Debunking the CPU+GPU Processing with Transactional Memory

Diogo Nunes
DEEC
Instituto Superior Técnico
Lisbon, Portugal

***Abstract*—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 paper introduces CSMV (Client Server Multiversioned), a multi-versioned Software TM (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 transaction (ii) it allows for implementing highly efficient collaborative commit procedures, tailored to take full advantage of the architectural characteristics of GPUs.

Via an extensive experimental study, we show that CSMV achieves up to 3 orders of magnitude speed-ups with respect to state of the art STMs for GPUs and that it can accelerate by up to 20× irregular applications running on state of the art STMs for CPUs.

***Index Terms*—Multi-Version Concurrency Control, Synchronization, GPU, Transaction, Transactional Memory**

I. 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].

In this work we present a novel Software Transactional Memory (STM) algorithm that we named CSMV (Client-Server Multi-Versioned). When compared to existing (S)TM solutions, CSMV has a number of innovative features.

1) CSMV tackles what we 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 terminology). Such a design, inherited by TM algorithms originally conceived for CPUs, generate prohibitive overheads when adopted in GPUs supporting thousands of concurrent threads, as we will show 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 does it avoid atomic operations targeting global memory, allowing them to operate instead on the fast scratchpad memory (shared memory in CUDA terminology) 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 multi-versioned (MV) concurrency control algorithm for GPUs. The advantages of MV approaches have been long studied both in the database and TM communities [18]–[20]: 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 [20], 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.

We evaluated CSMV using both synthetic workloads and a realistic application, MemcachedGPU [21], comparing it to a state of the art single versioned STM (PR-STM [16], [17]) and to a "conventional" multi-versioned implementation, which we call JVSTM-GPU, obtained by porting to GPU the algorithm of JVSTM, a state of the art MV STM for CPUs. Our 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.

II. Related Work

Multiversioning concurrency control. MV concurrency control schemes have been long investigated in the literature both in the database [22]–[24] and TM community [18], [25], [26]. From a theoretical standpoint, the key benefit of MV approaches are: i) Read-Only Transactions (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 [20], [27], which are prone to suffer from starvation in single-versioned approaches (due to the high likelihood to conflict with concurrent update transactions).

When compared to existing MV algorithms, CSMV adopts a unique client-server design and a number of associated optimizations (e.g., client-side pre-validation and server-side collaborative validation). Such a design aims to take advantage of the architectural characteristics of GPU systems, in particular their massive parallelism and the availability of both (i) fast communication channels among the threads in the same block via on chip memory and (ii) high latency global communication channels via off chip memory, whose throughput is strongly affected by the locality of accesses within the same warp.

STMs for GPUs. To the best of our knowkledge, Cederman et al. [13] were the first to propose the use of TM in GPUs. In more detail, they introduced two alternative single-versioned software TM (STM) implementations, whose algorithms can be seen as a relatively straightforward porting of TM designs previously proposed for multi-core CPUs (e.g., [28], [29]).

Xu et al. [14] later proposed a single-versioned STM for GPUs that combines time-based validation [9] - which is fast but prone to spurious aborts - with value-based validation [11] - which is accurate but more expensive. Also in this case, the design is strongly inspired to STM algorithms previously proposed in the CPU domain [8], [9]. Holey and Zhai [15] investigated alternative single-versioned STM designs, evaluating for the first time in GPUs the use of approaches, originally proposed for CPUs, that rely on visible vs invisible reads and eager vs lazy conflict detection. The most recent STM for GPU that we are aware of is PR-STM [17], which also uses a single versioned, encounter-based locking design (analogously to one of the solutions presented by Holey and Zhai [15]) extended with a contention management strategy [30] aimed at reducing the likelihood of aborts in contention prone workloads.

Compared to these solutions, CSMV has a number of unique features: CSMV is the first STM for GPUs to adopt a multi-versioned concurrency control and a client-server design that spares from the overhead imposed by the use of atomic operations (e.g., CAS) operating on global memory (enabling instead the use of scratchpad memory). As we will show experimentally in Section IV, thanks to these features, CSMV achieves up to more than one order of magnitude speedups when compared to state of the art STM system like PR-STM [17].

HTMs for GPUs. While the works just mentioned above rely on software based approaches, some TM proposals for GPUs assume *ad hoc* hardware supports [31]–[34]. By leveraging hardware mechanisms, these approaches can alleviate, or even totally avoid, some sources of overhead that are inherent to software-based designs (e.g., instrumenting read and write accesses). However, unlike CSMV, these proposal cannot be employed on existing GPUs given their reliance on custom hardware mechanisms. Among these solutions, it is worth highlighting the proposal by Chen et al. [34], which, to the best of our knowledge, represents the only MV GPU TM proposed in the literature. Besides being a hardware-based implementation, this solution adopts a weaker consistency level (Snapshot Isolation [35]) than the one typically assumed by TM systems and by CSMV (Opacity [36]).

Concurrency in GPUs. While GPUs have traditionally focused on applications with regular parallelism, irregular applications in a range of domains (including machine learning [2], graph manipulations [1], dynamic programming [37], 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 [38].

In the literature on optimizing fine-grained locking, the idea of exploiting scratchpad memory is not new [37], [39]. Among these works, the solution that is more closely related to CSMV is the work by Wang et al [39]. This work proposed to delegate 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. 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.

III. CSMV

This section is devoted to presenting the design of CSMV. We do so in an incremental fashion, by first surveying, in Section III-A, the algorithm used by JVSTM, a state of the art multi-versioned STM for CPUs [20], [25]. This allows us to pinpoint the main sources of inefficiencies that arise when adopting a MV STM designed for CPUs on GPUs, and to justify the key design choices at the basis of CSMV. Finally, Section III-B presents the mechanisms that CSMV employs in order to address the shortcomings of existing TM describes.

A. Dissecting a MV STM for CPUs: JVSTM

This section aims to pinpoint the key factors that hinder the efficiency of MV STM algorithms originally proposed for CPUs, when straightforwardly adopted in GPUs. To this end, we consider the algorithm employed by JVSTM [20], a state of the art MV STM for CPUs, which can be seen as an archetype of this class of systems.

Data structures. In JVSTM each shared object is encapsulated within a Versioned Box (VBox), which stores 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: i) a global (logical) time stamp (GTS), which is read by transactions upon their start to establish which *snapshot* they should observe and that is incremented whenever an update transaction commits; ii) a list, called the Active Transaction Record (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, we recall, 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: i) 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; ii) 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.

Key challenges arising in GPUs. The adoption of the above described MV STM algorithm in GPUs raises two key challenges.

i) 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). Furthermore, during validation, transactions generate a large number of read accesses to 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.

ii) Limited parallelism. Except for the validation phase, all the remaining phases (insertion in the ATR and write-back) of the commit process 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.

B. Description of CSMV design

CSMV builds on the JVSTM algorithm and extends it with the following mechanisms that operate in synergy to address the challenges identified in Section III-A.

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, one of the available Streaming Multiprocessors (SMs) executes a specialized kernel that handles solely the commit phase; the remaining SMs runs 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 we call CTS (commit timestamp).

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 are responsible for the actual execution of the commit logic (except the write-back phase, see next) and for notifying the clients of the transactions' outcome.

Note that this design introduces the latency of a bi-directional communication between client and server along the critical path of execution of transactions. However, it allows for maintaining the ATR in the scratchpad memory: this accelerates both the transaction validation and its insertion in the ATR, increasing the maximum throughput achievable by these critical phases. As we will show experimentally, given the throughput-oriented nature of GPUs, this trade-off pays off enabling substantial throughput gains.

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. We tackle this issue as follows:

1) We 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 transaction writes a data item, it also reads it) [19], 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, we 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.

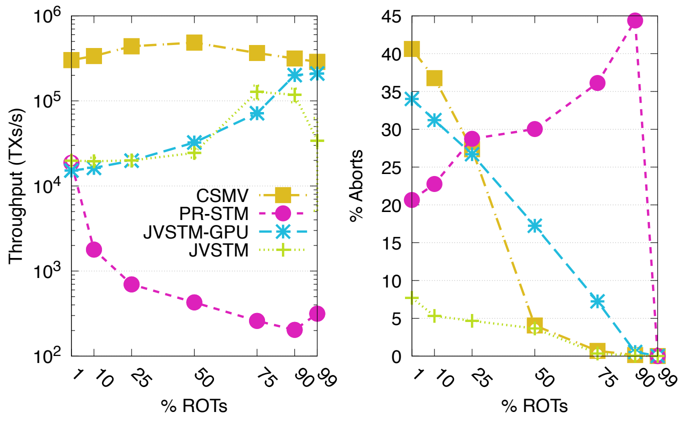
Client-side pre-validation. In order to streamline the server-side commit process, the transactions that execute within the same warp on the client side are validated to detect any intra-warp conflict. To this end, we exploit warp-level primitives (e.g., shuffle operations) to efficiently exchange the 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.

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 three main benefits: $i)$ it lower the frequency of manipulation of the ATR via CAS, which in turn reducing 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.

[1]This problem has been at least partially addressed in a later versions of JVSTM [25], 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.

TABLE I: Breakdown of the main commit phases for JVSTM-GPU and CSMV (in miliseconds) (Bank)

	JVSTM-GPU					CSMV						
%ROTs	Total	Valid.	Rec. Insert	Write-back	Divergence	Total	Wait server	Pre-Val.	Valid.	Rec. Insert	Write-back	Divergence
1	47.082	25.550	2.111	0.224	21.308	2.050	0.612	0.036	1.376	0.021	0.005	0.021
10	41.627	22.268	1.914	0.200	19.159	1.735	0.399	0.033	1.278	0.020	0.005	0.020
25	33.441	17.328	1.599	0.164	15.949	1.075	0.026	0.028	1.001	0.015	0.004	0.016
50	19.176	9.385	1.035	0.100	9.691	0.204	0.002	0.022	0.173	0.003	0.003	0.004
75	7.475	3.256	0.509	0.046	4.173	0.042	0.001	0.012	0.026	0.001	0.001	0.002
90	1.058	0.088	0.162	0.011	0.959	0.011	0	0.005	0.004	0	0.001	0.001
99	0.560	0.004	0.015	0.001	0.555	0.001	0	0	0	0	0	0.001



(a) Throughput.
(b) Abort rate.

Fig. 1: Comparison with alternative STM designs (Bank).

	Total/Wasted time per TX (ms)					
%ROTs	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 II: Total time and wasted time for a transaction. (in milliseconds) (Bank)

In more detail, we designed the message-passing protocol between client and server to ensure that a worker warp in the server is requested to concurrently process 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 we 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 of 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.

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, we designed a collaborative validation scheme 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. Each thread t in a worker warp targets 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.

IV. Evaluation

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

- 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? (§ IV-B)
- To what extent do the various mechanisms leveraged by CSMV contribute to enhance its efficiency? (§ IV-C)

A. Experimental setup

We evaluate CSMV by using two benchmarks: the Bank benchmark [28] and MemcachedGPU [21].

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 types of transactions: (i) update transactions that transfer a random amount of money between two bank accounts; (ii) ROTs, that

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

	JVSTM-GPU					CSMV						
ways	Total	Valid.	Rec. Insert	Write-back	Divergence	Total	Wait server	Pre-Val.	Valid.	Rec. Insert	Write-back	Divergence
4	823.3	295.7	0.987	0.017	526.6	9.34	0.009	0.279	8.564	0.014	0.023	0.452
8	835.5	308.9	0.965	0.018	525.6	10.80	0.010	0.305	9.971	0.010	0.024	0.477
16	865.5	344.2	0.922	0.019	520.4	11.95	0.010	0.376	11.029	0.009	0.025	0.498
32	935.3	390.0	0.842	0.023	544.5	18.55	0.008	0.540	17.460	0.009	0.025	0.509
64	1094.9	504.7	0.751	0.014	589.4	30.92	0.009	0.873	29.495	0.009	0.024	0.509
128	1298.2	670.0	0.594	0.020	627.6	53.14	0.012	1.533	51.047	0.006	0.023	0.515
256	1364.1	830.3	0.406	0.025	533.4	127.64	0.012	2.907	124.185	0.012	0.023	0.504

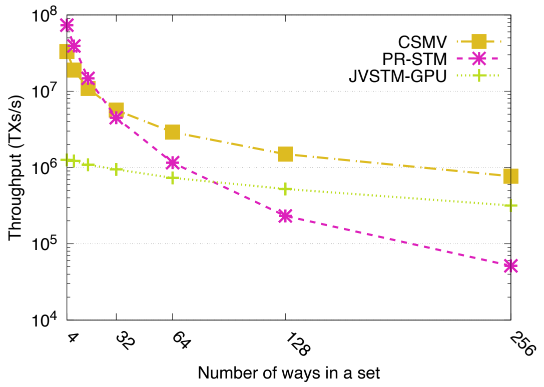


Fig. 2: Comparison with alternative STM designs (MemcachedGPU)

read all the accounts and compute the total balance of the simulated bank.

MemcachedGPU [21], [40] 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 [21] and later on [40] 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. We use keys/values 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, we access the key space 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. [41] .

We evaluate CSMV and the other STMs for GPUs using a machine equipped with an Nvidia GeForce GTX 1080 Ti (GP102 micro architecture, 28 SMs, 11 GB of RAM, CUDA v10.0), an i7-2600k CPU and 8GB of RAM. When evaluating STMs for CPUs we 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.

TABLE IV: 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

To ensure a fair comparison, we configure all the STMs for GPUs to use the same number of thread blocks and threads per block, namely 28 (i.e., as many as the number of available SM in our GPU) and 64, respectively. When testing STMs for CPUs, we use as many threads as available hardware threads. All the reported results were obtained as the average of at least 3 runs.

B. *Comparison with state of the art STMs*

This section is devoted to 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 (§ II);
2) JVSTM [25], a state of the art STM for CPUs that, analogously to CSMV, adopts a MV scheme (§ II, §III-A);
3) a MV STM for GPUs, which we refer to as JVSTM-GPU, which we obtained by directly porting the JVSTM algorithm to CUDA and that can be also regarded as a variant of CSMV from which we have removed all of the GPU-oriented algorithmic optimizations described in Section III-B.

Bank benchmark. Figure 1a reports the performance of all the above mentioned baselines with the Bank benchmark, which we 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 we vary the percentage of ROTs and report on the y-axis the throughput.

The plot in Figure 1 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 with respect to PR-STM in read-dominated workloads and of around 20× with respect to JVSTM-GPU and JVSTM in update-dominated workloads. Let us analyze more in depth the reasons underlying the performance gains of CSMV.

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 plot in Figure 1b, which reports the abort rate as a function of the percentage of ROTs we can 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 II, which reports the average total execution time and the wasted time (due to aborts) for a transaction. As we can see, 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.

Focusing on the comparison with JVSTM-GPU, we can appreciate the benefits deriving from CSMV's GPU-oriented design. 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 I: 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 I. 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

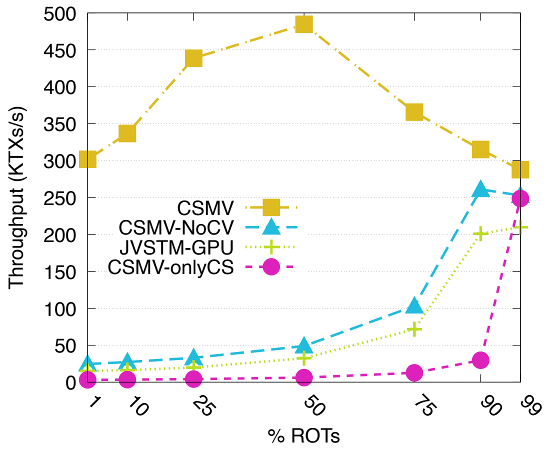


Fig. 3: Impact of selectively disabling CSMV optimizations (Bank)

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, that is developed 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. We configure MemcachedGPU to have a total capacity of 1M items and conduct a sensitivity study by varying its geometry. Specifically, we vary the cache associativity 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 (§ IV-A).

Figure 2 compares the performance of CSMV with all the previously considered baselines, except JVSTM (which we omit since we do not have access to a JVSTM-based implementation of MemcachedGPU). 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 IV allows us 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 IV).

We also observe that the performance gains versus PR-STM diminish 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 in mind 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 shorter 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 III, that reports detailed information on the execution times of the main phases of the commit procedure with JVSTM-GPU and CSMV. Also this benchmark 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.

C. *Impact of optimizations*

As already mentioned, CSMV makes uses of a number of mechanisms designed to take advantage of the architectural characteristics of GPUs and enhance efficiency. In this section we present a study that aims at evaluate their impact and interplay.

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

The first variant that we consider is obtained by disabling the collaborative validation mechanism. We shall refer to this variant as CSMV-NoCV. We then derive a second variant, 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.

We report in Figure 3 the throughput obtained by these two variants of CSMV using the Bank benchmark, with the same workload settings considered in Section IV-B. We include in the plots also CSMV and JVSTM-GPU. We omit reporting the abort rates, since the abort rates of both variants are very similar to those of CSMV (which we have already shown in Figure 1b).

The data in Figure 3 allows us to draw two main conclusions:

- The collaborative validation scheme (which is disabled in CSMV-NoCV), has the strongest performance impact, especially in update-intensive workloads. 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).
- The employment of a "vanilla" client-server architecture, i.e., without the contribution of the additional mechanisms presented in Section III-B (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.

V. Conclusions

This paper 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 our 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 memory; (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.

Our 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× irregular applications running on state of the art STMs for CPUs.

References

[1] Rupesh Nasre, Martin Burtscher, and Keshav Pingali. Morph algorithms on gpus. *SIGPLAN Not.*, 48(8):147–156, February 2013.

[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.

[3] Muhammad A. Awad, Saman Ashkiani, Rob Johnson, Martín Farach-Colton, and John 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.

[4] Zhaofeng Yan, Yuzhe Lin, Lu Peng, and Weihua 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.

[5] Nurit Moscovici, Nachshon Cohen, and Erez Petrank. A gpu-friendly skiplist algorithm. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 246–259, 2017.

[6] Maurice Herlihy and J. Eliot 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.

[7] C. J. Rossbach, O. S. Hofmann, and E. Witchel. Is transactional programming actually easier? In *PPoPP '10*, page 47, 2010.

[8] Dave Dice, Ori Shalev, and Nir Shavit. Transactional locking ii. In *Proceedings of the 20th International Conference on Distributed Computing*, DISC'06, page 194–208, Berlin, Heidelberg, 2006. Springer-Verlag.

[9] Pascal Felber, Christof Fetzer, and Torvald 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.

[10] Aleksandar Dragojevic, Rachid Guerraoui, and Michal 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.

[11] Luke Dalessandro, Michael F. Spear, and Michael L. Scott. Norec: Streamlining stm by abolishing ownership records. *SIGPLAN Not.*, 45(5):67–78, January 2010.

[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.

[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.

[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.

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

[16] Qi Shen, Craig Sharp, William Blewitt, Gary Ushaw, and Graham Morgan. Pr-stm: Priority rule based software transactions for the gpu. In *Euro-Par 2015*, volume 9233, pages 361–372, 08 2015.

[17] Qi Shen, Craig Sharp, Richard Davison, Gary Ushaw, Rajiv Ranjan, Albert Y. Zomaya, and Graham Morgan. A general purpose contention manager for software transactions on the gpu. *J. Parallel Distrib. Comput.*, 139:1–17, 2020.

[18] Idit Keidar and Dmitri Perelman. *Multi-versioning in Transactional Memory*, pages 150–165. Springer International Publishing, Cham, 2015.

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

[20] João Cachopo and António Rito-Silva. Versioned boxes as the basis for memory transactions. *Science of Computer Programming*, 63(2):172–185, 2006.

[21] Tayler H Hetherington, Mike O'Connor, and Tor 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.

[22] Philip A. Bernstein and Nathan Goodman. Multiversion concurrency control—theory and algorithms. *ACM Trans. Database Syst.*, 8(4):465–483, December 1983.

[23] P. di Sanzo, B. Ciciani, F. Quaglia, and P. Romano. A performance model of multi-version concurrency control. In *MASCOTS'08*, 2008.

[24] Yingjun Wu, Joy Arulraj, Jiexi Lin, Ran Xian, and Andrew Pavlo. An empirical evaluation of in-memory multi-version concurrency control. *Proc. VLDB Endow.*, 10(7):781–792, March 2017.

[25] Sérgio Miguel Fernandes and João 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.

[26] Nuno Carvalho, João Cachopo, Luís Rodrigues, and António Rito 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.

[27] Hagit Attiya and Eshcar Hillel. Single-version stms can be multi-version permissive. In *ICDCN*, 2011.

[28] Maurice Herlihy, Victor Luchangco, Mark Moir, and William 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.

[29] Robert Ennals. Software transactional memory should not be obstruction-free. Technical report, Intel Corporation, 2006.

[30] Michael F. Spear, Luke Dalessandro, Virendra J. Marathe, and Michael L. Scott. A comprehensive strategy for contention management in software transactional memory. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '09, page 141–150, New York, NY, USA, 2009. Association for Computing Machinery.

[31] Wilson W. L. Fung, Inderpreet Singh, Andrew Brownsword, and Tor M. Aamodt. Hardware transactional memory for gpu architectures. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, page 296–307, New York, NY, USA, 2011. Association for Computing Machinery.

[32] Wilson W. L. Fung and Tor M. Aamodt. Energy efficient gpu transactional memory via space-time optimizations. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, page 408–420, New York, NY, USA, 2013. Association for Computing Machinery.

[33] 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.

[34] Sui Chen, Lu Peng, and Samuel Irving. Accelerating gpu hardware transactional memory with snapshot isolation. *SIGARCH Comput. Archit. News*, 45(2):282–294, June 2017.

[35] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A critique of ansi sql isolation levels. *SIGMOD Rec.*, 24(2):1–10, May 1995.

[36] Rachid Guerraoui and Michal 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, New York, New York, USA, 2008. ACM Press.

[37] Ang Li, Gert-Jan van den Braak, Henk Corporaal, and Akash 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.

[38] Luke Durant, Olivier Giroux, Mark Harris, and Nick Stam. Inside volta: The world's most advanced data center gpu, Oct 2020.

[39] Kai Wang, Don Fussell, and Calvin 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.

[40] 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.

[41] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. 40(1), 2012.