Accelerating Validation in Software Transactional Memory

Oleksiy Tarlovskyy
Instituto Superior Técnico, Universidade de Lisboa, Portugal
oleksiy.tarlovskyy@tecnico.ulisboa.pt

September 2020

Abstract

The inability to sustain continued technology scaling with detained voltage reduction (end of Moore's and Dennard's laws) severely limits further performance increase of computer architectures. Cutting-edge technologies to overcome these limitations are not expected in the near future, so contemporary architectures rely on heterogeneity to improve performance and energy-efficiency. General-purpose systems that integrate multi-core CPUs and GPU on the same die share physical and virtual memory, and can provide atomic and cache coherent access to data. After research into common transactional memory mechanics (specifically TinySTM's internals) and existing OpenCL features on supported hardware, early experimental simulations demonstrate that an integrated GPU can run a persistent GPU daemon that would efficiently perform value based validation, required by modern transactional memory systems such as NOrec and TinySTM, without interfering with the underlying STM's real-time execution. A partial offloading of such computations yielded an up to 2.1x increase in transactional throughput of TinySTM in a popular STMBench7 benchmark when facing large, long running transactions.
Keywords: software transactional memory, heterogeneous computing, transactional validation, integrated architecture

1. Introduction

Hardware manufacturers are no longer scaling clock speed but are focusing their efforts on increasing the number of cores inside micro-processors. As noted by Herb Sutter in 2005 - in the past, sequential code would automatically become faster with each new hardware generation[1], and today, due to the inability of sustaining Moore's law, single-threaded applications no longer see the benefit from recent hardware, as Central Processing Unit (CPU) clock rates remain steady. Besides increasing the core count to support more concurrent threads, each new processor architecture has been steadily improving its vectorization capacity by increasing the size of Single Instruction, Multiple Data (SIMD) registers, making it possible to apply single instructions on even larger data sets with reduced power consumption [10].

Parallelism has become an integral part of designing performance critical software but is challenging because a programmer has to reason about shared memory and synchronization when accessing it simultaneously from multiple threads. The usual technique for a programmer to build concurrent data structures is to use locking. Using coarse grained locks sacrifices performance, while fine grained locking is error prone and their acquisition order is difficult to orchestrate correctly. Conventional locking techniques come with conventional locking problems, such as priority inversions, convoying, difficulty in avoiding deadlocks and lack of composability. Transactional memory, on the other hand, aids mainstream application development by providing a simple yet efficient use of the available and exponentially growing multi-core hardware by allowing the programmer to declare atomics blocks within code to be executed concurrently.

While conducting research into the latest Transactional Memory (TM) systems and their various incarnations, namely hardware, software, hybrid, as well as the inner workings of cutting edge STM systems, it was observed that their common Value Based Validation (VBV) process executes sequentially during each read and the commit phase, by iterating over a list of transaction private reads and checks that values previously read remain valid - that is, unchanged concurrently by other transactions - in the global main memory. This behavior reflects a single set of instructions affecting multiple data that can be naturally parallelized in order to increase the transactional throughput of a Software Transactional Memory (STM) system without halting its real-time execution - using the parallel nature of a contemporary mainstream Accelerated Processing Unit (APU), housing a CPU and a Graphics Processing Unit (GPU) on a single die.

Studies to improve STM systems mainly focus on the design of conflict detection, version

[1] http://www.gotw.ca/publications/concurrency-ddj.htm

management and conflict resolution [15]. To our knowledge, no other study investigates the possibility of validation *hot-spot* acceleration in TMs using an APU. Additionally, this work documents the research performed into the means of applying *zero cost submission* (section 5) for computations offloaded onto an integrated GPU, and presents results (section 6) from simulations of a persistent GPU daemon (to instantaneously validate a transaction), taking full advantage of the shared memory and coherent caches available on the same chip - integral for non-interference with parallel transactions while running validation.

The practical challenges, with respect to offloading read-set validation to the GPU, came down to locating the lower bound of the validation set size, starting at which the GPU outperforms the CPU, and the validation offloading is justifiable.

Generally, word based STM application access arbitrary memory addresses, while the efficiency of vector instructions in many-core environments is based on an ability to load multiple contiguous array elements into vector-wide registers. Having coalesced memory accesses is crucial for optimal performance and lower energy consumption with vectorization, thus it is necessary to experiment with various GPU memory access patterns.

2. Background

Heterogeneous applications execute on a set of different architectures consisting of host code - executed on general purpose CPUs, and kernel code - executed on parallel devices such as GPUs. The CPU takes responsibility of managing code, data and the environment before loading tasks to the device (iGPU) which runs highly parallel computations. After the kernel is launched, the control flow is returned to the CPU so that it can operate independently while data parallel code runs on the device asynchronously. Heterogeneous programming has vast benefits but increases the complexity of programs because of the differences in instruction set architectures and asymmetries in capabilities between the various processors used in this style of programming.

2.1. *The Intel® graphics processor compute engine*

An Intel® graphics processor provides graphics, media, compute and display capabilities for many of Intel's SoC products [7]. Some applications of Intel graphics for computation include face detection, dynamic crowd simulation algorithms, as well as malware detection that offloads computations to the integrated GPU [12].

Global memory coherency is supported between Intel Gen9 processor graphics and the CPU cores though snooping mechanisms and updated cache protocols ([7], Section 5.7.2). Notably, Intel's GEN architecture has a clearly defined notion of hardware threads, which can be accessed programmatically, and are responsible for running SIMD instructions.

Intel graphics compute capabilities are accessed through the OpenCL [6] portable standard (maintained by the Khronos[2] group) for cross-platform and many-core programming. It generates SIMD code that maps a kernel to multiple *work-items* (within a *work-group*) for simultaneous execution across thousands of threads. To maximize the simultaneous utilization of an Execution Unit (EU), all work-item instances within a thread should be executing the same instruction. Divergent branching work-items are masked off and are executed serially in separate cycles.

A kernel enqueue creates an N-dimensional abstract index space of work-items, called the *NDRange*, consisting of global and local dimensions. The Shared Local Memory (SLM) supports fast data sharing among EU hardware threads inside a single work-group. Inter work-group communication has to be done through the global memory. Additionally, each work-group can enjoy a rich suite of 32-bit atomic read-modify-write memory operations on a slice's L3 cache, global memory and on the SLM.

2.2. Performance Characterization and Simulation of Intel's Integrated GPU Architecture [3]

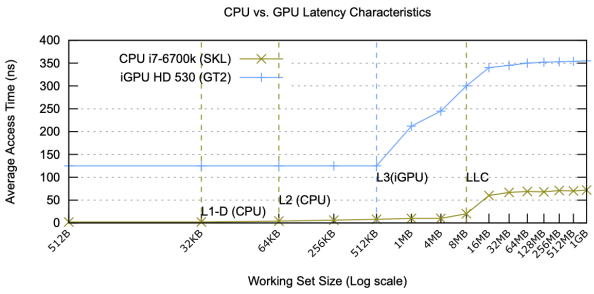

Figure 1: Memory hierarchy access latency Intel CPU (i7-6700k) vs. iGPU (HD 530 GT2). Single-threaded micro benchmark. Data obtained from joining Figures 4 and 5 in work by Gera et al.[3].

Memory Hierarchy Latency

Gera et al.[3] conducted memory hierarchy latency experiments with a single threaded random-access micro-benchmark to determine the latencies for various levels of the Intel iGPU's memory hierarchy. The single-threaded random pointer chasing algorithm they executed on the CPU and iGPU resemble what we are aiming to achieve in our work

[2] https://www.khronos.org/opencl/

with an STM validation system, where each CPU thread is responsible for validating it's own transaction's read-logs whose entries map to random memory addresses in an array of locks (ownership records).

The latency for the first general purpose L3 cache in Intel HD530 is 125ns (Figure 1). Next in hierarchy is the Last Level Cache (LLC) which is shared with the CPU. Here, we can highlight the difference between the latency of accessing the same resource between the CPU and the GPU. Notably, the resource we are after with the iGPU (read-log + lock table) is most likely already resident withing the CPU core's LLC slice (the read-log and locks on which the transaction has been operating). The access time for the LLC from the CPU is ≈10 ns, whereas for the GPU, it starts at 212 ns (first working set size larger than GPU 's L3).

Because the access time steadily increases in the LLC region for the GPU (1MB-8MB) and are pretty flat for the CPU, the authors believe that the GPU is not able to take advantage of the full capacity of the LLC, and perhaps some capacity is always reserved for the CPU. Finally, for 64MB and beyond there is a stable average DRAM access time of about 73 ns for the CPU and 355 ns for the GPU.

The authors point out, that this sort of single threaded pointer chasing workload is unusual for a GPU, as they are designed for high throughput, and not low latency.

2.3. *An overview of transactional memory*

It is simpler to write correct concurrent programs using transactions than it is with locks, by not having to reason about resource acquisition and release orders. Transactions provide a basis to construct parallel abstractions which can be combined, much as procedures and objects provide composable abstractions for sequential code. It requires programmers to simply identify which code blocks should be executed atomically by enclosing a sequence of statements, that access shared resources which appear to happen instantaneously, into atomic transactions.

A transaction is executed speculatively which means tentative changes are made that are only visible to the world when it commits. Whenever there is a failure during a transaction it is aborted and the effects of its changes are not visible - the system stays consistent. For a transaction to get executed successfully, any data read during the transaction must not be modified during its execution by other tasks or threads (tracked by validation). Generally transactions that abort must retry until they are successful.

Software transactional memory, which is what we are focusing upon in our work, is a flexible system that does not have any specific hardware requirements. It relies on a hypothesis that conflicts are unlikely and in most cases, transactions can commit. Besides the cost of instrumenting instructions within blocks of code that are explicitly denoted as atomic, the TM must also make sure to instrument function or method calls from within atomic blocks and make sure they use the TM API.

Validation

A validation technique is used by most STMs (e.g., TinySTM, NOrec) that use invisible reads. Time based validation allows to mark each update with a timestamp taken from a discrete global time that is shared by all transactions. It is a costly operation that ensures the consistency of a transaction through a comparison of logical timestamps of every entry in the read-log.

The concept of transactional validation is one of the largest remaining STM design questions [2] in TM development, and is what this work aims to accelerate.

2.4. *STM on the GPU*

Besides the idea of implementing TM on discrete GPUs [11, 14], there has been recent work done by Villegas et al. [13] in the field of transactional memory and its acceleration on Heterogeneous System Architecture (HSA) compliant hardware, namely the integrated GPU.

The authors created a configurable transactional memory system called APU-TM that can run on the CPU, GPU or split the workload in between. The CPU version is highly inspired by NOrec and has a timestamp based conflict detection mechanism. The STM that runs on the GPU (inspired by GPU-STM [14]) uses a similar global sequence lock to the CPU counterpart, which makes it possible for both sides to shared a lock. Because of this principle transactional conflicts must firstly be detected within the same GPU wavefront (AMD's equivalent of a CUDA's warp) and only then checked against other wavefronts on the GPU and other CPU transactions.

APU-TM replicates an STM on the GPU, and does not alter the sequential validation process, while the work proposal described in this project aims to accelerate it using the highly parallel nature of the integrated GPU.

3. Methodology and Testbed

To evaluate the proposed augmentation to software transactional memory systems we have set up numerous existing benchmarking applications [4, 8, 9] with state-of-the-art STM systems, to execute transactions on a configurable number of threads.

Four of the most performant, and referenced in

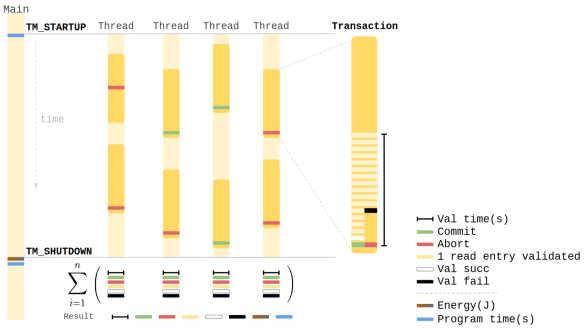

Figure 2: Instrumented counters from n threads. Performance counters are gathered from n STM threads from all transactions for the program's duration and are aggregated at the end of program execution.

academia, word based STM systems were selected to evaluate the solution proposed in this work and instrumented to track relevant performance counters. Namely, they are SwissTM, TinySTM, TL2 and NOrec. Amongst the STM systems we conducted a study on the validation call frequency and the amount of work performed on average in that benchmark configuration. Ultimately, we settled to develop the proof of concept on a single STM (TinySTM) system without compromising the applicability of our proposed solution.

All 12 benchmarking applications were parameterized for their duration and a balance between being more read, write or read-write dominant - which provokes every transaction to spend more or less time validating its read set.

In the initial stages of development of the validation tool we focused mainly on the *reads-validated/s* metric. This metric paints a clearer picture on any (possibly slightest) differences and improvements among the various versions of the validation tool (OpenCL kernel modifications; communication protocol changes; atomic memory orders).

Time spent performing validation is much smaller compared to the total program execution time. This means that slight performance gains would not have been so easily detectable. In later stages of development, the *commits/s* performance metric was considered, as we approached to our most performant kernel and synchronization algorithm.

The current work has very specific hardware and software requirements. Shared Virtual Memory support is enabled in the MS Windows version of the Intel OpenCL driver. However, we found that support under Linux is lacking. We suspect this is due to the existence of many commercial applications in MS Windows such as the Adobe suite, that require state of the art features that drive commercial progress. The only hardware/software configuration that permitted us to work on Intel were the 4th and 6th generation Core™ processors under the Linux 4.7 kernel patched for OpenCL[3]. Luckily, we had the i7-6700k processor available (the more performant of the two commodity CPUs that ships with an integrated GPU). In contrast, on the AMD platform, we used the 2400G APU with Vega 11 graphics.

4. Validation analysis in benchmarks

We are seeking the best possible use cases to prove the utility of our work within transactional memory. In this section, we attempt to look inside the behavior of value based validation.

We started by gathering counters from a multitude of benchmark executions on a set of STM systems with varied degree of parallelism and input parameters. We are looking for STM/Benchmark combinations with the largest transactions. They possess a notably larger read-set, and most importantly, complete validating its majority.

Contention among threads influences the number of reads actually validated before returning because of an invalidation - naturally reducing the utility of our GPU validation tool as the number of executing STM threads increases.

Among all benchmark/STM combinations evaluated, TinySTM executing STMBench7 showed the largest *reads-validated/transaction*. Enabling long structural traversals in STMBench7 lead to larger transactions, regardless of Read, Write or ReadWrite dominated worloads.

5. Proposed transactional APU validation

TM systems have surpassed the boundaries of CPU execution and extended their reach onto the GPU architectures. There are solutions that implement transactions simultaneously on the CPU and GPU [13], however, in these systems, validation remains a single-threaded phenomenon. To the best of our knowledge, there are no systems that fully dedicate the computational capabilities of the integrated GPU to read-set validation in software.

There is a multitude of challenges accompanying the heterogeneity of this type of hardware, with the most notable one being a high latency in inter-device communication [3]. However, recent developments and software/hardware support, as well as some unconventional programming models [5] have interesting characteristics able to tackle these limitations.

OpenCL 2.0 Shared Virtual Memory (SVM) features with *fine-grained system/buffer sharing with atomics*, executed on an Intel APU allows for

[3] https://www.spinics.net/lists/intel-gfx/msg160963.html

pointer-rich data structures, like TinySTM's concurrently accessed global lock table and transactional read-logs, to be shared seamlessly in real-time between the host application's transactions and device validation without data structure marshalling or software translation techniques such as *mapping* and *unmapping* shared buffers on the host (required in the OpenCL 1.x programming models).

Persistent GPU threads The traditional way of doing computations on GPUs is writing C-like *kernels* and having them queued in and executed by the framework's runtime, passing the request through the graphics driver stack, waiting for the thread creation, execution and then mapping and copying the results back from the device memory to the host. However, low latency is critical for lock/Orec access in transactional validation.

The Persistent Threads (PT) GPU programming method has been concisely defined by Gupta et al.[5] to address this shortcoming. According to the authors, PT can achieve an order-of-magnitude speedup over some non-PT kernels by reducing kernel launch latencies.

Using the OpenCL2.0 SVM features with *fine-grained buffer sharing + atomics*, it is possible to exclude steps from the kernel enqueue process each time a parallel computation is needed, and communicate between the CPU and a device without going through the OpenCL driver. Specifically - removing the *clEnqueueNDRangeKernel + clFlush/clFinish* latency. By creating a lightweight communication protocol, new work such as validation requests may be submitted directly to an already running kernel daemon - executing it with minimum delay required for seamless integration into the STM - although possibly requiring some scheduling of the available GPU resources.

With zero cost submission, an STM transaction requiring validation on its private read-set can signal the integrated GPU through a shared variable in a pre-shared SVM buffer for validation to commence immediately on the entries using the awaiting pool of GPU threads.

5.1. Implementation

Basic overview of *instant validation* After the first enqueue of the *Instant Kernel*, the CPU waits until it is submitted into the *cl_queue*, and each hardware thread (OpenCL 2.0 sub-group) responds that it is ready to poll for incoming validation requests. Having collected responses from all hardware threads, the CPU resumes its regular STM operation. In the meantime, every work-group leader in the GPU polls for instructions on the pre-allocated SVM buffer (this is done to reduce atomic traffic on the data-port), while all the other work-items poll on a SLM *round* variable which will be set by the work-group leader once an STM thread submits a validation request and fills the necessary validation metadata (read-set index, write-set address offsets, etc.).

The validation tools created during our research have been modified multiple times and undergone numerous architectural changes over the course of this work. Small changes were introduced, and measured, to interfere as little as possible with the otherwise normal execution of transactions in an unaltered STM environment.

Early adopted approaches offload the entirety of the validation to the GPU, idle the CPU, and return control of the sequence after all work-groups in the *Instant Kernel* complete.

Coalesced validation Our GPU kernel follows the most sought after access pattern in GPGPU programming - the *coalesced* memory access pattern (Figure 3). It is an important optimization technique in high performance kernels that combines multiple memory accesses into a single transaction. Data elements that are spatially close to each other in memory are loaded in chunks to be processed simultaneously by the parallel nature of the GPU.

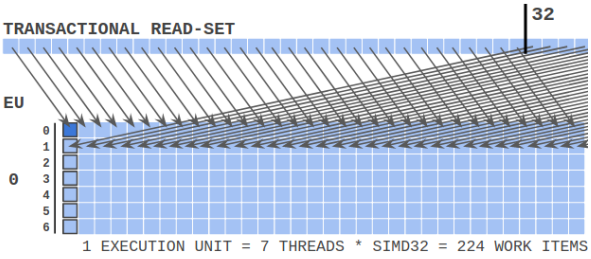

Figure 3: Full GPU validation - *coalesced* read-set access by work-items in a hardware thread (row).

Strided validation Besides the simplest and most intuitive assignment of elements (a coalesced memory access), we experimented with alternative memory access patterns (Figure 4).

Work-items in a hardware thread get executed as memory fetch instructions are complete (pipelined execution). The first work-items (*sub_group_local_id* in OpenCL) of every Hardware threads in an EU gets truly executed in parallel, because memory for those work-items is the first to be fetched.

This memory access pattern is possible because in a persisting *Instant Kernel* execution that uses no more than the device's maximum occupancy, the mapping of *sub_group_local_id*/ *sub_group_id*/ *group_id* is invariant for the duration of the kernel's execution.

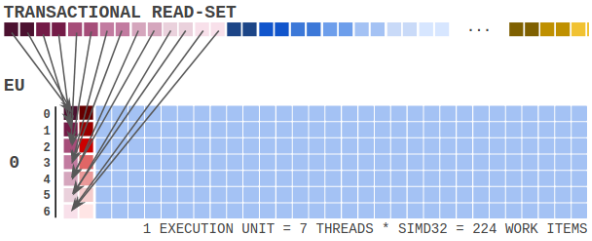

Figure 4: Full GPU validation - *strided* memory access; synchronous work-items in the EU load spatially local read-set entries. In this example - two elements per work-item (K=2).

In Figure 5 we show the *strided* memory access pattern outperform *coalesced* by up to 2.3x (1M entries).

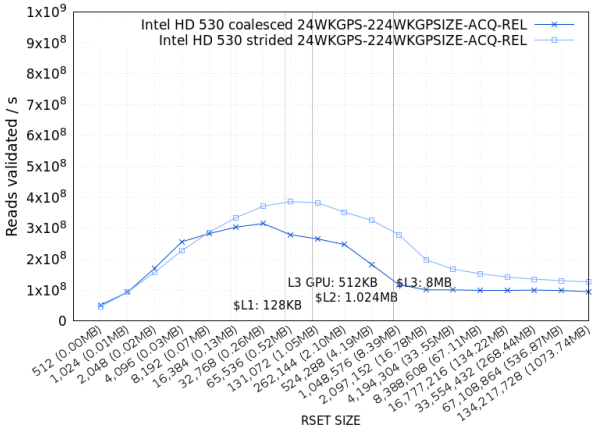

Figure 5: Full GPU validation - *coalesced* vs *strided* memory access (Intel).

Iterative validation in blocks This mode of operation allows the substitution of 5376 loops that calculate offsets, within each kernel instance of the *Instant Kernel*, for a single loop on the CPU thread delegating SVM communication. This new iterative kernel dispatch triggers instant validation *ceil* (*read-set*/*occupancy*) times, where *occupancy* is 5376 work-items on the Intel HD530 testbed machine.

In this mode, each work-item is responsible for a single read-set entry at a time until the entire device re-submerges and moves to the next block. It was configured to operate with both the *coalesced* and *strided* memory access patterns. Block iterations perform best out of all versions that we have conceived.

The overhead of triggering the Instant Kernel *iteration* number of times apparently outweighs the computations of the offsets and the banking con-

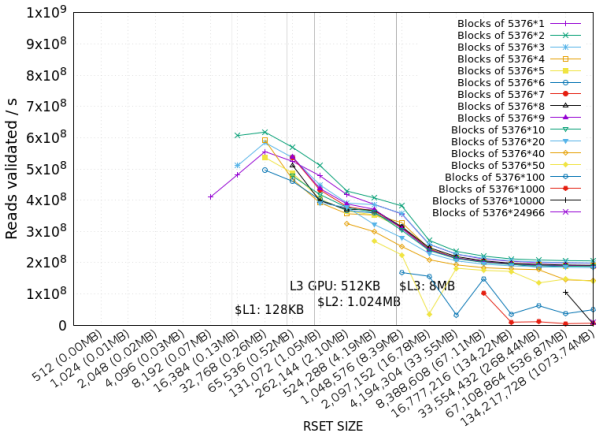

Figure 6: Full GPU validation - *blocks* with variable K (1-24966); $(MAX\text{-}occupancy * K)$ - coalesced memory access (Intel); *reads-validated*/*s*.

flicts, possibly occurring when thousands of work-items iterate individually over the read-set to load multiple read-set entries. Additionally, the synchronization on the stop/invalidated condition no longer happens on global memory within each work-item in the kernel - but once, between iterations, in the CPU delegate thread.

However, we suspect that mapping a kernel work-item to a single read-set entry (K=*1*) may not be sufficient to saturate the GPU with enough memory accesses for an optimal Memory Level Parallelism (MLP). At the same time, having a large K will most likely spill GRF (register) memory into the L3-data cache, and result in worse performance.

We present an ideal K assignment, with our empirical analysis, to be 2 elements in a coalesced memory access in an *Instant Kernel* (Figure 6), for the Intel HD530 (GT2) GPU.

Cooperative validation With a cooperative architecture we tackle two issues: one, we task the otherwise idle CPU with work, as opposed to idling while waiting for the GPU to complete. And two, simultaneously increase the performance of each individual device by making them cooperate.

A variation of work attribution and the dataset size was experimented with to discover the optimal separation of work among the devices. The benchmark to discover this balance was a simple transactional *array walk* (random element accesses), with no computations in between loads and stores.

Data from the benchmark (Figure 7) shows an increase in performance over the baseline TinySTM's *reads-validated*/*s*, beginning at approximately 250k elements, and maintains dominance over the baseline throughout the entire read-set range - until reaching ≈130M elements with an

up to 1.7x increase in performance in the largest read-sets. We observed an ideal partition of work for large read-sets to be close to 50%, while leaning slightly in the CPU's favor.

A **dynamic partitioning** of the read-set between the two devices brought on a more complicated synchronization logic, due to the workload assignment not being predetermined:

The GPU works at the granularity of a work-group. An *Instant Kernel* running on Intel has 24 work-groups which would have to constantly poll the CPU's position, and add extra branching logic (to stop if they surpass the CPU's position). The solution to this problem was for the GPU to process its assignment in iterations of *blocks* of maximum device occupancy. This concept has the device submerge into work by validating a block and re-surfacing to synchronize (Figure 8). The GPU notifies the CPU whether the transactions should be aborted because it found a discrepancy in the read-set (within any work-item), and queries the CPU for the same information.

The comparison between the *strided* and *coalesced* memory access patterns (Figure 5), and a variation of the K value (Figure 6) helped us determine the optimal memory access pattern and work assignment within the GPU's blocks, which was ultimately applied to this dynamic partition scheme.

For a multi-threaded environment, we devised a simple and inexpensive competitive method of dynamically sharing the GPU between STM threads through a *compare-and-swap* resource acquisition. The winning thread would employ the GPU, while the others carry on performing their own validation themselves.

From similar experiments to Figure 7 we observed that, in a dynamic partitioning of work, the GPU

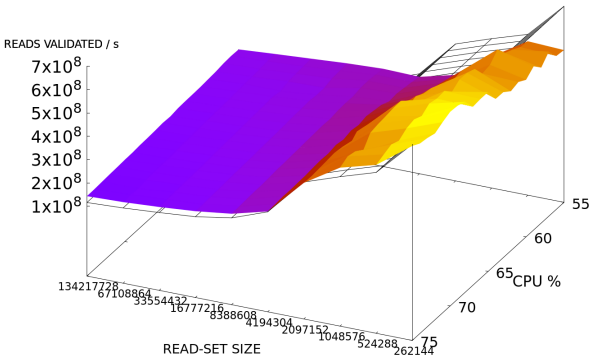

Figure 7: Statically assigned, cooperative CPU-GPU validation. *reads-validated/s* with variable data-set size, intersected with the baseline TinySTM (black plane).

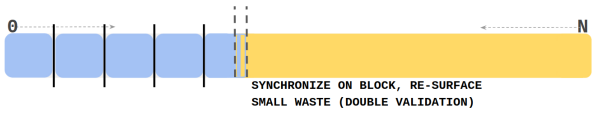

Figure 8: CPU-GPU cooperative validation in blocks on the GPU (blue, left); CPU (yellow,right).

catches-up to the CPU much sooner, in the amount of *reads-validated/s*, at $\approx 250 - 500$k elements. However, the GPU reaches larger increases in raw validation volume (compared to the static partition version) because its kernel has an optimized value for K (number of read-set elements/work-item), and performs the validation in iterations of *Blocks*, whereas the static split had to compute $ceil(read\text{-}set/5376)$ elements per work-item.

Finally, it is noted that the relative cost of synchronization between the devices is a systematic loss of 2k read-set elements (wasted work/double validation).

CPU only - multi-threaded validation We took a side-step and pivoted development into an alternative validation scheme using a pool of unutilized hardware threads to be used when the degree of parallelism is low.

This alternative validation scheme may be adopted in special situations with a purposefully reduced degree of parallelism and a prior knowledge of a highly abort-prone workload. In this case some CPU hardware threads shall be reserved for the unique purpose of serving as validating threads - either as a pool of workers or a statically assigned number of threads.

Validator thread pools are initialized at *TM_INIT*, and sleep awaiting a signal from their employing STM thread. The read-set is partitioned evenly, with irregular parallelism stacked at the last thread, and validation is performed the regular way.

We experiment with 2,4 and 8 *validator* threads per STM thread. Among others, combinations such as $(2, 8)$ and $(4, 8)$ of $(STM\text{-}threads, Validators)$ result in an over-subscription of hardware resources (8 threads), but interesting nonetheless, because STM threads do not always call validation simultaneously.

For the following experiment, as the number of threads increased, the array size they operate within remained constant as a single list of *read-set size*. According to our study of a CPU *validator* thread pool's performance in the amount of reads-validated per second with the micro-benchmark, in Figure 9, the optimal thread count is between 4 and 8 threads. From the figures, the trade-off seems ap-

parent, for the same overall performance, half the threads being employed is a better choice.

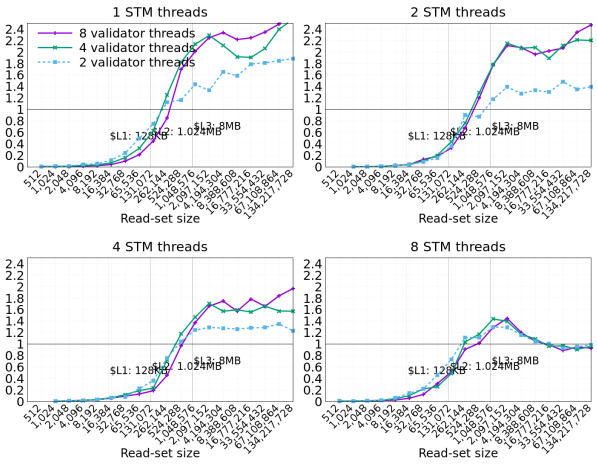

Figure 9: CPU Validator threads reads-validated/s (TX Array micro-benchmark (RND elements).

6. Results

We shall attempt to verify the concept of validation offloading, orchestrated with some existing benchmarks as well as some micro-benchmarks that we have created for specific use cases.

We aim to discover the lowest bound, presenting itself as the smallest read-set size, starting from which it is justifiable to involve the GPU in performing value based validation. The obtained results are presented by means of empirical observations and analysis, under a variety of workload sizes and parallelism degrees.

6.1. Kernel deconstruction analysis

We performed an *Instant Kernel* kernel deconstruction analysis, in a simple transactional array benchmark environment, with systematically subtracted features.

The overhead of every step of an *Instant Kernel* with *Persistent Threads* was measured. The specific set of inspected operations is the following:

- a loop to compute a read-log *chunk*, and an out of bounds check;
- load of read-entries in a *chunk* (has spatial locality - relatively cheap operation);
- following/de-referencing a pointer to some arbitrary address in the lock table (most expensive operation);
- validation branching logic checking whether the lock is taken and retrieving the owner's write set start address and offset (cheap in single-threaded, can get more expensive in multi-threaded).

Unfortunately this experiment is inconclusive with regard to the subsequent branching logic divergence expected to occur in a multi-threaded environment after the aforementioned loads. Nevertheless, it provides an important insight into the overhead introduced by a GPU *Instant Kernel*.

From our analysis, we conclude that the most expensive set of operations is the chasing of the lock pointer. This indicates that the kernel performance is bound only by memory latency of loading potentially arbitrary memory locations.

We expect CPU threads to have a clear advantage, as they access the locks prior to performing validation (i.e., performing writes, retrieving timestamps) with a much smaller memory access latency, and cache them into a much larger LLC.

6.2. Multi-threaded array traversal micro-bench

Until now, we have only focused on analyzing a single-threaded validation scenario. The validation code would always fall into the same branch, because no ownership record is ever locked by a competing thread. It is expected, however, that the presence of contention among STM threads will negatively affect GPU performance, because of kernel thread divergence during lock availability verification. The cost of branching on a GPU, after all, is much higher due to a thread lockstep execution.

The potential for the GPU to have a higher stake in validation may increase when the CPU threads throttle their performance by reaching their hardware limits/thermal ceiling much sooner. However, this case is difficult to reason about, as the GPU is not employed in every validation request of every STM thread. It is a contended for resource, and its utility is less prominent as the degree of parallelism increases.

Figure 10 shows our cooperative validation tool being applied to multiple STM threads in an array micro-bench with disjoint sets. It includes executions of a cooperative validation on both Intel and AMD platforms, and CPU *validator threads* normalized to TinySTM-untouched.

6.3. STMBench7 analysis

Through our custom array traversal micro-benchmark, we conducted an analysis of our best kernel and work-separation algorithm for the CPU and the GPU to cooperatively validate read-sets.

We performed a study on the amount of *reads-validated* in the most popular transactional benchmarks and their various input parameters. Out of that study STMBench7 produced the largest number of *reads-validated* per validation call, thus delivering the best potential utility for our coop-

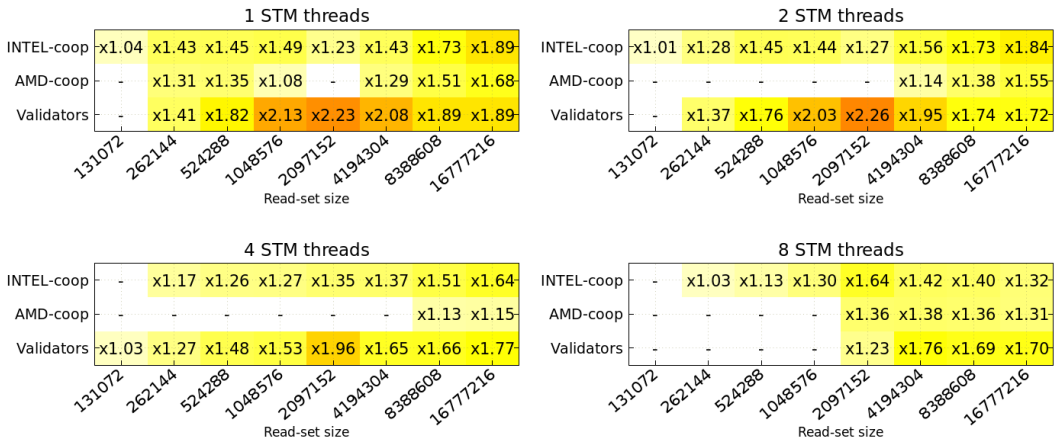

Figure 10: **Disjoint** set array traversal - *reads-validated/s* (speedup). GPU's ability to aid validation scaled with parallelism. Threads operate on isolated data sets (no contention). Speed-up of Intel-coop, AMD-coop, Intel-CPU-validator-threads, in *reads-validated/s* over TinySTM-untouched (baseline).

erative GPU validation tool. We would now like to present our further assessment of the validation tool, with a more relevant transactional throughput metric, using a handful of selected *STMBench7* versions, from a previous analysis, with *long traversals* enabled.

TinySTM-cooperative-validation Finally, we evaluated our work with a high caliber benchmark - *STMBench7*. We expected, however, the GPU to vastly under-perform, as it was shown in Figure 1, where the memory latency of the iGPU is an order of magnitude greater compared to the CPU. However, given a large volume of long transactions, our validation tool outperformed the baseline TinySTM by up to 2.1x in *commits/s*, as shown in Figure 11, where each validation call consisted of ($\approx$ 260k validated elements). Notably, the highest speedup coincides with a full usage of hardware threads on the CPU.

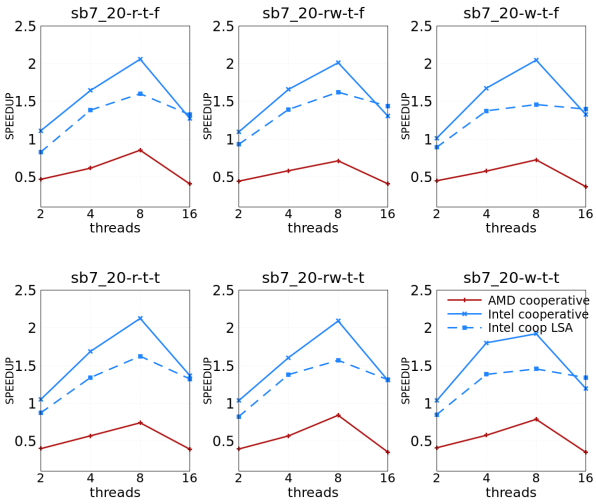

Figure 11: **STMBench7** - 20 second runs - transactional throughput (*speedup*).

7. Conclusions

After having configured and instrumented a multitude of benchmarks to execute with multiple cutting-edge STMs, we performed a broad study of read-set validation volumes throughout a high variety of program executions, under different sets of parameters. Then, after thoroughly studying *zero-copy* DMA methods, the *Persistent Threads* and *Instant Kernel* programming models, as well as various hardware architectures for GPGPU, we developed the novel iGPU STM validation tool.

Given enough execution time to hide the *Instant Kernel* initialization, the system managed to overcome the baseline TinySTM in STMBench7's long running, large transactions. Long running systems are additionally beneficial to our validation tool, as it is able to service more "well timed" validation requests (where GPU is not wasted on short, doomed [to be invalidated early] read-sets).

We provide evidence suggesting that a partial offloading of validation to the iGPU has placement in transactional memory applications. All our results measured in *reads-validated/s* demonstrated some improvement in performance compared to a baseline, unaltered TinySTM (Figure 10).

Our cooperative validation tool achieves best utility, given enough time to operate, by delivering an improvement in transactional throughput of up to 2.1x, with large read-sets ($\approx$ 250k) in STMBench7 (Figure 11).

We have also applied a novel (*strided*) mem-

ory access pattern to an *Instant Kernel/Persistent Threads* programming paradigm, with offset calculations based on the permanent residency of work-groups in hardware threads, which showed good results (Figure 5).

In the future, we would like to perform an in-depth study on the trade-off between consistency and performance for the optimal K value; i.e., a highly contained workload will require faster response from individual work-items/work-groups to terminate validation and free the GPU resource. The higher the K value, the larger the block size and latency. At the same time, under low contention, performance is reduced while performing more frequent and unnecessary synchronization. We have not yet determined the effects of variable K in a highly contended work environment.

Fine grained work-group validation - We experimented with, but ultimately abandoned the sharing of the iGPU on a finer-grained level. Future work should consider a finer grained/non-compete employment of GPU resources, and attempt a simultaneous sharing of the GPU on the work-group level. Thus, having every work-group simultaneously validating a different transaction's read-set.

Workload grouping - There have been recent developments in optimizations for irregular data-parallel workloads, through custom workload scheduling and work-item re-arranging algorithms on the integrated GPU [1]. More abort prone transactions can be grouped to a predetermined subset of work-groups of the kernel to reduce branch divergence.

References

[1] Y. Cho, F. Negele, S. Park, B. Egger, and T. R. Gross. On-the-fly workload partitioning for integrated cpu/gpu architectures. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*, pages 1–13, 2018.

[2] L. Dalessandro, M. F. Spear, and M. L. Scott. Norec: streamlining stm by abolishing ownership records. In *ACM Sigplan Notices*, volume 45, pages 67–78. ACM, 2010.

[3] P. Gera, H. Kim, H. Kim, S. Hong, V. George, and C.-K. C. Luk. Performance characterisation and simulation of intel's integrated gpu architecture. In *2018 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 139–148. IEEE, 2018.

[4] R. Guerraoui, M. Kapalka, and J. Vitek. Stmbench7: a benchmark for software transactional memory. Technical report, 2006.

[5] K. Gupta, J. A. Stuart, and J. D. Owens. A study of persistent threads style gpu programming for gpgpu workloads. In *Innovative Parallel Computing-Foundations & Applications of GPU, Manycore, and Heterogeneous Systems (INPAR 2012)*, pages 1–14. IEEE, 2012.

[6] L. Howes and A. Munshi. *The OpenCL Specification*. Khronos OpenCL Working Group.

[7] S. Junkins. The compute architecture of intel® processor graphics gen9. *Intel whitepaper v1*, 2015.

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

[9] R. Nambiar, N. Wakou, F. Carman, and M. Majdalany. Transaction processing performance council (tpc): state of the council 2010. In *Technology Conference on Performance Evaluation and Benchmarking*, pages 1–9. Springer, 2010.

[10] Quantifi. Vectorization: The rise of parallelism. *Wilmott*, 2018(95):14–17, 2018.

[11] Q. Shen, C. Sharp, W. Blewitt, G. Ushaw, and G. Morgan. Pr-stm: priority rule based software transactions for the gpu. In *European Conference on Parallel Processing*, pages 361–372. Springer, 2015.

[12] R. Velea, Ş. Drăgan, and F. Gurzău. Cpu/gpu hybrid detection for malware signatures for battery-powered devices using opencl. In *Recent Trends in Computer Applications*, pages 139–152. Springer, 2018.

[13] A. Villegas, A. Navarro, R. Asenjo, and O. Plata. Toward a software transactional memory for heterogeneous cpu–gpu processors. *The Journal of Supercomputing*, pages 1–16, 2017.

[14] 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*, page 1. ACM, 2014.

[15] N. Zhou, G. Delaval, B. Robu, E. Rutten, and J.-F. Méhaut. Autonomic parallelism and thread mapping control on software transactional memory. In *2016 IEEE International Conference on Autonomic Computing (ICAC)*, pages 189–198. IEEE, 2016.