the client kernel that is responsible for generating and executing the transactional logic (as well as any non-transactional code). Upon reaching the commit call for a transaction T, the client kernel uses an efficient/high-throughput message passing library to transmit to the server kernel (via off chip memory) T's read-set and write-set and request it to determine: i) T's final outcome (commit/abort) and ii) in case T committed, the timestamp reflecting its serialization order in the ATR, which is called CTS (commit timestamp).

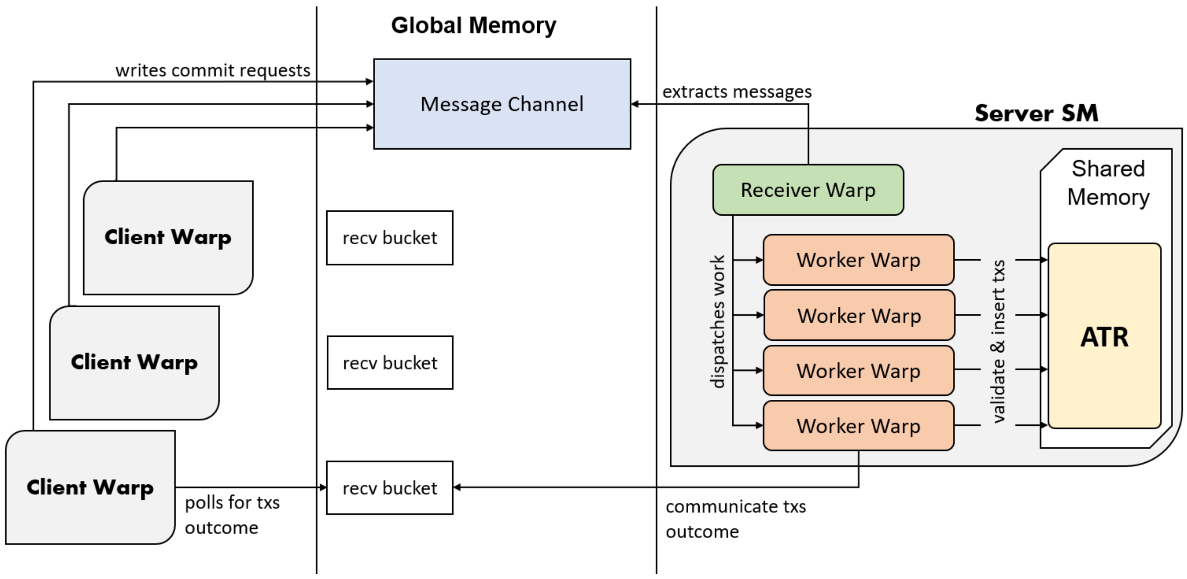


Figure 4.1: High-level view of the client-server architecture employed by CSMV.

The server-side warps are organized as follows: one warp is designated as receiver and the remaining ones as workers. The receiver warp is responsible for listening for new requests coming from the clients and dispatching them to the workers. The worker warps validate and add the transaction to the ATR in a similar fashion to JVSTM-GPU. Afterwards, they send a message back to the client with the outcome of the transaction, so that, in case of abort, the client can repeat the transaction or, in case of commit, the client can write-back the modifications if any. This communication back to the clients is performed via the use of pre-allocated receiver buckets that the worker warp writes to directly.

By using the client-server architecture, CSMV can make use of faster memory subsystems than the global memory, such as the CUDA's *Shared Memory* that cannot be easily used by the baseline as this memory subsystem is only local to each SM. Allowing the ATR to be maintained in *Shared Memory* accelerates both the transaction validation and its insertion in the ATR, increasing the maximum throughput achievable by these critical phases.

Nevertheless, there are caveats associated with the client-server architecture approach. Firstly, the amount of Shared Memory space is constraint to 48kB in our system, thus the number of entries in the circular buffer used to store the ATR list is relatively small, which can lead to more spurious aborts than on JVSTM-GPU, that has room to accommodate larger ATR sizes, due to old entries being overwritten more often. It is worth noting that read-only transactions do not need to be inserted in the ATR, which alleviates this issue in read dominated workloads.

In order to mitigate spurious aborts of update transactions, three variants of the algorithm are considered, each differing in how much of the ATR is stored in the scratchpad memory: **sh** stores the ATR in its entirety in *Shared Memory*, **ts** stores the timestamp of the transaction in *Shared Memory* while moving the write sets of committing transactions to the global memory, and **gb** that has the ATR stored in global memory, in a similar fashion to JVSTM-GPU.

Secondly, the server can easily become the bottleneck given the centralized nature of this approach, so it is desirable to offload work to clients and/or streamlining the server activity.

4.1.2 Client-side write-back

As already mentioned, CSMV offloads the write-back phase to the client side. Note that the write-back phase is executed in JVSTM after acquiring the lock on the ATR. Thus, shifting this phase to the client allows not only for alleviating the load on the server. It also reduces the duration of the sequential part of the commit process, allowing the server kernel to attain higher parallelism levels.

Note that, from the perspective of correctness, allowing the write-back phase to be executed concurrently by multiple client threads raises a non-trivial issue: the order with which the clients apply their updates should not contradict the order with which their corresponding transactions are serialized in the ATR. I tackle this issue as follows:

1. Extend the transaction validation to check also for write-write conflicts, which precludes the possibility that two concurrent transactions can ever update an item in common. In turn, this prevents that the updates applied by two concurrent clients executing their write-back phases can ever contradict the serialization order defined by the ATR. As a matter of fact, realistic existing workloads tend to have no "blind writes" (if a transactions write a data item, it also reads it) [33], in which case the existence of a write-write conflict between two transactions implies also a read-write conflict. In such a case, extending the validation to detect also write-write conflicts does not introduce any additional aborts. Indeed, if one can a priori exclude (e.g., via static code analysis techniques) the possibility of blind writes, this additional validation step can be safely omitted.

2. Once a client completes the write-back for a transaction T, let it increment the GTS (which effectively makes the transaction's updates externally visible) only when the write-back of all the transactions serialized before T has completed, i.e., when $T.CTS = GTS - 1$. This ensures that, if a transaction, upon its start, obtains a snapshot associated with some value of GTS, all the update transactions included in that snapshot have completed their write-back phases.

4.1.3 Client-side pre-validation

In order to streamline the server-side commit process, the transactions that execute within the same warp on the client side are validated to detect any intra-warp conflict. To this end, warp-level primitives are exploited (e.g., shuffle operations) to efficiently exchange the read-sets and write-sets of the transactions within the same warp. This pre-validation phase ensures that the batch of transactions submitted by a client warp to the server have no mutual conflicts, which allows the server to focus solely on checking between conflicts among concurrent transactions generated by different client warps.

4.1.4 Batched ATR insert

The client-side pre-validation opens a new opportunity to optimize the insertion of the transactions in the ATR. Since the transactions processed by the same client warp are guaranteed not to conflict among themselves and are submitted as a single batch to the server, their insertion in the ATR can be performed at once for all of them. This brings two main benefits: $i)$ it lowers the frequency of manipulation of the ATR via CAS, which in turn reduces the likelihood of contention among concurrent CAS operations; $ii)$ it amortizes the overhead of lock acquisition across all the transactions generated by the same client warp.

In more detail, the message-passing protocol between client and server was designed to ensure that a worker warp in the server is requested to process concurrently a batch of transactions generated by the same client warp. After validating the batch, a leader thread is responsible for reserving (via a single CAS) space in the ATR to insert the transactions being processed by the entire worker warp. Once the CAS succeeds, the leader notifies the rest of the warp. Each thread then proceeds with the insertion of a different transaction in the corresponding pre-reserved slot in the record.

Note that I exploit the fact that transactions of the same client warp are processed in batched mode on the server side also to further optimize the client-side write back phase: since the committed transactions of the same warp are assigned consecutive commit timestamps by the server, the client warp can make the updates of all its committed transactions publicly visible all at once, instead than individually. This is achieved by incrementing the GTS only once by a factor N, where N is the number of committed transactions in the batch, rather than N times by a factor 1.

4.1.5 Collaborative validation

Validation is definitely the most computationally intensive operation of the commit phase executed on the server-side, since, recall, it requires checking for any intersection between the read-set and write-set of the transaction being validated and the write-set of (a potentially large number) of concurrently committed updated transactions.

Optimizing the validation phase is thus crucial to maximize server side performance. To this end, a collaborative validation scheme was devised that aims at increasing the locality of accesses to off-chip memory by promoting coalesced memory accesses. Specifically, instead of having each thread of a worker warp validate a distinct transaction independently (as typically done in STMs for CPUs), we let them cooperatively validate the same transaction as follows. We assign to each thread t in a worker warp the same entry, noted e_t of the read-set and write-set of a validating transaction T; each thread is then responsible for checking where e_t is included in the write-set of any concurrently committed transaction. Recall that the write-sets of committed transactions are maintained in the server-side on-chip scratchpad memory, whereas the read-sets and write-sets of validating transactions are maintained in the off-chip memory (since they need to be transmitted by the client to the server). As such, the proposed collaborative validation scheme ensures that the accesses performed by all the workers in the same warp target the same element e_t residing on the off-chip memory.

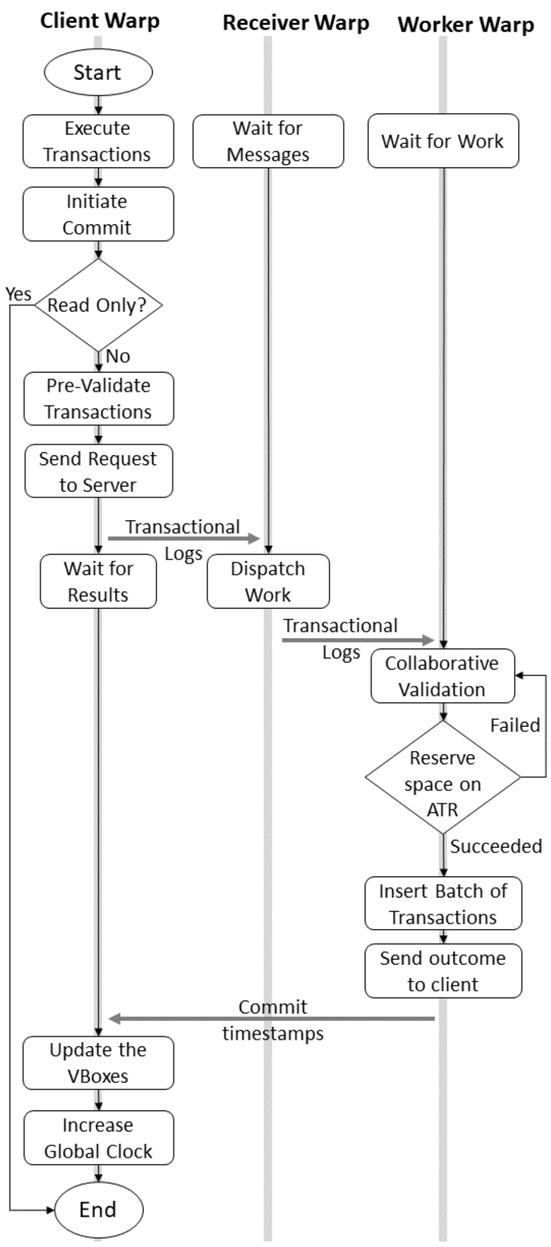


Figure 4.2: Flowchart of the execution of a CSMV's transaction.

Figure 4.2 presents the paths that a group of warp transactions can take during its execution. This includes all of the mechanisms described in this section and how they interact with one another to help commit the batch of transactions. For simplicity, the various types of aborts and their consequences are omitted from the flowchart, given that they differ significantly depending on the state of the other transactions in the warp.

Such intricacies of each of stages are described in detail in Section 4.2.2.

4.2 Detailed Description of the Algorithm

This section presents a detailed description of CSMV. I start by presenting two important data structures in CSMV, and, afterwards, each of the mechanisms used by CSMV during the processing of a transaction.

4.2.1 Data Structures

There are two relevant data structures in CSMV, namely, the Versioned Boxes (VBoxes) and the Global Time Stamp (GTS). A Vbox represents a memory location and holds its history. As mentioned before, this data structure is implemented as a circular buffer with a statically determined number of entries where each entry holds a pair $\langle$value,version$\rangle$. For each Vbox there are also two pointers: one for the most recent (*head pointer*); and, a second for the oldest entry (*tail pointer*). When a transaction reads from a Vbox it starts from the head pointer and ends in the tail pointer, scanning in a backward fashion, while looking for a suitable version that is consistent with the GTS observed at the start of the transaction. When the transaction wishes to add a new value/version pair to the Vbox, it first advances the tail pointer (assuming a steady state in which the set of versions of the VBox is full), writes the updated values and finally advances the head pointer.

The Active Transaction Record (ATR) stores the history of the transactions that have already committed or are in the final stages of their commit logic (i.e., just after the moment they were added to the ATR list). This data structure, which is implemented as a circular buffer, is composed by entries that contain the following: $i)$ the commit timestamp; and, $ii)$ the list of addresses for each write. The ATR list includes three pointers that enable its access: $i)$ a read pointer; $ii)$ a write pointer; and, $iii)$ a tail pointer. The tail pointer indicates the oldest valid transaction entry in the ATR list, while the read and write pointer usually sit at the head of the list, pointing to the most recently committed transaction in the record. In scenarios where no transactions are being inserted into the record, both the read and the write pointers point to the same entry. When a new batch of transactions has to be added to the ATR, then CSMV tries to advance the write pointer via a CAS operation, and after completing the insertion, the read pointer to once again match with the write pointer. Transactions that have to be validated start at the read pointer and scan the ATR list backwards, as it represents the most stable data available in the active record list.

An example of 2 transactions, $T1'$ and $T2'$, being added to the ATR can be found in Figure 4.3. In Step 1), the record is in a stable state, i.e., the read and write pointers point to the same entry which means that no other transaction is being added. The new transactions $T1'$ and $T2'$ have already passed validation and have to be inserted in the ATR. In order to proceed with the insert the write pointer has to be manipulated via a CAS operation (to solve the race condition).

In this case, the CAS for $T1'$ succeed and fails for $T2'$, as can be seen in Step 2). Still in this step, the write set of $T1'$ is copied to the position where the write pointer currently is. The fact that the read pointer and the write pointer are not pointing to the same entry tell the other transactions to wait. Which is the case of $T2'$, since $T2'$ has not won the CAS, it waits for the $T1'$ write set to finish copying into the ATR. $T2'$ can try again when the write pointer and the read pointer match once again.

In Step 3), $T1'$ has finished copying its write set into the ATR, it is given a commit timestamp, i.e., $ts = 4$, and we rename $T1$' with $T4$ in the drawing to represent the order by which the transaction joined the ATR list. Afterwards, the read pointer advances to the location of the write pointer, indicating to other

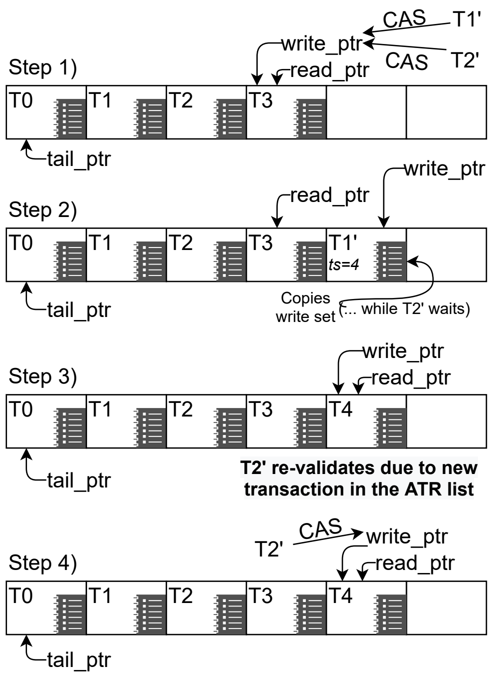


Figure 4.3: Use case of two concurrent transactions adding themselves to the ATR list.

transactions that this entry is now safe to read. $T2'$ now needs to be validated again, but this time only against the new entry, i.e., $T4$. In Step 4), $T2'$ has passed the validation against $T4$ and is therefore free to try to increase the write pointer using the CAS once more.

Besides the ATR and the VBoxes, there is also a global clock (GTS), which indicates how many transactions have been committed so far, and is used to attribute the timestamps of each transaction.

4.2.2 Algorithm

After presenting the VBoxes and the ATR list, we will now turn the focus to the execution flow of a transaction, including the various steps of the commit phase. The pseudocode for the most complex stages of the commit phase is also present in this section, as to assist in better understanding of the corresponding stage.

Transaction begin: The transaction is wrapped in between two primitives: TXBegin and TXCommit. TXBegin starts a transaction. It resets the read and write sets of the transaction and also takes a snapshot of the GTS. A boolean flag can also be passed to this function to specify that it is a ROT.

Read instrumentation: When reading, the transaction checks if the address is in the write set, returning the most recent value if the address is already written. Afterwards, the transaction looks in the respective Vbox for the most recent entry that is older than the start of the transaction. Having found the value, the transaction adds the address of the Vbox to its read set and returns the value that it found. ROTs do not need to track the read set, for which this operation simply reads from the Vbox and returns it immediately.

Write instrumentation: When writing, the transaction also checks first its write set already contains the address, in case the address is present, the value is simply updated. However, if this is the first time the address is written to, the transaction will have to create a new entry in its write set and store both the address and the value. Note that at this stage, the actual Vbox is not modified, instead the transaction just marks in its local write set its intention to modify the address along with the tentative value.

Transaction commit: After the transaction has finished all the write and read operations, it is time to call TXCommit. If the transaction has not performed any write operations, i.e. it is a Read-Only Transaction (ROT), the transaction can skip further verification and end the commit phase immediately. ROTs do not have to go through the server since: $i)$ they have already observed a valid snapshot of the shared resources; $ii)$ the ATR list is only needed to serialize update transactions. Hence, the transaction does not have to validate or be inserted into the ATR.

Pre-validation: In case a transaction modifies any address, it is an update transaction and requires validation. CSMV starts by pre-validating the transaction against others in the same warp. The pseudocode for this stage can be found in Algorithm 1.

Algorithm 1 Pre-validation logic.

```
1:  function TXPreValidation
2:      for all LaneID < myLaneID do              ▷ Check all lanes before myself (i.e. with higher priority)
3:          for all writeAddr ∈ WriteSet[LaneID] do
4:              for all readAddr ∈ ReadSet[myLaneID] do
5:                  if readAddr = writeAddr then            ▷ If I read a value updated by a coworker...
6:                      conflictArray.toggleBit(LaneID)     ▷ ...then I conflict with that thread in LaneID
7:
8:      if conflictArray.isZero() then
9:          myStatus ← "commit"                                   ▷ No Read-Write conflicts found
10:     else
11:         myStatus ← "tentative abort"                ▷ Else I may abort if the conflictees do not
12:
13:     for all LaneID ∈ Warp do              ▷ Each thread sends its status to another thread in rounds...
14:         otherStatus ← broadcast(LaneID, myStatus)     ▷ ... and observes their status on each round
15:         if conflictArray.getBit(LaneID) then              ▷ If that other thread conflicts with me...
16:             if otherStatus = "commit" then
17:                 myStatus ← "abort"                       ▷ ... and commits, then I have to abort
18:             else if otherStatus = "abort" then
19:                 conflictArray.toggleBit(LaneID)     ▷ ... and definitively aborts, then I can commit
20:                 if conflictArray.isZero() then
21:                     myStatus ← "commit"
22:     return myStatus
23:
```

Threads in the warp can pre-validate via comparing their transaction read set against the write set of the other transactions that executed within the same warp. CSMV uses a 32-bit array to track conflicts, which is represented by the variable $conflictArray$ in Algorithm 1. Conflicts among threads are marked by means of setting the corresponding position in $conflictArray$ to 1 (Line 6).

In case of conflict the policy that CSMV follows is to give priority to smaller ID threads, i.e., a transaction will only conflict with another whose thread ID is smaller than its own (Line 2). As an example, assume threads $T0$, $T1$ and $T2$ are from the same warp, $T1$ conflicts with $T0$ and $T2$ conflicts with $T1$ (i.e., $T1$ wrote an item that $T2$ read). The $conflictArray$ for each of them would be: $T0$=[000], $T1$=[001],

$T2$=[010]. While it would be easy to simply abort all transactions whose conflict array is different from 0 it could lead to more spurious aborts. In the example, $T2$ would be aborted even though $T1$ (which conflicts with $T2$) would also abort. Aborting $T2$ is, thus, unnecessary.

To avoid these spurious aborts, CSMV implements a mechanism, where a transaction can have one of three possible states: *commit*, *abort* or *tentative abort*; and each in turn broadcasts its state to the others using warp communication primitives The higher priority transaction (i.e., the thread with $laneID$ equal to 0) starts the rounds of broadcasts (Line 14) by means of sending its status (which, in the case of thread 0 is always "commit") via a `__shuffle` operation, while the other threads observe the status of thread 0. The other threads follow the same scheme and eventually send their status according to their $LaneID$ (i.e., from higher to lower priority). When a transaction receives the state of a higher priority sender transaction, the receiver checks first if it conflicts with the sender (Line 15). If the broadcasted state was "commit", then all conflicting transactions with lower priority (i.e., higher ID) would be forced to abort (Line 17). However, if the broadcasted state was "abort", the conflicting transactions eliminate that conflict from their conflict array (Line 19). If by doing so, they eliminated all active conflicts, they can change their state to "commit" (Line 21). It is worth noting that, because of the imposed order, the "tentative abort" status of a given thread will change to either "abort" or "commit" when it gets its turn, thus, broadcasting the definitive status.

When the pre-validations starts, any transaction whose conflict array is 0 can be considered "commit" (Line 9), while all others are "tentative abort" (Line 11). In the example, T0 broadcasts "commit", causing T1 to change its state to "abort". T1 then broadcasts this state, which means that T2 can eliminate the conflict it had with T1 in its conflict array, which in turn makes this array equal to zero, so T2 can change its state to "commit".

With the pre-validation step concluded, the warp sends its batch of transactions to the server.

Message-passing system: The communication with the server has the granularity of a warp, as to ensure that once the messages get in the server they are all handled by the same warp, without mixing in any transactions from other warps. Therefore, commit requests are sent in groups with the size of a warp (32 lanes in our settings). If some of the transactions aborted during pre-validation, then the client thread responsible for that transaction sends a dummy message that will be ignored by the worker thread on the server side. To send a request to the server, a warp leader reserves a contiguous space the size of the warp on the message buffer, sharing the address with all the other threads in the warp. Each thread writes a message in the corresponding position in the buffer. The message contains the timestamp of the transaction and the ID of the thread that sent the request, the latter is needed to access the read and write sets of the transaction.

CSMV makes use of the message passing algorithm developed by Wang et al. [54] since it makes an efficient use of the synchronous execution model of the GPU, both for client to server, as well as for the intra-server communication. I changed the original algorithm in order to force the receiver warp to wait for a full batch of 32 requests. In the original algorithm such was not enforced, and, as I will show in Section 5.3, leveraging the GPU lock step warp execution improves throughput considerably. The receiver warp is responsible for dispatching the commit requests to the worker warps and only does so when there is enough work to complete the warp.

Collaborative validation: After obtaining the batch of transactions the server can start validating transactions. The pseudocode for this stage of the commit is shown in Algorithm 2.

Algorithm 2 Collaborative Validation logic.

```
1:  function TXVALIDATE
2:      if myTS < ATR[tail_ptr].TS then              ▷ Reads that can no longer be validated due to...
3:          myTS ← 0                                 ▷ ...newer transactions overwriting write-sets of older ones...
4:          abort[myLane] ← true                     ▷ ...cause the transaction to abort spuriously
5:
6:      for all lane ∈ Warp do                       ▷ Collaboratively validate each transaction in the warp
7:          otherTS ← broadcast(lane, myTS)          ▷ Entire warp decides the next target transaction
8:          if otherTS = 0 then                      ▷ If the target transaction has already aborted...
9:              continue                             ▷ ...then just skip it
10:
11:         anchor ← otherTS mod ATRSize             ▷ Oldest transaction that may conflict with this one
12:         nbIters ← (read_ptr - myLane - anchor)/warpSize   ▷ First item in the partition of work
13:
14:         targetEntry ← read_ptr - myLane
15:         while nbIters > 0 do                     ▷ Processes the items in the partition
16:             for all writeAddr ∈ ATR[targetEntry].writeSet do
17:                 for all readAddr ∈ ReadSet[lane] do
18:                     if readAddr = writeAddr then ▷ If conflict is found...
19:                         abort[lane] ← true       ▷ ...notify that this transaction aborted
20:             if ATR[anchor].TS ≠ otherTS then     ▷ Oldest transaction was overwritten, thus...
21:                 abort[lane] ← true               ▷ ...it is no longer possible to check for conflicts
22:             targetEntry ← targetEntry - warpSize ▷ Move to next item (strided pattern)
23:             nbIters ← nbIters − 1
24:     return !abort[myLane]
```

Firstly, the transaction's timestamp is compared with the timestamp of the oldest valid transaction in the ATR list, i.e., the timestamp of the entry pointed by the tail pointer (Line 2). CSMV can check immediately if the ATR list has all the transactions that have been committed since the start of the validating transaction. If the timestamp is indeed smaller than the oldest timestamp stored in the record, then the transaction can never be sure if there were any previous transactions that could have eventually conflicted with it but were since then overwritten. To ensure consistency, the transaction has to abort spuriously (Line 4).

Do note that even if a transaction is aborted, the thread does not exit validation right away. Instead the thread will stick around to help other threads from the warp validate their transactions. When it comes the time for this aborted transaction to begin validation, the warp simply skips it (Line 9).

After this initial check, the actual collaborative validation process can begin. All the threads in the worker warp validate the same transaction against different positions of the ATR, i.e. all the threads are accessing the same read set, which is stored in global memory (Line 17), and comparing it to different write sets of the ATR (Line 16). If there is an intersection between these two sets, then a conflict occurred and the transaction has to abort (Line 19).

The worker warp starts validating backwards from the most recently added transaction (i.e., from the read pointer), with each thread targeting a different entry in the ATR (Line 14) and with a stride the size of the warp (Line 22), until it can find a transaction whose committing timestamp is equal to the starting timestamp of the validating thread, called the anchor entry.

This entry is the most recently committed transaction when the transaction being validated began, which means that the worker warp does not have to validate further backward (i.e., beyond the anchor).

Determining the anchor entry a priori is of great importance to the warp, since it allows each thread to calculate exactly how many iterations of of the validation loop they must do (Line 12). This matters because going over the anchor could potentially have a validating thread access entries that are also beyond the tail pointer of the ATR, meaning the thread could potentially compare against transitive data, which should never occur for the sake of correctness. On top of that, the anchor point must be checked after every iteration to ensure it has not been overwritten by another transaction (Line 20), since losing this point would mean that the thread does not have the necessary information to determine whether it did or did not conflict with another transaction. Hence, it spuriously aborts (Line 21).

Threads continue validating that transaction until they reach the anchor point for that transaction or until a thread finds a conflict, at which point they move on to the next transaction in the warp, until all of them are validated.

Batched ATR Insert: Transactions that have successfully passed validation can begin the process of insertion on the record. The pseudocode for this stage is present in Algorithm 3 as to provide a better understanding of this process.

Algorithm 3 Batched Insert logic.

```
 1: function TXADDTORECORD
 2:     oldPtr ← currPtr ← readPtr                      ▷ Copies the read pointer of the ATR ...
 3:     liveTX ← TXValidate(currPtr, tailPtr)            ▷ ... and validates from that copy backwards
 4:     liveBallot ← ballot(liveTX)                      ▷ Gets to know the state of the others in the warp
 5:     warpLeader ← electLeader(liveBallot)             ▷ Elect a leader thread from that information
 6:     do
 7:         if currPtr ≠ readPtr then                    ▷ If more entries were added since it last validated...
 8:             currPtr ← readPtr                        ▷ ..then it validates again...
 9:             liveTX ← TXValidate(currPtr, oldPtr)     ▷ ...from the new pointer up to the old pointer
10:             oldPtr ← currPtr
11:             liveBallot ← ballot(liveTX)
12:             warpLeader ← electLeader(liveBallot)
13:         else if readPtr ≠ writePtr then              ▷ If another batch of transactions is currently writing...
14:             continue                                 ▷ ...wait for them to finish
15:         else if warpLeader then                      ▷ The elected leader of the warp of threads...
16:             if CAS(writePtr, countBits(liveBallot)) then   ▷ tries to advance the write pointer
17:                 insertPos ← writePtr
18:                 tailPtr ← writePtr+1                 ▷ Leader advances the tail pointer before anything else
19:         broadcast(warpLeader, insertPos)             ▷ Leader shares with other the writing position
20:     while insertPos < 0                              ▷ Threads continue in this loop until leader wins the CAS
21:
22:     if liveTX then                                   ▷ All the transactions that passed validation...
23:         insertPos ← insertPos + getOffset(liveBallot)    ▷ ...compute their own write position
24:         commitTS ← ATR.writeTX(insertPos, writeSet)      ▷ and insert their write set in the ATR
25:         if warpLeader then
26:             readPtr ← writePtr                       ▷ The leader advances the read pointer
27:     return commitTS
```

To be able to insert the entire batch of the transaction at the same time, the threads first need to know the status of their fellow transactions in the warp. To do so, they perform a $ballot$ operation over the result of the validation, that returns an array with the transactions that successfully passed validation (Line 4). From this array, the warp elects a leader (Line 5) that will perform the CAS operation on behalf of the entire warp. This leader is usually the thread in the warp with the lowest ID that managed to pass validation.

Since only a batch of transactions can be added to the record at the same time, transactions have to make sure no other batch is already doing so. This can be quickly verified by comparing the write and read pointers of the record list, since mismatched pointers mean a batch is currently writing, causing others to wait (Line 13).

Additionally, transactions must also ensure that they are validated against all the transactions in the ATR. They can check this by comparing the copy of the read pointer they made during their first validation to the current value of the that pointer (Line 8). Should any new transactions have been added to the record since this transaction last validated, then the validation must be repeated (Line 10), this time however, only validating against the new transactions on the ATR.

Having checked all the above conditions, then the warp leader will then try to win a CAS operation on the write pointer on behalf of the entire warp (Line 16). This leader will try to advance this pointer a number of entries that is equal to the number of transactions in the warp that have passed validation, effectively reserving this space in record for warp's transactions. If the leader fails to win the CAS, then the warp will have to wait for the winner to finish adding themselves, and the validation will have to be repeated, albeit this time only against the winner batch (Line 10).

However if the leader does manage to win the CAS, first it will also advance the tail pointer an equal amount of entries so that validating transactions know not to go further than that point (Line 18), then it will notify the rest of the warp by sending them the position in the record list where they can write to (Line 19). The transactions write to the corresponding position both their write set and also a new commit timestamp, that is essentially the order that the transactions were added to the record list (Line 24). When all threads have finished writing, the warp leader will move the read pointer to be the same as the write pointer, thus allowing other batches of transactions to insert themselves to the ATR (Line 26).

Server to client communication: Once all the transactions in the warp have been either successfully added to the record or failed the validation and aborted, it is time to communicate these results to the client so that it can proceed with the final stage of the commit. Unfortunately, it was not within the scope of Wang et al. work [54] to allow the server to reply back to the clients, so I had to extend their message passing system to encompass this feature.

Since the communication of server to client is much simpler than the inverse situation, because it is a singular server warp that communicates with another singular client warp, a simple lightweight approach was used that allows for greatly reducing the complexity when compared to the client to server channel. In this solution, each client warp has a small pre-allocated bucket to where the server warps write the commit timestamp of each transaction or a negative value in case the transaction aborted.

Write-back: When the client gets the results back from the server, and in case the transaction was indeed successful, the write back process can begin, where the transactions can apply to the VBoxes the contents of their respective write sets. Before they do that however, the tail-pointer of each accessed Vbox is advanced, so that other transactions that could be reading that Vbox do not access beyond that point. After this the transaction is free to update all VBoxes it wrote to with the pairs of value/version, where version is the commit timestamp that the transaction got in the server. Finally, the head pointers of the VBoxes are updated, finalizing then the write back phase.

However, the updates made by the transaction are not yet visible to new transactions. Since the global clock has not been updated, new transactions do not see the VBoxes values because their version is higher than the timestamp of new transactions. To make them visible, the transaction has to finalize the commit phase by incrementing the global clock to be the same as its commit timestamp. Yet the order on which the transactions were added to the record list must be respected, because by increasing the global clock to the commit timestamp right away, the transaction may inadvertently publicize the updates of a transaction that is still in the process of updating the VBoxes which must be avoided for the sake of correctness.

Increasing the global clock is therefore a sequential portion of the code, and one that must respect a strict order. Still, the same method used by the batched insertion on the record can also be applied here, since there is an assurance that in the same warp the commit timestamps are contiguous, so a warp leader increases the global clock by the number of transactions that have been committed by the whole warp, instead of having each transaction individually and sequentially increasing the clock, effectively speeding up this final portion of the commit.

Chapter 5

Experimental Evaluation

This section presents the results of an experimental study that aims at answering three main questions:

- What is the impact of the level of concurrency on the performance of CSMV for a given percentage of read-only transactions? (§ 5.2)
- To what extent do the various mechanisms leveraged by CSMV contribute to enhance its efficiency with different workload characteristics? (§ 5.3)
- How competitive is CSMV when compared to state of the art STMs for GPUs and CPUs and in which type of workloads can it achieve the largest speed-ups? (§ 5.4)

5.1 Experimental setup

CSMV is evaluated by using two benchmarks: the Bank benchmark [59] and MemcachedGPU [34].

The Bank benchmarks, as the name suggests, simulates the activities of a bank that maintains a number of accounts with a given initial balance. This benchmarks generates two type of transactions: (i) update transactions that transfer a random amount of money between two bank accounts; (ii) ROTs, that read all the accounts and compute the total balance of the simulated bank.

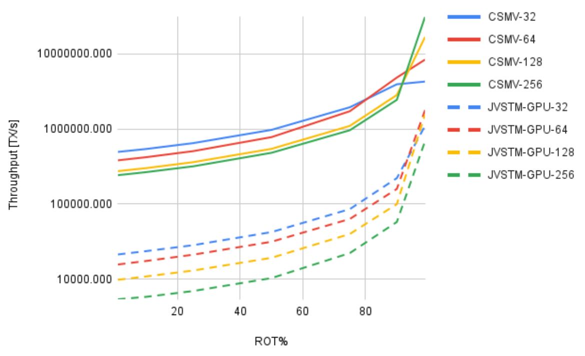
MemcachedGPU [34, 60] builds on Memcached, a popular in memory object caching system, that it accelerates via the use of GPU. MemcachedGPU was initially accelerated using a lock-based synchronization [34] and later on [60] adapted to make use of a STM. The mutable shared state of this benchmark is an n-way set associative cache with an LRU replacement policy. The keys/values used are of 16B/32B, respectively. MemcachedGPU provides two methods, namely GET/PUT, that are used to retrieve/store a value in the cache and that are wrapped, respectively, within read-only/update transactions. Given a $\langle key, value\rangle$ pair, the key is hashed to a set, which is then scanned to find a matching key and retrieve/set the corresponding value (for GETs/PUTs, respectively), updating the LRU metadata of the corresponding slot. Since transactions scan the ways of a set until they find a matching key, this benchmark generates transactions that read a variable number of items, upper bounded by the cache associativity value. Update transactions, in addition to scanning the set for a matching key also issue 4 write operations to update the key's metadata. In order to mimic a realistic workload, analogously to what done in the original evaluation of MemcachedGPU, the key space is accessed via a Zipfian distribution and generate 99.8% of GET operations (i.e., ROTs in this STM-based implementation), as suggested by the work of Atikoglu et al. [61] .

Both CSMV and the other STMs for GPUs are evaluated using a machine equipped with an Nvidia GeForce GTX 1080 Ti (Pascal micro architecture, 28 SMs, 11 GB of RAM, CUDA v10.0), an i7-2600k CPU and 8GB of RAM. When evaluating STMs for CPUs I use a machine equipped with an Intel Xeon CPU E5-2648L v4 (14 cores/28 hardware thread, 1.80GHz), 32 GB of RAM, running openjdk 1.8.0_292 and Ubuntu 18.04LTS.

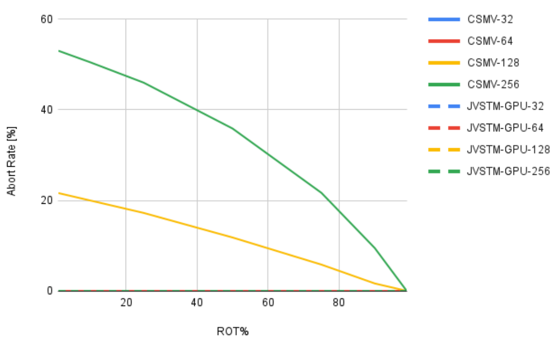
To ensure a fair comparison, all the STMs for GPUs are configured to use the same number of thread blocks, namely 28 (i.e., as many as the number of available SMs in our GPU). Client-server based solutions reserve one of the SMs to act as the server node. This server SM is configured to use 1024 threads per block, where 32 of these (a warp) play the receiver role, while the remaining transactions act as workers. The size of the ATR also varies according to the solution used, due to how much available space each of these has, which depends directly how much of the ATR is stored in the scratchpad memory. CSMV-sh, that keeps everything in *Shared Memory*, has room for 2000 entries; CSMV-ts, which stores only the CTS in *Shared Memory* keeps an ATR with 8000 entries, and both CSMV-gb and JVSTM-GPU, that store the ATR completely in off-chip memory, maintain an ATR with 65k entries. Regarding the VBoxes, these store the 10 most recent versions of each data item, which is enough to not have observed any spurious aborts related to lack of room on these structures for the studies conducted below.

When testing STMs for CPUs, as many threads as available hardware threads are used. All the reported results were obtained as the average of at least 3 runs.

5.2 Threads per Block selection



(a) Throughput



(b) Abort Rate

Figure 5.1: JVSTM-GPU compared against CSMV for a varying number of ROTs (x-axis) and for 32, 64, 128 and 256 threads per block (shown in the curve labels).

This initial study shows how the level of concurrency and the ratio of ROTs can affect the performance of two of the solutions developed in this dissertation, namely JVSTM-GPU, the direct porting of an existing GPU MV STM, and CSMV, the primary solution developed in the scope of this thesis. In this section, the version of CSMV used is **sh**, which is the version that stores the ATR in its entirety in the scratchpad memory. To access the impact of the level of concurrency different values of thread per thread block are used in this study: 32, 64, 128 and 256 threads per block.

This study is conducted using the Bank benchmark, which for this study is configured to be a contentionless, i.e. each update transaction accesses accounts from a set of bank accounts that is exclusively accessed by that transaction, thus generating no conflicts. ROTs are also configured differently from the standard Bank benchmark to access 200 bank accounts rather than the entire Bank, whereas update transactions, transfer money from one account to another.

Figure 5.1 reports the results of this experiment. Each line represent a pair of solution and number of threads per block. JVSTM-GPU is represented by dashed lines and CSMV is represented by solid lines, while each color corresponds to a different number of threads per block. The ratio of ROTs varies along the x-axis, while the y-axis for the left plot reports the throughput and the one for the right plot reports the abort rate. This experiment shows that all the different versions of CSMV used consistently outperform JVSTM-GPU, with gains averaging at 23$\times$ when comparing the best variant of each solution. The larger speed-ups occur for low percentages of ROTs, since the changes applied on CSMV from JVSTM where focused on boosting the commit phase of update transactions.

Both solutions show better performance for lower concurrency levels when the ratio of ROTs is low, as the variant that uses the fewer threads per block at 32, outperforms the others up to 75% ROTs. But as this ratio increases, increasing the level of concurrency to 64 or higher threads per block achieves better results, both for JVSTM-GPU and CSMV. For the highest possible level of ROTs at 99%, all the solutions but especially the ones that have a higher level of concurrency get a significant speed-ups of up 15$\times$ from the previous point in the plot at 90%.

Additionally, the abort rate plot in Figure 5.1 shows that even though there are no conflicts between transactions, since the each transaction only performs accesses within its own slice, there are still spurious aborts which are caused by lack of space to accommodate all the transactions. Recall that for this particular variant of CSMV, **sh**, there are only 2000 entries available in the ATR. As the number of active threads exceeds that threshold, which occurs for 128 and 256 threads per block, some will be forced to spuriously abort because when it reaches the time for them to validate This happens since the anchor point for these transactions has already been overwritten, making it impossible for these transactions to ensure a no conflicts have occurred.

A better performance of lower concurrency variants for lower percentages of ROTs is expected since, even though they do not generate conflicts directly, more concurrent active transactions affect the system in two ways: $i)$ when validating, the transactions will have to compare themselves against a greater number of transactions and eventually the toll of validating more transactions becomes more expensive than reducing the level of parallelism, $ii)$ the overhead incurred by the sequential sections of the algorithm as well as the lock acquisition is exacerbated. These two points are true for both solutions, but

CSMV suffers from yet another problem related to the increase in concurrency, related to the server. Since the number of worker threads in the server is limited to 992 threads, the server cannot process more than this number of requests at a time. Increasing the number of clients active at the same (i.e. the level of concurrency) to the point where there are more clients than worker threads, and having all those clients execute exclusively update transactions, which require handling from part of the server, may cause transactions to wait for available worker threads before these can start validating.

Nonetheless, CSMV is a MV STM and as it commonly the case with MV TMs, it is expected to be employed in read-dominated workloads [29, 40]., as it allows ROTs to be committed much faster in detriment of update transaction's execution. Using this system to test the workloads with only update transactions is, therefore, an unfair comparison.

Considering this, and since most other workloads present in this chapter feature a high ratio of ROTs, the experimental studies presented next in this chapter will use 64 threads per block as the selected kernel configuration. This configuration, and as seen in Figure 5.1, is among the best performers on workloads with a ROT% of 90% to 99% for both solutions, which represent the interval where most of the next studies will take place, while still maintaining a competitive performance for lower ROT ratio.

5.3 Optimization Comparison

As already mentioned, CSMV makes uses of a number of mechanisms designed to take advantage of the architectural characteristics of GPUs and enhance efficiency. This section presents several studies that aim at evaluate their impact and interplay.

To this end, two extra variants of CSMV are considered, which are obtained by disabling, in an incremental fashion, different mechanisms at the core of CSMV's design.

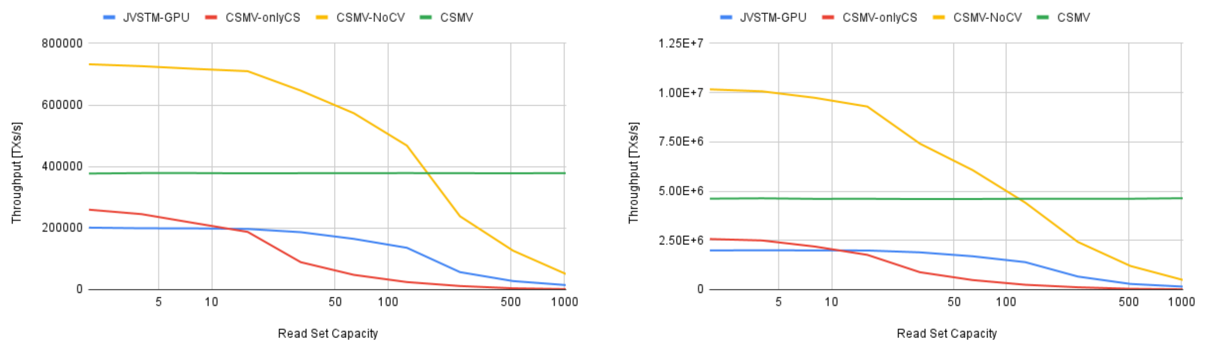
The first variant is obtained by disabling the collaborative validation mechanism. Let this variant be henceforth referred to as CSMV-NoCV. Then a second variant is derived, called CSMV-onlyCS, by disabling from CSMV-NoCV the Batched ATR Insert, the client-side pre-validation and the client-side write-back. As such, CSMV-onlyCS can be also regarded as a variant of CSMV that preserves only the client-server design, while disabling any other complementary mechanism proposed in this work. Both CSMV and JVSTM-GPU are also added to the plots as to make a better comparison between the developed solutions.

In this section, all the experiments are conducted using the Bank benchmark with a sharded workload, i.e. no conflicts happen between update transactions. This is done to consider a scalable workload and focus on the scalability of the solutions. The experiments look to evaluate the variants in regards to a number of parameters affecting both at the the STM level and the level of application workload, namely the maximum read and write set capacities and the actual size of the transaction in regards to how many reads and writes it does. The experimental studies reported below depart from the same common ground: a read set max size of 1024, 2 writes and 2 reads per transaction; and in each study one of these parameters is treated as the independent variable (which is varied on the x-axis) and lock the values of the other two. Three of the experiments reported below were conducted on two workloads with a different percentage of ROT, 0% and 90%. An additional experiment is also conducted in Section 5.3.2, to expand on the conclusions drawn from the study that treats the number of writes per transaction as the independent variable. This study is conducted on a workload that features a ratio of ROTs of 99%.

Besides the relevant plots for each study, there is also a table that showcases the time that on average a transaction takes to complete the given stage of the commit phase. For both simplicity and readability, only three of the stages that account for the largest portion of the commit phase are shown: $i)$ the wait server, which is the time that the transaction has to wait before a worker from the server starts handling their commit request; $ii)$ the validation phase, which represents all the time that the transaction spent validating, including the pre-validation and the repeated validations that occur when a thread fails to win the CAS and $iii)$ the ATR insert phase, which is comprises the time spent by a thread to perform the CAS, update the ATR and perform the write-back phase.

5.3.1 Varying the Read Set Capacity

The first experiment looks to find how varying the read set capacity for update transactions affect the performance of all the variants in question. Note that this is a change at the level of the configuration of the system rather than on the workload itself, i.e. it is not varying how much each transaction is in fact reading but instead how many items each could possibly read. Update transactions read and write to a pair of bank accounts, while ROTs read the value of 100 accounts. Do note that in the case of ROTs, due to the fact that these are guaranteed to commit and thus do not need to validate, they can skip the instrumentation overhead of storing their reads on the read set, allowing them to read more than what the read set size would normally permit for an update transaction.



(a) Throughput for the workload without any ROTs

(b) Throughput for the workload with 90% of ROTs

Figure 5.2: Effects of varying the read set maximum size for workloads with 0% and 90% ROTs.

Figure 5.2 reports the absolute throughput values for all the variants used on both workloads. The abort rates are omitted here because neither variant ever aborts for the setting of these tests, namely the workload is sharded and the number of active concurrent threads is inferior to the size of the ATR, unlike the experiments on Section 5.2.

The performance of CSMV remains the constant as the read set size is increased. Other solutions however, start with a good performance but quickly deteriorate as this parameter rises. CSMV-noCV starts with an absolute throughput that is up to 2$\times$ as large as the solution with all optimizations, CSMV, but for read set sizes larger than 128 entries the situation inverses, and CSMV becomes the top performer with speed-ups of up to 9$\times$ when compared to CSMV-noCV and 30$\times$ when compared to JVSTM-GPU. But what exactly is causing these results?

Note that increasing the max read set size has the effect of causing a change on locality of the accesses made by transactions when they are validating their read set against the write set of transactions stored in the ATR. When validating, the transactions have their read sets in global memory, which is slow but has the advantage of being able to coalesce the accesses made to it by the same warp, depending on how close the data is to each other. Since the read sets of the transactions in the same warp are stored contiguously in the global memory, having small read set sizes ensures that accesses by the same warp can leverage the coalescing mechanism made available by the GPU architecture, but as they increase, the reads sets are further and further apart in memory, coalescing these accesses efficiently becomes impossible and so the performance of the system drops.

As seen in Table 5.1, CSMV does not suffer from this problem since the validation time remains constant at 2.5ms from a read set capacity of 2 up to 1024 entries. Recall from Section 4.1 that the main goal of the collaborative validation mechanism is to improve the locality of the accesses made by the warp, and it does so by having every thread accessing the same element e_t of the same validating transaction, to then validate against a different entry of the ATR. As such, this scheme is trading the locality when reading from the ATR, which is stored in the scratchpad memory that benefits from already sufficiently low latency, to maximize locality when targeting the higher latency global memory.

The collaborative validation mechanism does incur some overhead with the communication necessary to synchronize the validating warp. These costs outweigh the benefits they bring in terms of improved memory locality, when the read-set maximum sizes are small, since, as already mentioned, access locality is already good in this case, which causes CSMV-noCV to outperform CSMV. This can be observed in Table 5.1, by checking the times that a transaction takes on average to validate for lower capacity sizes. If the programmer were to know *a priori* that the workload only generates transactions with small read sets and the read set capacity can be set accordingly, CSMV-noCV is even faster than CSMV with speed-ups that can go up to 2$\times$.

	Commit			**Wait Server**		**Validation**			**ATR Insert**		
Read Capacity	JVSTM GPU	CSMV	CSMV noCV	CSMV	CSMV noCV	JVSTM GPU	CSMV	CSMV noCV	JVSTM GPU	CSMV	CSMV noCV
2	8.73	2.26	4.37	1.81	0.91	2.98	2.55	1.34	2.78	0.01	0.01
4	8.73	2.26	4.36	1.80	0.91	2.98	2.54	1.34	2.79	0.01	0.01
8	8.74	2.30	4.37	1.81	0.92	2.91	2.55	1.36	2.70	0.01	0.01
16	8.77	2.32	4.37	1.81	0.93	3.02	2.55	1.37	2.80	0.01	0.01
32	9.17	2.62	4.37	1.81	1.06	3.29	2.55	1.55	2.72	0.01	0.01
64	10.28	2.94	4.37	1.81	1.20	4.62	2.55	1.73	3.22	0.01	0.01
128	12.25	3.58	4.37	1.81	1.47	6.55	2.55	2.10	2.46	0.01	0.01
256	27.32	6.84	4.36	1.81	2.86	13.85	2.55	3.96	1.14	0.01	0.02
512	54.12	13.28	4.36	1.80	5.60	31.48	2.55	7.66	1.15	0.01	0.02
1024	100.83	32.74	4.37	1.81	13.87	68.84	2.55	18.85	1.34	0.01	0.02

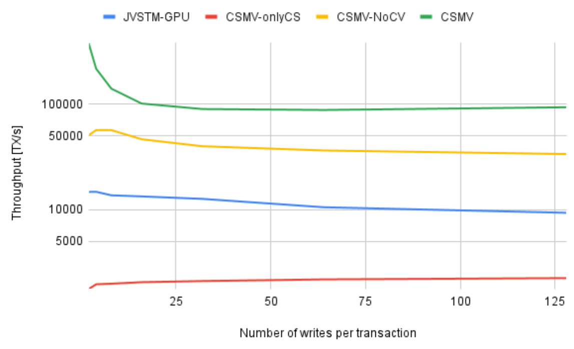
Table 5.1: Breakdown of the average time taken by some of the commit stages when varying the Read Set Capacity for the 0% ROT workload (in milliseconds)

Given these insights, for the remainder of this chapter, I shall continue using the fully optimized

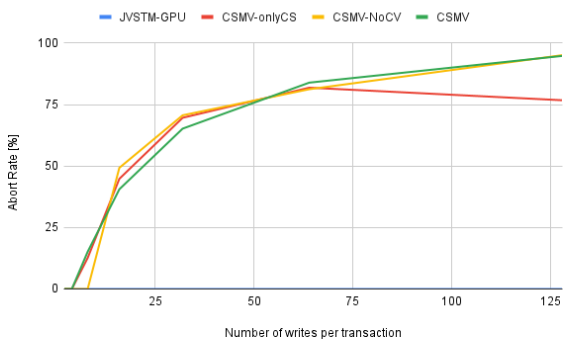
CSMV as the main solution developed, since it offers a more robust approach that can maintain constant performance regardless of the read intensitivity of the workload.

5.3.2 Varying the Number of Writes per Transaction

The second study of this section looks to quantify how the differently optimized variants handle a change in the workload with respect to the number of write operation that a transaction does. As before, this experiment locks two of the parameters, in this case, the read set capacity and the number of reads a transaction does. Update transactions will then increasingly execute longer transactions. This study also changes the write set size in tandem with the number of writes each transaction will perform for simplicity in reporting the implications that these parameters have on the performance. As with the previous experiment, two workloads are used, with ROT ratios of 0% 90%.



(a) Throughput



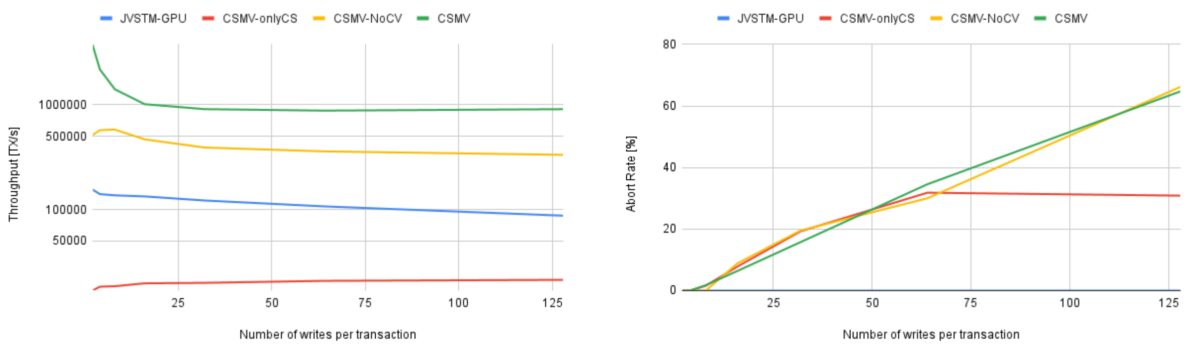
(b) Abort Rate

Figure 5.3: Varying the number of writes each transaction performs on a workload with only update transactions.

Figure 5.3 reports on the throughput and on the abort rate of the developed variants on the y-axis while varying on the x-axis the number of writes per transaction. As expected from executing longer transactions, the throughput for CSMV gets lower as you increase the number of writes, dropping from 380k TX/s to 90k TX/s, since not only it makes the execution of the transaction longer, but it also means that when validating, the transactions have to compare their read set against a greater number of writes per entry escalating the validation times.

Despite not showing a drop in performance at the same level as the CSMV, the other variants still suffer a hit in throughput albeit a small one. The reason why this drop is smaller than for CSMV is that, as seen for the previous study, the commit time, and more specifically the validation time for these transactions is completely dominated by the high latency on the global memory exacerbated by the fact that it cannot make efficient use of the coalescing mechanism. In comparison, increasing the number of accesses on a fast memory subsystem like the *Shared Memory* has little impact on the overall performance of these solutions.

Figure 5.4 shows how the variants handle a similar workload but now where 90% of the transactions are ROT. The trends for all the solutions remain the same as the workload with only update transactions, with the absolute throughput being higher as expected since ROTs are executed much faster than



(a) Throughput (b) Abort Rate

Figure 5.4: Varying the number of writes each transaction performs on a workload with 90% of ROTs.

updates, for all the variants in the study. For both workloads, CSMV remains the best performer, with speed-ups of up to 24× and 7× compared to JVSTM-GPU and CSMV-noCV, respectively, for lower writes per transaction values, and an average of 10× and 2.5× when the transactions are more write intensive. The variant that only employs the client-server architecture, CSMV-onlyCS, is constantly the worst performer out of all variants, for both workloads, failing to surpass the direct porting of JVSTM to the GPU, JVSTM-GPU, which shows that the use of *Shared Memory* alone is not enough to overcome the overheads of a bi-directional communication introduced by this architecture.

Another interesting aspect can be seen from looking at the Abort Rate plot in Figure 5.3, is that this increases along with the number of writes per transaction. The memory variant used in the studies of this section is CSMV-sh, which recall from Chapter 4, is the variant that stores the totality of the ATR in *Shared Memory*, including the write sets of committed transactions. As these write sets increase, so does the size of each entry in the ATR and so fewer entries can fit inside the limited *Shared Memory* available.

As such, an ATR that can initially support 2000 entries, only has room for 50 entries at 128 writes per transaction, which makes the number of active concurrent transactions bigger than the entries on the ATR, hence the spurious aborts whose rate goes up to 95% for both CSMV and CSMV-noCV, that normally would not be there given that this is a sharded workload.

	Commit			**Wait Server**		**Validation**			**ATR Insert**		
#writes	JVSTM GPU	CSMV	CSMV noCV	CSMV	CSMV noCV	JVSTM GPU	CSMV	CSMV noCV	JVSTM GPU	CSMV	CSMV noCV
2	101.96	4.36	32.75	1.81	1.80	70.36	2.55	18.85	1.35	0.01	0.02
4	110.16	7.58	29.12	1.80	3.14	86.20	4.42	16.80	1.54	0.02	0.03
8	116.84	9.05	28.77	1.81	3.03	88.64	5.99	16.64	1.57	0.03	0.04
16	116.35	10.53	26.34	1.81	2.47	71.21	8.01	19.87	1.65	0.05	0.05
32	131.15	10.59	26.43	1.81	1.45	76.29	9.08	22.82	2.15	0.06	0.06
64	153.29	9.93	26.69	1.81	0.56	82.74	9.30	24.43	3.03	0.07	0.07
128	187.22	1.74	11.78	1.81	0.01	98.08	1.66	11.38	4.26	0.08	0.09

Table 5.2: Breakdown of the average time taken by some of the commit stages when varying the number of writes per transaction for the 0 %ROT workload(in milliseconds)

As for the abort plot in Figure 5.4, all the client-server based solutions show much fewer aborts, up to 65%, even though the size of the ATR remains the same, due to the high amount of ROTs, that do not need to be added to the ATR.

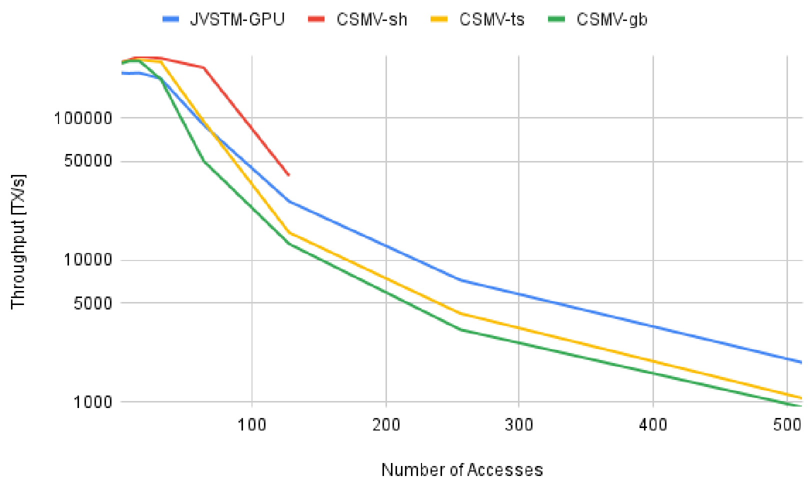


Figure 5.5: Comparing the different memory variants (sh,ts and gb) by varying the size of the transaction for a workload with 99% ROTs.

The abort rate for CSMV-onlyCS is lower due to the low throughput of this variant, since transactions are committed at a slower pace, new transactions have greater chance of finding their anchor point still in the ATR when they initiate validation.

CSMV-sh systems therefore have a limitation on how much a transaction can write, since to guarantee the correct functioning of the system, the ATR must have at least 32 entries. The other variants, CSMV-ts and CSMV-gb should be used instead for workloads that require a greater number of writes from their transactions.

On Table 5.2 it is possible to see the average times the stages of the commit phase take, which go in line with that was shown in the plots with respect to the throughput values. However for the 128 writes point, the validation times shrink from the previous point. This is explicable because since the ATR is reducing in size, which means the transactions do not have as many entries to validate against. The throughput however stays mostly the same, as seen in Figure 5.3, because these transactions instead of validating are aborting right away due to the lack of space in the ATR, not contributing to the overall throughput of the experiment.

Since CSMV-sh systems have a limitation on the number of writes a transaction can possibly execute in its life time, a new experiment is conducted to evaluate the different variants with respect to how the ATR is stored in memory. Recall from Section 4.1 that these are: **sh** that stores everything in *shared memory*, **ts** that stores the commit timestamp in the scratchpad memory and finally **gb** that stores everything in *global memory*. The experiment looks to evaluate how the different variants perform as the size of the update transactions increases. The ratio of ROTs is 99% where each of these is reading from 6000 different bank accounts.

The results of this experiment are presented in Figure 5.5, where the y-axis reports the throughput and the x-axis varies the size of each update transaction. The abort rates are omitted because no aborts occur for any of the variants, due to the high number of ROTs.

Despite being the best variant for under 128 accesses per transaction, CSMV-sh has a limit on how much each transaction can write, which makes it impossible to run to transactions that write to more than 128 items. The other memory variants of CSMV can be used instead for values superior to those but as seen from the plot, they fail to outperform JVSTM-GPU for all transaction sizes that are larger than 64 accesses. JVSTM-GPU exhibits an average gain of 70% over CSMV-ts and 110% over CSMV-gb for those transaction sizes.

Table 5.3 provides the breakdown of some parts of commit phase for JVSTM-GPU, CSMV-sh and CSMV-ts. From it, its possible to observe the difference in the validation times, namely how CSMV-ts starts incurring in longer validation phases from 128 accesses and up.

	Commit			**Wait Server**		**Validation**			**ATR Insert**		
TX Size	JVSTM GPU	CSMV sh	CSMV ts	CSMV sh	CSMV ts	JVSTM GPU	CSMV sh	CSMV ts	JVSTM GPU	CSMV sh	CSMV ts
2	0.82	0.002	0.002	0.0001	0.0001	0.005	0.001	0.001	0.037	0.0002	0.0002
4	0.81	0.002	0.002	0.0001	0.0001	0.005	0.001	0.001	0.031	0.0002	0.0002
8	0.83	0.004	0.004	0.0001	0.0001	0.012	0.003	0.003	0.040	0.0004	0.0004
16	0.89	0.007	0.008	0.0001	0.0001	0.038	0.006	0.007	0.044	0.0006	0.0006
32	1.47	0.050	0.07	0.0001	0.0001	0.360	0.048	0.072	0.052	0.0011	0.0011
64	10.58	0.120	6.54	0.0001	0.0001	7.77	0.118	6.54	0.051	0.0020	0.0020
128	43.19	4.101	68.83	0.6052	15.63	35.23	3.492	53.19	0.067	0.0039	0.0047
256	147.43		240.01		48.10	138.25		191.91	0.092		0.0084
512	563.63		953.13		180.51	547.47		772.60	0.120		0.0144

Table 5.3: Breakdown of the average time taken by some of the commit stages when varying the number of accesses done by a transaction for the 99 %ROT workload(in milliseconds)

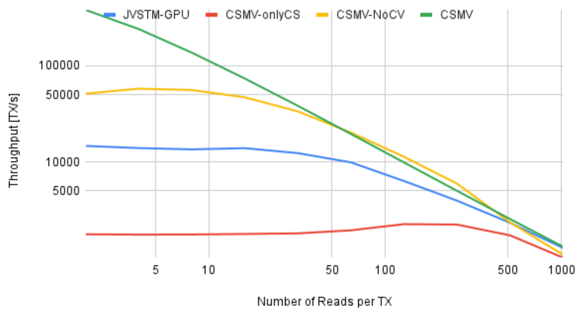
CSMV-ts stores the transactions timestamps in *shared memory*, while the rest of the ATR, namely the write sets go back to *global memory* in a similar manner to JVSTM-GPU. As the transactions grow in size, the ratio of shared to global memory accesses diminishes to the point that there is not a sufficient difference between the validation in JVSTM-GPU and CSMV-ts, at which point the former comes out ahead since it is not encumber by overheads introduced by the collaborative validation scheme and the limited number of worker threads in the server.

5.3.3 Varying the Number of Reads per Transaction

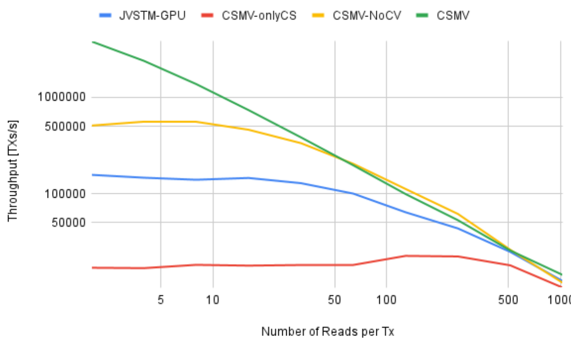
The fourth and final study of this section evaluates the impact on the performance when the update transactions perform a growing number of reads when executing. For this experiment, read set capacity and the number of writes parameters are locked to 1024 entries and 2 writes each and the number of reads is increased up to a full usage of the read set.

Figure 5.6 reports the results of this study, for both the workloads considered, 0% and 90% ROTs. The number of reads that each transaction performs is reports in the x-axis, while the y-axis shows the throughput. The abort rate plots are omitted because none of the presented solutions incur in any type of abort, which is expected since the ATR has enough room to accommodate all the concurrently running transactions at once.

As with the previous study on the number of writes, CSMV achieves the best performance for a low number of read operations per transaction, but now, and unlike what was observed before, the increased number of read operations have a greater toll on the validation phase than an increase on the number of writes such that CSMV has closer performance to CSMV-noCV, and actually being marginally surpassed



(a) Throughput for the workload without any ROTs



(b) Throughput for the workload with 90% of ROTs

Figure 5.6: Varying the number of reads each transaction performs on workloads with 0% and 90% ROTs.

by it on the range of 100 to 500 reads per transaction.

For the largest amount of read operations per transactions, at 1024 reads, CSMV is still the best performer among the variants presented but only by a small margin of speed-ups of 5% when compared to JVSTM-GPU and 20% when compared to CSMV-noCV.

#reads	Commit			Wait Server		Validation			ATR Insert		
	JVSTM GPU	CSMV	CSMV noCV	CSMV	CSMV noCV	JVSTM GPU	CSMV	CSMV noCV	JVSTM GPU	CSMV	CSMV noCV
2	101.2	4.4	32.8	1.8	13.9	69.5	2.5	18.9	1.34	0.01	0.02
4	106.9	6.9	29.2	2.9	12.3	82.2	4.0	16.8	1.39	0.01	0.02
8	111.5	12.1	29.7	5.1	12.6	90.7	7.0	17.1	1.35	0.01	0.02
16	110.0	22.5	35.8	9.5	15.1	59.3	13.0	20.6	0.91	0.01	0.02
32	127.5	43.2	50.6	18.2	21.3	70.4	25.0	29.2	0.89	0.01	0.01
64	162.8	84.3	82.3	35.5	34.7	93.6	48.8	47.6	0.87	0.01	0.01
128	257.4	165.0	146.2	69.2	61.4	179.6	95.8	84.9	0.96	0.01	0.01
256	412.6	321.6	269.6	133.6	112.2	319.5	188.0	157.3	0.97	0.01	0.01
512	704.7	619.6	681.5	252.6	278.4	564.2	367.0	403.1	0.93	0.01	0.01
1024	1267.7	1164.1	1400.9	458.2	545.3	1051.8	705.8	855.5	0.87	0.01	0.01

Table 5.4: Breakdown of the average time taken by some of the commit stages when varying the number of reads per transaction for the 0 %ROT workload(in milliseconds)

Table 5.4 showcases the breakdown of the commit phase for this study. As seen in the table, the increase of the number of read operations on update transactions makes the validation of these a massively more expensive operation for all the considered variants when compared to the other studies in this section, with validation times that are up to 5$\times$ longer than the writes study with a similar number of accesses. Since the validation scheme used compares the read logs of the validating transaction to the writes of the committed transactions, and since a validating transaction has its read set located in the global memory, the impact on raising the number of reads is superior to the that of raising the writes. Therefore, the lion's share of the transactions lifetime is occupied by the commit stage, roughly representing the 99% of the average transaction life time for all the variants. For client-server based approaches, transactions spend around 60% of this time validating and another 38% waiting for worker threads to handle their commit request. As for JVSTM-GPU, the transaction do not have to wait for the server, but still spends 80% of its lifetime validating and another 20% that is not directly attributable to any of the inner commit phases, but is instead due to thread divergence at the level of a warp.

The data gathered from these experimental studies allows for three main conclusions:

- The collaborative validation scheme (which is disabled in CSMV-NoCV), has the strongest performance impact, especially in workloads without any ROTs. That is explainable by considering that in workloads dominated by ROTs, it is expected that during validation it will be necessary to check for conflicts against a reduced number of transactions in the ATR (recall that ROTs bypass the commit phase and are not registered in the ATR).
- CSMV-noCV can be used instead of CSMV with improved performance if the workload's need are known a priori and they do not require a large read set capacity.
- The CSMV-sh variant is the best out of the three developed memory variants. It achieves better performance than CSMV-ts and CSMV-gb for any workload with the exception of write-heavy ones, where it simply does not run. Even for said workloads, CSMV-ts and CSMV-gb fail to outperform the direct porting of the JVSTM to the GPU, making them not worth to use for any situation.
- The employment of a "vanilla" client-server architecture, i.e., without the contribution of the additional mechanisms presented in Section 4.1 (which corresponds to CSMV-onlyCS) achieves worse performance than a straightforward porting of a design originally conceived for CPUs, namely JVSTM-GPU. Overall this confirms the relevance of the proposed optimization mechanisms, demonstrating the importance of employing them in a synergistic fashion.

5.4 Comparison with state of the art STMs

This section presents the results of an experimental study that aims at demonstrating how competitive is CSMV when compared to state of the art STMs for GPUs and CPUs and in which type of workloads can it achieve the largest speed-ups?

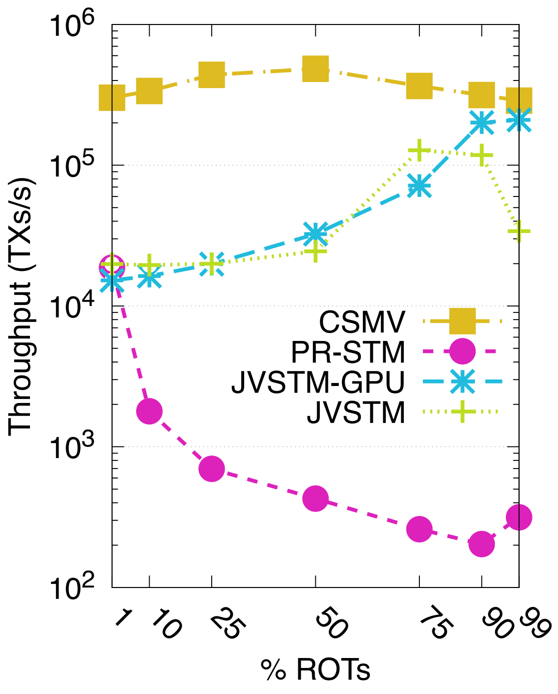
That is achieved by comparing the performance of CSMV with the following STM implementations:

1. PR-STM [16], a state of the art single versioned STM for GPUs that uses invisible reads, encounter time locking and a priority-based contention management;
2. JVSTM [40], a state of the art STM for CPUs that, analogously to CSMV, adopts a MV scheme;
3. JVSTM-GPU, a MV STM for GPUs, which recall from Section 3.2, was obtained by directly porting the JVSTM algorithm to CUDA and that can be also regarded as a variant of CSMV from which all of the GPU-oriented algorithmic optimizations described in Section 4.1 are removed.

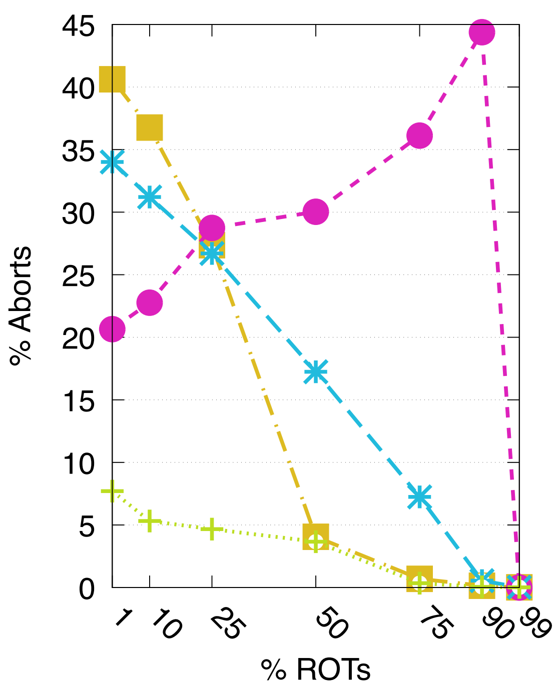
Bank benchmark. Figure 5.7 reports the performance of all the above mentioned baselines with the Bank benchmark, which is configured to maintain 6k accounts. In these settings, conflicts among update transactions have a relatively low probability to occur, compared to the likelihood of conflicts between update and read-only transactions.

On the x-axis varies the percentage of ROTs and the throughput is reported on the y-axis.

The throughput plot in Figure 5.7 shows that CSMV is consistently the best performing solution for all the considered values of the percentage of ROTs, with peak gains of around three orders of magnitude



(a) Throughput.



(b) Abort rate.

Figure 5.7: Comparison with alternative STM designs (Bank).

with respect to PR-STM in read-dominated workloads and of around 20× with respect to JVSTM-GPU and JVSTM in update-dominated workloads. Lets analyze more in depth the reasons underlying the performance gains of CSMV.

%ROTs	Total/Wasted time per TX (ms)					
	CSMV		PR-STM		JVSTM-GPU	
	Total	Wasted	Total	Wasted	Total	Wasted
1	4.05	1.92	80	30	80	33
10	3.99	1.59	631	108	72	29
25	3.66	0.91	1717	380	59	23
50	3.41	0.037	2635	448	38	12
75	4.54	0.003	4098	458	19	3.46
90	5.26	0.001	8846	4252	8.47	0.035
99	5.79	0.000	5410	0	8.32	0.001

Table 5.5: Total time and wasted time for a transaction. (in milliseconds) (Bank)

As already mentioned, the largest speed-ups with respect to PR-STM can be observed in correspondence of the largest considered percentages of ROTs (90% and 99%). Looking at the second plot in Figure 5.7, which reports the abort rate as a function of the percentage of ROTs it is possible to conclude that there are indeed two main factors contributing to CSMV's superior performance w.r.t. PR-STM: (i) Due to its single version nature, PR-STM is unable to avoid contention between read-only and update transactions (unlike CSMV). As a consequence, when the workload comprises 90% of ROTs, these are likely to run concurrently and conflict with some update transaction, suffering of frequent aborts. (ii) Even in absence of contention between ROTs and update transactions, PR-STM incurs large instrumentation and validation overheads when processing long ROTs, which, conversely, CSMV avoids thanks to its MV scheme. This phenomenon is clearly visible when the percentage of ROTs is set to 99%: in these settings the abort rate for PR-STM is close to 0, given that the likelihood of concurrency (and, thus, contention) with update transactions is very low. Yet, the speed-up achieved by CSMV vs PR-STM remains massive (approx. 1000×). This result can be explained by analyzing the data in Table 5.5, which reports the average total execution time and the wasted time (due to aborts) for a transaction. As seen on the table, with 99% of ROTs, the wasted time is 0 for both CSMV and PR-STM. However, the total execution time is approximately 1000 times larger in PR-STM, which, unlike CSMV, needs to track the read accesses of ROTs and undergo expensive validations.

Table 5.6: Breakdown of the main commit phases for JVSTM-GPU and CSMV (in miliseconds) (Bank).

	JVSTM-GPU				**CSMV**				
%ROTs	Commit	Validation	ATR Insert	Other	Commit	Wait server	Validation	ATR Insert	Other
1	47.082	25.55	2.335	21.308	2.05	0.612	1.412	0.026	0.021
10	41.627	22.268	2.114	19.159	1.735	0.399	1.311	0.025	0.02
25	33.441	17.328	1.763	15.949	1.075	0.026	1.029	0.019	0.016
50	19.176	9.385	1.135	9.691	0.204	0.002	0.195	0.006	0.004
75	7.475	3.256	0.555	4.173	0.042	0.001	0.038	0.002	0.002
90	1.058	0.088	0.173	0.959	0.011	0	0.009	0.001	0.001
99	0.56	0.004	0.016	0.555	0.001	0	0	0	0.001

Focusing on the comparison with JVSTM-GPU, the benefits deriving from CSMV's GPU-oriented design are evident. In this case the larger speed-ups are observed in the presence of a larger fraction of update transactions. This is expected, given that these two systems manage ROTs in the same way, while adopting different mechanisms to regulate the commit process of update transactions. The reason underlying the performance gains of CSMV w.r.t. JVSTM-GPU in update intensive workloads can be found again in Table 5.6: with 1% ROTs, the total transaction execution time with JVSTM-GPU is about 20× larger than with CSMV, since the latter can substantially accelerate the commit process of update transactions as shown by the data reported in Table 5.6. This table provides a break-down of the execution times of the main phases of the commit process for CSMV and JVSTM-GPU. The first observation that can be drawn is that the total commit time for JVSTM-GPU is approximately 23× larger than for CSMV when considering the same scenario of 1% ROTs. This gap is due to two dominant causes: (i) the validation time, which in JVSTM-GPU incurs the cost of accessing the ATR, accounting for about 50% of the commit latency and taking 20× longer than in CSMV; (ii) the time during which threads block due to warp divergence, which in JVSTM-GPU accounts for almost 45% of the time, being instead negligible for CSMV thanks to its GPU-optimized design is optimized to take advantage of the SIMT execution.

Finally, the comparison with JVSTM highlights that, thanks to CSMV's design, it is possible to take advantage of GPUs to accelerate also challenging irregular applications running on state of the art STMs for CPUs by up to approx. 20×/10× in update/read dominated workloads, respectively.

MemcachedGPU. MemcachedGPU is configured to have a total capacity of 1M items and conduct a sensitivity study by varying its geometry. Specifically, the cache associativity varies from 4 to 256. Recall that this has a direct impact on the maximum number of elements read by the transactions that encapsulate both GETs and PUTs operations (§ 5.1).

Figure 5.8 compares the performance of CSMV with all the previously considered baselines, except JVSTM (which is omitted since I do not have access to a JVSTM-based implementation of Mem-

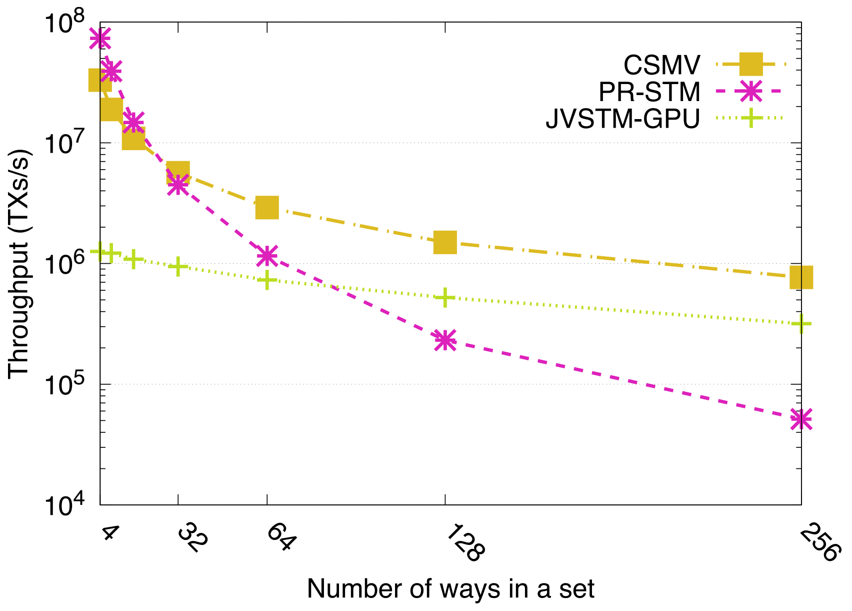


Figure 5.8: Comparison with alternative STM designs (MemcachedGPU)

Table 5.7: Total and wasted time for a transaction (in milliseconds) (Memcached).

Total/Wasted time per TX (ms)

ways	JVSTM-GPU		CSMV		PR-STM	
	Total	Wasted	Total	Wasted	Total	Wasted
4	0.844	0	0.027	0	0.016	0
8	0.878	0.001	0.048	0.003	0.039	0
16	0.950	0	0.088	0.007	0.113	0
32	1.139	0.001	0.168	0.012	0.407	0
64	1.446	0.001	0.326	0.025	1.615	0
128	1.981	0.001	0.635	0.050	8.021	0
256	2.540	0.001	1.288	0.116	38.510	0

cachedGPU). Also in these settings, CSMV achieves substantial gains with respect to PR-STM as the size of the ROTs (i.e., the number of ways) increase, namely up to approx. 15× when using 256 ways. The analysis of the data reported in Table 5.7 allows to conclude that the main source of inefficiency for PR-STM is not contention among transactions (the abort rate and wasted time are close to 0 for all solutions in this scenario), but rather the large overhead imposed by PR-STM's read-tracking and validation mechanisms (see Total time for PR-STM in Table 5.7).

Moreover the performance gains versus PR-STM diminishes as the number of ways decreases, causing transactions to accordingly access a smaller number of elements. Indeed, as the number of ways drop to 4, PR-STM outperforms CSMV by approx. 60%. This is expectable, keeping into account that CSMV, as typical of MV scheme, is optimized for long running ROTs.

Finally, when compared with JVSTM-GPU, CSMV is consistently faster, with throughput gains that range from approx. 50× (4 ways) to approx. 2× (256 ways). This trend can be explained by considering that the smaller the number of ways, the shortest the ROTs, the larger the relative impact on throughput from update transactions — for which CSMV, as already mentioned when discussing the Bank benchmark, employs a much more efficient commit procedure. This observation is also confirmed by the data in Table 5.8), that reports detailed information on the execution times of the main phases of the commit procedure with JVSTM-GPU and CSMV. Also this benchmarks confirms the effectiveness of CSMV's design in accelerating the validation phase of conventional MV algorithms. especially in high throughput scenarios (i.e., small number of ways). In such cases, in fact, the number of concurrently committed transactions that need to be considered during validation naturally increases if, as in this case, the %ROTs remains constant. This, on the one hand, exacerbates the scalability limitations of JVSTM-GPU, while, on the other hand, amplifying the throughput gains enabled by CSMV's optimized commit logic.

Table 5.8: Breakdown of the main commit phases for JVSTM-GPU and CSMV (in microseconds) (Memcached).

	JVSTM-GPU				**CSMV**				
#ways	Commit	Validation	ATR Insert	Other	Commit	Wait server	Validation	ATR Insert	Other
1	823.3	295.7	1.004	526.6	9.34	0.009	8.84	0.038	0.452
10	835.5	308.9	0.982	525.6	10.80	0.010	10.28	0.035	0.477
25	865.5	344.2	0.941	520.4	11.95	0.010	11.40	0.034	0.498
50	935.3	390.0	0.865	544.5	18.55	0.008	18.00	0.034	0.509
75	1094.9	504.7	0.766	589.4	30.92	0.009	30.37	0.033	0.509
90	1298.2	670.0	0.614	627.6	53.14	0.012	52.58	0.029	0.515
99	1364.1	830.3	0.432	533.4	127.64	0.012	127.09	0.035	0.504

Chapter 6

Conclusions

This thesis analysed the state of the art on STM for GPU systems and presented CSMV, a multi-versioned STM for GPUs that adopts an innovative design tailored to take advantage of the unique architectural characteristics of these massively parallel, throughput-oriented computing devices. To the best of my knowledge, CSMV is the first TM in the literature to adopt a client-server architecture that decouples transaction execution from the commit process. Such a design provides two substantial benefits: (i) it allows for accessing the global metadata required to synchronize transaction execution via fast on chip; (ii) it enables the implementation of highly efficient collaborative validation schemes, designed to take full advantage of the SIMT execution model and intra-warp communication primitives of GPUs. Further, CSMV introduces a number of algorithmic and system-level optimizations that operate in synergy to enhance the efficiency of the commit procedure by: (i) offloading crucial phases of the commit logic to the client-side and (ii) reduce the frequency with which the commit resorts to employing expensive synchronization primitives.

The proposed experimental study highlighted that CSMV can achieve up to 3 orders of magnitude speed-ups with respect to state of the art STMs for GPUs, as well accelerating by up to 20$\times$ irregular applications running on state of the art STMs for CPUs.

6.1 Future Work

Many challenges rose along the development of CSMV, some of which remain unsolved even in the presented approach. As a matter of fact, potential approaches for tackling said challenged have been already designed during the course of this work, but ultimately, due to time constraints, they were not implemented, being instead exposed in this section.

The centralized nature of the of the server imposes an obvious upper bound on the scalability of this method. As already discussed in Chapter 5, on current GPU architectures, a centralized approach, like the one of CSMV, does provide significant speed-ups with respect to traditional/CPU-inspired approaches in which, conversely, the commit phase is completely decentralized. However, a single server has a limit on a number of worker warps it can support which in turn limits how many client's requests can be handled simultaneously, and also one server has limited shared memory, limiting the size available for the ATR which can hinder CSMV as seen on Chapter 5.

To overcome both issues, one can add more dedicated server SMs. The synchronization between these new server nodes can be as follows:

- $i)$ a portion of the ATR is stored on each node, essentially multiplying the size of the ATR by the number servers used. Worker threads from different servers validate the same transaction against the portion of the ATR stored in that server node, and all the threads among all servers communicate in the end to determine the fate of that transaction. Only one of the server SMs needs to add the transaction to their respective portion of the record.

- $ii)$ All server SMs keep the same copy of the ATR, and all transactions validate and try to insert different transactions, which essentially multiplies the number of available workers by the number of server used. Different server threads compete to insert their transactions which means they need to CAS on global memory, which, as seen for the case of the CSMV-ts and CSMV-gb variants, can add significant overhead. On the other hand, note that in such a scheme the contention on the ATR would be immensely relieved relatively to a fully decentralized commit scenarios, as, in the former case, the ATR would be updated by a relatively small number of servers and not by any active thread on the GPU, as in the latter case. When a thread wins the CAS, this also serves to notify the other SMs so that the winning transaction can also be added to their copy of the ATR.

The first version augments the size of the ATR while the second one increases the overall concurrency of the server. Note that the use of a version does not preclude the usage of the other, they both can be used simultaneous, for example: four server nodes, where two of them keep the same copy of the ATR, each of these extended by a second server that expands said ATR.

Once the architecture is made capable of supporting multiple servers, this also opens the optimization problem of how many server SMs to allocate, given that this opens a trade-off with the number of SMs available for running client code (including non-transactional one).

Finally, another key challenge is related to the fact that the limited capacity of the scratchpad memory can lead to spurious aborts in workloads dominated by small update transactions. To tackle this problem, I have designed, but not implemented, an alternative validation algorithm that ensures that that all the transactions that reach the server are given the chance to be validated, before any of them is added to the ATR. Assume an incoming batch of m candidate transactions, denoted C, and an ATR that maintains n transactions, where $n¡m$.

The intuition is to operate in rounds, where each round works according to the following phases: $1)$ validate the read set of the incoming transactions against the current ATR, remove from C the transactions that fail this validation step. $2)$ pick the next n transactions of C (e.g., using the order in the batch) that do not mutually conflict, and can as such all be committed and add them to the ATR. $3)$ remove these transactions from C and repeat step 1 until C is empty.

This approach has the advantage of keeping in scratchpad memory the write-set of the transactions that are used during phase 1), which is arguably the most computationally expensive. The algorithm opens also future research avenues aimed at designing parallelization schemes optimized to take advantage of modern GPU architectures.

Bibliography

[1] R. Nasre, M. Burtscher, and K. Pingali. Morph algorithms on gpus. *SIGPLAN Not.*, 48(8):147–156, Feb. 2013. ISSN 0362-1340. doi: 10.1145/2517327.2442531. URL https://doi.org/10.1145/2517327.2442531.

[2] J. Nelson and R. Palmieri. Don't forget about synchronization! a case study of k-means on gpu. In *PMAM'19 Workshop*, PMAM'19, page 11–20, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450362900. doi: 10.1145/3303084.3309488. URL https://doi.org/10.1145/3303084.3309488.

[3] M. A. Awad, S. Ashkiani, R. Johnson, M. Farach-Colton, and J. D. Owens. Engineering a high-performance gpu b-tree. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, PPoPP '19, page 145–157, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450362252. doi: 10.1145/3293883.3295706. URL https://doi.org/10.1145/3293883.3295706.

[4] Z. Yan, Y. Lin, L. Peng, and W. Zhang. Harmonia: A high throughput b+tree for gpus. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, PPoPP '19, page 133–144, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450362252. doi: 10.1145/3293883.3295704. URL https://doi.org/10.1145/3293883.3295704.

[5] N. Moscovici, N. Cohen, and E. Petrank. A gpu-friendly skiplist algorithm. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 246–259, 2017. doi: 10.1109/PACT.2017.13.

[6] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *ISCA'93: Proceedings of the 20th annual international symposium on computer architecture*, volume 21, page 289–300, New York, NY, USA, May 1993. ACM. doi: 10.1145/173682.165164. URL https://doi.org/10.1145/173682.165164.

[7] C. J. Rossbach, O. S. Hofmann, and E. Witchel. Is transactional programming actually easier? In *PPoPP '10*, page 47, 2010. ISBN 9781605587080. doi: 10.1145/1693453.1693462. URL http://portal.acm.org/citation.cfm?doid=1693453.1693462.

[8] D. Dice, O. Shalev, and N. Shavit. Transactional locking ii. In *Proceedings of the 20th International Conference on Distributed Computing*, DISC'06, page 194–208, Berlin, Heidelberg, 2006.

Springer-Verlag. ISBN 3540446249. doi: 10.1007/11864219_14. URL https://doi.org/10.1007/11864219_14.

[9] P. Felber, C. Fetzer, and T. Riegel. Dynamic performance tuning of word-based software transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, page 237–246, New York, NY, USA, 2008. Association for Computing Machinery. ISBN 9781595937957. doi: 10.1145/1345206.1345241. URL https://doi.org/10.1145/1345206.1345241.

[10] A. Dragojevic, R. Guerraoui, and M. Kapalka. Stretching transactional memory. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, page 155–165, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605583921. doi: 10.1145/1542476.1542494. URL https://doi.org/10.1145/1542476.1542494.

[11] L. Dalessandro, M. F. Spear, and M. L. Scott. Norec: Streamlining stm by abolishing ownership records. *SIGPLAN Not.*, 45(5):67–78, Jan. 2010. ISSN 0362-1340. doi: 10.1145/1837853.1693464. URL https://doi.org/10.1145/1837853.1693464.

[12] A. Wang, M. Gaudet, P. Wu, J. N. Amaral, M. Ohmacht, C. Barton, R. Silvera, and M. Michael. Evaluation of blue gene/q hardware support for transactional memories. In *PACT '12*, page 127–136. ACM, 2012. ISBN 9781450311823. doi: 10.1145/2370816.2370836. URL https://doi.org/10.1145/2370816.2370836.

[13] D. Cederman, P. Tsigas, and M. T. Chaudhry. Towards a software transactional memory for graphics processors. In *Proceedings of the 10th Eurographics Conference on Parallel Graphics and Visualization*, EG PGV'10, page 121–129, Goslar, DEU, 2010. Eurographics Association. ISBN 9783905674217.

[14] Y. Xu, R. Wang, N. Goswami, T. Li, L. Gao, and D. Qian. Software transactional memory for gpu architectures. In *CGO '14*. ACM, 2014. ISBN 9781450326704. doi: 10.1145/2544137.2544139. URL https://doi.org/10.1145/2544137.2544139.

[15] A. Holey and A. Zhai. Lightweight software transactions on gpus. In *2014 43rd International Conference on Parallel Processing*, pages 461–470, Sep. 2014. doi: 10.1109/ICPP.2014.55.

[16] 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*, volume 9233, pages 361–372, 08 2015. ISBN 978-3-662-48095-3. doi: 10.1007/978-3-662-48096-0_28.

[17] Q. Shen, C. Sharp, R. Davison, G. Ushaw, R. Ranjan, A. Y. Zomaya, and G. Morgan. A general purpose contention manager for software transactions on the gpu. *J. Parallel Distrib. Comput.*, 139:1–17, 2020. ISSN 0743-7315. doi: https://doi.org/10.1016/j.jpdc.2019.12.018. URL https://www.sciencedirect.com/science/article/pii/S0743731519301376.

[18] S. Mittal and J. S. Vetter. A Survey of CPU-GPU Heterogeneous Computing Techniques. *ACM Comput. Surv.*, 47(4):69:1–69:35, July 2015. ISSN 0360-0300. doi: 10.1145/2788396.

[19] J. Nelson and R. Palmieri. Don't forget about synchronization! a case study of k-means on gpu. In *Proceedings of the 10th International Workshop on Programming Models and Applications for Multicores and Manycores*, PMAM'19, page 11–20, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450362900. doi: 10.1145/3303084.3309488. URL `https://doi.org/10.1145/3303084.3309488`.

[20] D. Dice, O. Shalev, and N. Shavit. Transactional locking ii. In *Proceedings of the 20th International Conference on Distributed Computing*, DISC'06, page 194–208, Berlin, Heidelberg, 2006. Springer-Verlag. ISBN 3540446249. doi: 10.1007/11864219_14. URL `https://doi.org/10.1007/11864219_14`.

[21] P. Felber, C. Fetzer, and T. Riegel. Dynamic performance tuning of word-based software transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, page 237–246, New York, NY, USA, 2008. Association for Computing Machinery. ISBN 9781595937957. doi: 10.1145/1345206.1345241. URL `https://doi.org/10.1145/1345206.1345241`.

[22] A. Dragojevic, R. Guerraoui, and M. Kapalka. Stretching transactional memory. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, page 155–165, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605583921. doi: 10.1145/1542476.1542494. URL `https://doi.org/10.1145/1542476.1542494`.

[23] L. Dalessandro, M. F. Spear, and M. L. Scott. Norec: Streamlining stm by abolishing ownership records. *SIGPLAN Not.*, 45(5):67–78, Jan. 2010. ISSN 0362-1340. doi: 10.1145/1837853.1693464. URL `https://doi.org/10.1145/1837853.1693464`.

[24] A. Wang, M. Gaudet, P. Wu, J. N. Amaral, M. Ohmacht, C. Barton, R. Silvera, and M. Michael. Evaluation of blue gene/q hardware support for transactional memories. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, page 127–136, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450311823. doi: 10.1145/2370816.2370836. URL `https://doi.org/10.1145/2370816.2370836`.

[25] D. Cederman, P. Tsigas, and M. T. Chaudhry. Towards a software transactional memory for graphics processors. In *Proceedings of the 10th Eurographics Conference on Parallel Graphics and Visualization*, EG PGV'10, page 121–129, Goslar, DEU, 2010. Eurographics Association. ISBN 9783905674217.

[26] 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*, CGO '14, page 1–10, New York, NY, USA, 2014. Association for Computing

Machinery. ISBN 9781450326704. doi: 10.1145/2544137.2544139. URL https://doi.org/10.1145/2544137.2544139.

[27] A. Holey and A. Zhai. Lightweight software transactions on gpus. In *2014 43rd International Conference on Parallel Processing*, pages 461–470, Sep. 2014. doi: 10.1109/ICPP.2014.55.

[28] Q. Shen, C. Sharp, W. Blewitt, G. Ushaw, and G. Morgan. Pr-stm: Priority rule based software transactions for the gpu. volume 9233, pages 361–372, 08 2015. ISBN 978-3-662-48095-3. doi: 10.1007/978-3-662-48096-0_28.

[29] J. Cachopo and A. Rito-Silva. Versioned boxes as the basis for memory transactions. *Science of Computer Programming*, 63(2):172–185, 2006. ISSN 01676423. doi: 10.1016/j.scico.2006.05.009.

[30] H. Attiya and E. Hillel. Single-version stms can be multi-version permissive. In *ICDCN*, 2011.

[31] R. Palmieri, F. Quaglia, P. Romano, and N. Carvalho. Evaluating database-oriented replication schemes in software transactional memory systems. In *2010 IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW)*, pages 1–8, 2010. doi: 10.1109/IPDPSW.2010.5470866.

[32] I. Keidar and D. Perelman. *Multi-versioning in Transactional Memory*, pages 150–165. Springer International Publishing, Cham, 2015. ISBN 978-3-319-14720-8. doi: 10.1007/978-3-319-14720-8_7. URL https://doi.org/10.1007/978-3-319-14720-8_7.

[33] N. Bernstein, Philip A and Hadzilacos, Vassos and Goodman. *Concurrency control and recovery in database systems.* Addison-Wesley Pub. Co. Inc., Reading, MA, 1987.

[34] 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 - SoCC '15*, pages 43–57, New York, USA, 2015. ACM Press. ISBN 9781450336512. doi: 10.1145/2806777.2806836. URL http://dl.acm.org/citation.cfm?doid=2806777.2806836.

[35] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, ISCA '93, page 289–300, New York, NY, USA, 1993. Association for Computing Machinery. ISBN 0818638109. doi: 10.1145/165123.165164. URL https://doi.org/10.1145/165123.165164.

[36] R. Guerraoui and P. Romano, editors. *Transactional Memory. Foundations, Algorithms, Tools, and Applications - COST Action Euro-TM IC1001*, volume 8913 of *Lecture Notes in Computer Science*. Springer, 2015. ISBN 978-3-319-14719-2. doi: 10.1007/978-3-319-14720-8. URL https://doi.org/10.1007/978-3-319-14720-8.

[37] P. A. Bernstein and N. Goodman. Multiversion concurrency control—theory and algorithms. *ACM Trans. Database Syst.*, 8(4):465–483, Dec. 1983. ISSN 0362-5915. doi: 10.1145/319996.319998. URL https://doi.org/10.1145/319996.319998.

[38] P. di Sanzo, B. Ciciani, F. Quaglia, and P. Romano. A performance model of multi-version concurrency control. In *MASCOTS'08*, 2008. doi: 10.1109/MASCOT.2008.4770559.

[39] Y. Wu, J. Arulraj, J. Lin, R. Xian, and A. Pavlo. An empirical evaluation of in-memory multi-version concurrency control. *Proc. VLDB Endow.*, 10(7):781–792, Mar. 2017. ISSN 2150-8097. doi: 10.14778/3067421.3067427. URL `https://doi.org/10.14778/3067421.3067427`.

[40] S. M. Fernandes and J. a. Cachopo. Lock-free and scalable multi-version software transactional memory. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP '11, page 179–188, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450301190. doi: 10.1145/1941553.1941579. URL `https://doi.org/10.1145/1941553.1941579`.

[41] N. Carvalho, J. a. Cachopo, L. Rodrigues, and A. R. Silva. Versioned transactional shared memory for the fénixedu web application. In *Proceedings of the 2nd Workshop on Dependable Distributed Data Management*, WDDDM '08, page 15–18, New York, NY, USA, 2008. Association for Computing Machinery. ISBN 9781605581217. doi: 10.1145/1435523.1435526. URL `https://doi.org/10.1145/1435523.1435526`.

[42] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990. ISSN 0164-0925. doi: 10.1145/78969.78972. URL `https://doi.org/10.1145/78969.78972`.

[43] C. H. Papadimitriou. The serializability of concurrent database updates. *J. ACM*, 26(4):631–653, Oct. 1979. ISSN 0004-5411. doi: 10.1145/322154.322158. URL `https://doi.org/10.1145/322154.322158`.

[44] 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*, PPoPP '08, page 175–184, New York, NY, USA, 2008. Association for Computing Machinery. ISBN 9781595937957. doi: 10.1145/1345206.1345233. URL `https://doi.org/10.1145/1345206.1345233`.

[45] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. Nvidia tesla: A unified graphics and computing architecture. *IEEE Micro*, 28(2):39–55, Mar. 2008. ISSN 0272-1732. doi: 10.1109/MM.2008.31. URL `https://doi.org/10.1109/MM.2008.31`.

[46] Cuda programming guide. URL `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`.

[47] L. Sousa and P. Tomás. Graphics processing units, 2019. URL `https://fenix.tecnico.ulisboa.pt/downloadFile/1689468335624141/14%20-%20Graphics%20Processing%20Units.pdf`.

[48] N. Corporation. Whitepaper - nvidia tesla p100 - the most advanced datacenter accelerator ever built - featuring pascal gp100, the world's fastest gpu, 2016. URL `https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf`.

[49] N. Corporation. Whitepaper - nvidia tesla v100 gpu architecture - the world's most advanced data center gpu, 2017. URL https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf.

[50] N. Corporation. Whitepaper - nvidia turing gpu architecture - graphics reinvented, 2018. URL https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf.

[51] N. Corporation. Whitepaper - nvidia ampere ga102 gpu architecture - the ultimate play, 2020. URL https://images.nvidia.com/aem-dam/en-zz/Solutions/geforce/ampere/pdf/NVIDIA-ampere-GA102-GPU-Architecture-Whitepaper-V1.pdf.

[52] A. Li, G.-J. van den Braak, H. Corporaal, and A. Kumar. Fine-grained synchronizations and dataflow programming on gpus. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, ICS '15, page 109–118, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450335591. doi: 10.1145/2751205.2751232. URL https://doi.org/10.1145/2751205.2751232.

[53] L. Durant, O. Giroux, M. Harris, and N. Stam. Inside volta: The world's most advanced data center gpu, Oct 2020. URL https://developer.nvidia.com/blog/inside-volta/.

[54] K. Wang, D. Fussell, and C. Lin. Fast fine-grained global synchronization on gpus. In *ASPLOS'19*, ASPLOS'19, page 793–806, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450362405. doi: 10.1145/3297858.3304055. URL https://doi.org/10.1145/3297858.3304055.

[55] W. W. L. 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*, MICRO-44, page 296–307, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450310536. doi: 10.1145/2155620.2155655. URL https://doi.org/10.1145/2155620.2155655.

[56] W. W. L. Fung and T. M. Aamodt. Energy efficient gpu transactional memory via space-time optimizations. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, page 408–420, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450326384. doi: 10.1145/2540708.2540743. URL https://doi.org/10.1145/2540708.2540743.

[57] S. Chen and L. Peng. Efficient gpu hardware transactional memory through early conflict resolution. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 274–284, March 2016. doi: 10.1109/HPCA.2016.7446071.

[58] S. Chen, L. Peng, and S. Irving. Accelerating gpu hardware transactional memory with snapshot isolation. *SIGARCH Comput. Archit. News*, 45(2):282–294, June 2017. ISSN 0163-5964. doi: 10.1145/3140659.3080204. URL https://doi.org/10.1145/3140659.3080204.

[59] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer. Software transactional memory for dynamic-sized data structures. In *Proceedings of the Twenty-Second Annual Symposium on Principles of Distributed Computing*, PODC '03, page 92–101, New York, NY, USA, 2003. Association for Computing Machinery. ISBN 1581137087. doi: 10.1145/872035.872048. URL https://doi.org/10.1145/872035.872048.

[60] D. Castro, P. Romano, A. Ilic, and A. M. Khan. Hetm: Transactional memory for heterogeneous systems. In *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 232–244, Los Alamitos, CA, USA, sep 2019. IEEE Computer Society. doi: 10.1109/PACT.2019.00026. URL https://doi.ieeecomputersociety.org/10.1109/PACT.2019.00026.

[61] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload analysis of a large-scale key-value store. 40(1), 2012. ISSN 0163-5999. doi: 10.1145/2318857.2254766. URL https://doi.org/10.1145/2318857.2254766.