Accelerating Validation in Software Transactional Memory

Oleksiy Tarlovskyy

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisor: Prof. Paolo Romano

Examination Committee
Chairperson: Prof. Nuno João Neves Mamede
Supervisor: Prof. Paolo Romano
Member of the Committee: Prof. Aleksandar Ilic

September 2020
To "Vera, Nadezhda, Lyubov".
Acknowledgments

Looking back, I used to daydream about how it would feel to study in Instituto Superior Técnico of Lisbon. Now, I am ecstatic to have made it to the end. Thank you, Mother - you have always helped me in everything. To Professor Paolo Romano, thank you for encouraging, and motivating me numerous times throughout this journey.
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
Resumo

A inabilidade em sustentar o escalonamento contínuo de tecnologia de processamento, em conjunto com a redução da voltagem (fim da lei de Moore e Dennard), severamente limita potenciais ganhos de eficiência de arquiteturas dos computadores. Não se espera que estas limitações sejam superadas, no futuro mais próximo, com os mais recentes desenvolvimentos tecnológicos. Arquiteturas contemporâneas dependem da heterogeneidade de processamento para melhorar o desempenho e a eficiência energética. Sistemas de fins gerais, que integram CPUs e GPUs de vários núcleos no mesmo chip partilham memória física e virtual, e fornecem acesso atómico e cache-coerente aos dados. Após o estudo dos mecanismos mais comuns de memória transacional, e alguns aspetos de OpenCL suportados pelo hardware recente, experiências prévias demonstram que o GPU integrado consegue executar um daemon persistente que correrá validação de memória transacional - um processo obrigatório nos sistemas de memória transacional modernos (NOrec, TinySTM) - sem interferir com a execução em tempo real do sistema. Uma descarga cooperativa deste processamento atinge uma capacidade de melhorar o desempenho de execução em transações grandes e prolongadas até 2.1x mais rapidamente que na versão base do TinySTM.

Palavras-chave: memória transacional em software, computação heterogênea, validação transacional, arquiteturas integradas
3 Proposed transactional APU validation

3.1 The OpenCL 1.x computing model ........................................... 34
3.2 Modern OpenCL 2.0 computing ............................................... 35
  3.2.1 Persistent GPU threads .................................................. 35
  3.2.2 Zero-cost submission validation ........................................ 36
  3.2.3 Hardware characterization .............................................. 36
3.3 Integrating the validation tool into STM .................................. 39
3.4 Implementation roadmap ..................................................... 40
  3.4.1 Overview of basic instant validation kernel instructions ............ 41
  3.4.2 GPU validation correctness ............................................. 43
  3.4.3 GPU only - Instant Kernel - Coalesced memory access ............ 44
  3.4.4 GPU only - Instant Kernel - Strided memory access ............... 45
  3.4.5 GPU only - Instant Kernel - Iterative validation in blocks of MAX occupancy ............... 46
  3.4.6 CPU-GPU - Instant Kernel - cooperative validation with blocks .... 47
  3.4.7 CPU only - multi-threaded validation - CPU validator threads .... 50
3.5 Overview .............................................................................. 50

4 Experimental Evaluation ......................................................... 51
  4.1 Instant Kernel evaluation .................................................... 52
    4.1.1 Memory ordering semantics and work-group optimality .............. 52
    4.1.2 Instant Kernel deconstruction ......................................... 53
  4.2 Transactional array traversal .............................................. 54
    4.2.1 GPU-only validation ................................................... 54
    4.2.2 Optimal number of read-set entries per work-item (K) ............. 57
    4.2.3 CPU-GPU cooperative validation - static work partition ............ 57
    4.2.4 CPU-GPU cooperation validation - dynamic work partition ........ 58
    4.2.5 Single-threaded summary .............................................. 58
  4.3 Multi-threaded evaluation .................................................. 60
    4.3.1 Alternative design - CPU validator threads ......................... 60
    4.3.2 Multi-threaded array traversal ....................................... 61
    4.3.3 Effects of contention ............................................... 63
    4.3.4 LSA favorable workload ............................................. 63
    4.3.5 Aborts .................................................................. 65
    4.3.6 Transactional throughput analysis .................................... 65
    4.3.7 Array micro-benchmark - summary .................................. 67
  4.4 STMBench7 analysis .......................................................... 67

5 Conclusions and Future Work ................................................ 69
  5.1 Main contributions ............................................................ 70
  5.2 Future Research .............................................................. 70
List of Tables

2.1 Quick overview of benchmarks and some of their characteristics (Table 2.3 in [10]) . . . 21
2.2 Concurrent datastructures benchmark . . . . . . . . . . . . . . . . . . . . . . . . . . . 22
2.3 STMBench7 workload variation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 23
2.4 Recommended configurations and data sets for STAMP . . . . . . . . . . . . . . . . . . 24
2.5 TPC-C workload variation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 25
2.6 Characteristics of the Intel and AMD machines used in the study . . . . . . . . . . . . . 27

3.1 Maximum occupancy guide for Intel GEN9 HD530 (GT2). . . . . . . . . . . . . . . . . . 38
3.2 Instant Kernel configurations for AMD and Intel . . . . . . . . . . . . . . . . . . . . . . . . 38

A.1 iGPU comparison (Intel & Amd) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 77
A.2 Counters collected during the analysis of STM execution . . . . . . . . . . . . . . . . . . 77
# List of Figures

2.1 Intel® Gen9 Slice, Subslice, and EU diagram of the GT2 graphics chip configuration ... 8
2.2 SIMD branch divergence ................................................. 8
2.3 Memory hierarchy latency Intel CPU (i7-6700k) vs. iGPU (HD 530 GT2) ................. 9
2.4 Flow/packet classification performance comparison ........................................ 12
2.5 APUNet performance metrics ........................................... 13
2.6 Value based validation of a single read-set entry (comparing versions) ................. 15
2.7 TinySTM metadata: Orecs map to four memory words (by default) .................... 18
2.8 TinySTM `read_entry,t array ........................................ 18
2.9 Modes of operation in the Transactional Array traversal benchmark .................. 21
2.10 Instrumented counters from \( n \) threads ................................ 26
2.11 Validation study of cutting edge STM systems ....................................... 28
2.12 Per thread counters from aggregate execution counters .............................. 29
2.13 TinySTM-STM Bench7, largest reads-validated executions ............................ 30
2.14 Commit procedure in TinySTM - validation invocation ............................... 30
2.15 Maximizing LSA snapshot extension ................................... 32
3.1 OpenCL 1.2 kernel program launch on every STM validation request ................. 34
3.2 Initializing of the iGPU Validation tool in TinySTM on an Intel 6700k CPU; 530HD GPU (GT2) .............................................................. 36
3.3 Basic overview of the Persistent Kernel architecture .................................. 36
3.4 OpenCL work-groups mapping to hardware threads for simultaneous execution .... 37
3.5 Initialization of OpenCL structures and memory pre-allocation ....................... 39
3.6 Summary of implemented versions ........................................... 41
3.7 Kernel instructions - GPU initialization and polling .................................... 42
3.8 Instructions - CPU signalling the GPU for work ..................................... 42
3.9 Kernel instructions - validation logic on every work-item ............................ 43
3.10 Coalesced memory access ............................................... 44
3.11 Coalesced memory - more elements than work-items .................................. 45
3.12 Strided memory - more elements than work-items ................................... 46
3.13 Validating in blocks of maximum occupancy - Intel .................................. 47
3.14 GPU’s contribution ...................................................... 48
Acronyms

**AMD** Advanced Micro Devices. ix, 10

**AoS** Array of Structures. 3

**APU** Accelerated Processing Unit. 2, 3, 32, 69

**CPU** Central Processing Unit. 2, 5–7, 11, 40

**EU** Execution Unit. 8, 9, 37

**GPU** Graphics Processing Unit. ix, 2–8, 11, 29, 32, 33, 35, 36, 48, 69

**HSA** Heterogeneous System Architecture. 2, 20

**HTM** Hardware Transactional Memory. 4, 14

**LLC** Last Level Cache. 7, 9

**MLP** Memory Level Parallelism. 3, 37, 39, 55

**NIDS** Network Intrusion Detection System. 12

**PT** Persistent Threads. 35

**SIMD** Single Instruction, Multiple Data. 1, 8, 37, 69

**SIMT** Single Instruction, Multiple Threads. 6, 19

**SLM** Shared Local Memory. 9, 37, 43, 70

**SM** Streaming Multiprocessor. 6, 7

**SMT** Simultaneous Multi-threading. 8

**SoA** Structure of Arrays. 3

**SSL** Secure Socket Layer. 12

**STM** Software Transactional Memory. 2–5, 11, 13, 16, 28, 32, 33, 35, 39, 40, 60, 63, 67
**SVM** Shared Virtual Memory. 35

**TBV** Time Based Validation. 19

**TM** Transactional Memory. 1, 2, 4, 13, 70

**TX** Transaction. 13, 18, 19, 21

**VBV** Value Based Validation. 1, 19, 35
Chapter 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 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 [35].

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 composable. 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 [30]. In fact, popular programming languages such as Clojure, Haskell, Go, Scala and C++ (proposed as SG5) now have an embedded software transactional memory implementation.

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 the system without halting.

\(^{1}\) http://www.gotw.ca/publications/concurrency-ddj.htm
the real-time execution of the Software Transactional Memory (STM) - using the parallel nature of a contemporary mainstream Accelerated Processing Unit (APU), housing a CPU and a GPU on a single die.

Studies to improve STM systems mainly focus on the design of conflict detection, version management and conflict resolution [48]. To our knowledge, no other study investigates the possibility of validation hot spot acceleration in TMs using an APU. Additionally, this thesis documents the research performed into the means of applying zero-cost submission (subsection 3.2.1) for computations offloaded onto an integrated GPU, and presents results 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.

1.1 Motivation

Heterogeneous computing is showing visible growth through a set standards and specifications, namely the Heterogeneous System Architecture (HSA)\(^2\). Intel\(^\circledR\) has invested heavily into the integrated GPU architecture and expect its customers and developers to see double the performance of the previous generation with Gen11 this year.\(^3\) In heterogeneous systems, the cost of memory transfers is substantially reduced though a shared virtual address space overlaying a physically shared memory - an invaluable requirement of the architecture proposed in this work (detailed architecture in subsection 3.2.1).

Emerging technologies based on the HSA specification ease the burden on programmers by seamlessly offloading computations to the GPU, eliminate explicit and heavyweight synchronization and, when applied in TM, allow to free the CPU from performing validation, hence boost transactional throughput when executing on an APU.

1.2 Objectives

The goal of the proposed thesis is to develop the first GPU validation tool that can be unobtrusively bootstrapped onto STM systems, and accelerate the set of costly operations - using emerging hardware and software technologies, that are focused upon and discussed throughout this thesis, i.e., the validation step - whose call frequency is to be measured in a transactional environment.

The practical challenges, with respect to the final solution, come down to the lower bound of the validation set size, starting at which the GPU outperforms the Central Processing Unit (CPU), and the validation offloading is justifiable. Additionally, due to a variety of configurations controlling transactions’ behavior, it will be necessary to identify an optimal environment in which the final solution best operates.

Normally, there are many concurrent threads executing transactions, but only a single GPU that must

\(^2\)http://www.hsafoundation.com/

\(^3\)https://newsroom.intel.com/press-kits/11th-gen-launch/
be scheduled efficiently to service validation requests. The final version of the validation tool should be able to efficiently schedule and dispatch validation requests from multiple simultaneous STM threads, with as little overhead as possible.

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. At the same time, it may be required to transform data on the host before sending it to the GPU for validation. While having an Array of Structures (AoS) is more natural to code, Structure of Arrays (SoA) better suits memory coalescence because of adjacent GPU vector access.

A desired property of the validation tool is its portability, alongside its accompanying STM system, in the same sense that an OpenCL language program is portable among GPU architectures. An integrated GPU is present in many computing devices, including servers and commodity machines, and a solution that utilizes the entirety of the APU could potentially have a large scope of application.

1.3 Contributions

The above goals were pursued by implementing the APU validation tool, the first STM system to include a compute kernel to perform expensive STM validation cooperatively, by aiding the CPU, and effectively functioning as an additional computational resource which would otherwise remain idle. Our kernel validation has an effectively zero-cost submission, that instantly processes validation requests through a custom cache coherent SVM communication protocol.

A multitude of STM benchmarks were evaluated as to their read-set sizes, and the number of reads validated during their execution with a varied degree of parallelism. Our prototype was then benchmarked on a subset of ideally selected benchmarking software versions, which distinguish themselves with a sufficiently large work-size, necessary to fully saturate a high-performance computing device such as a GPU. We have also developed a simple suite of synthetic benchmarks resembling list traversals with very large transactions.

With our work, we provide evidence to suggest that a partial offloading of validation to a heterogeneous architecture device has placement in transactional memory applications. Our cooperative validation tool shows good results in both raw number of reads-validated/s, and transactional throughput, with an improvement of up to 2.1x (Figure 4.19) with large read-sets in STMBench7 (Figure 4.19), and long running execution times to help hide the setup of the Instant Kernel.

Through theoretical and empirical analysis we discovered and applied the best (strided) memory access pattern to a Persistent Threads programming paradigm, with a novel offset calculation that is tightly coupled to the permanent residency of work-groups in hardware threads, which was only possible with and Instant Kernel programming paradigm (Figure 4.3). Additionally, we discovered an optimal Memory Level Parallelism (MLP) for the GPU resources to parse transactional read-sets with a persistent threads method of programming.
1.4 Outline

The remainder of this dissertation is organized as follows:

- **Background**: chapter 2 begins with a combination of hardware and software (section 2.1) considerations for the proposed solution. It presents GPGPU architectures by NVIDIA, AMD and Intel, discusses the advantages of using the integrated GPU (subsection 2.1.4), and a study of an iGPU’s memory access latency in section 2.1.2. It provides some related use cases of an APU in workload offloading in the following section 2.2.

Next, the chapter follows through to an introduction of transactional memory, focusing more on its various implementations in software, benchmarks generally applied to assess STMs, and applications of a GPU in the field of transactional memory. Some internal TM mechanisms are covered in section 2.3, Hardware Transactional Memory (HTM) in subsection 2.3.1, and practical elements of STM design in subsection 2.3.2, as well as some of its correctness properties. The Chapter then introduces the process of validation (subsection 2.3.2), the Lazy Snapshot Algorithm (discussed in subsection 2.3.3) and how it is implemented in TinySTM (section 2.3.3).

Popular STM benchmarks are later introduced in section 2.4, then our performance analysis methodology is explained in section 2.5, and finally, the Chapter concludes with a study of said benchmarks, as to their read-set sizes and the average number of reads validated when facing contention (section 2.6).

- **Developed architectures**: chapter 3 begins with an introduction to some concepts necessary to fully utilize the architecture of an integrated GPU. It explains how the GPU validation tool can be integrated into an STM, and advances into Section 3.4 for a visually detailed development roadmap of various versions of the kernel validation tool and synchronization algorithms that were experimented with.

- **Evaluation**: chapter 4 provides a comprehensive evaluation of the alternatives to a sequential value-based validation design normally used in STMs. Additionally, it shows results in which employing the GPU as a co-processor accelerates the validation process of a traditional sequential CPU validation.

- **Contributions**: chapter 5 outlines the main contributions of the current work, while providing some possibilities for future development.
Chapter 2

Background

This Chapter is devoted to discussing the state-of-the-art in three main areas, which are closely related to the work proposed in this dissertation. An overview of GPU architectures and their programming models is provided in section 2.1. Then, a general overview of transactional memory in section 2.3 and hardware transactional memory in subsection 2.3.1. Next, this Chapter discusses general design choices for STM systems (subsection 2.3.2) that are important to consider during the development of the GPU validation tool, and follows up on why validation is an important process in certain STM architectures, until finally giving an example of such an algorithm and its implementation in section 2.3.3. Subsection 2.3.4 mentions some recent advances in hybrid transactional memory, and subsection 2.3.5 discusses STM systems that were transformed to run on conventional GPU and APU architectures. Subsection 2.1.2 presents some examples and hints at the performance of an integrated GPU in similar types of work. Then, this Chapter discusses popular benchmarking methods for measurements of TM system performance in section 2.4, and our methodology for ubiquitously instrumenting the benchmarks for this study in section 2.5. Finally, it concludes with a broad study of the behaviour of validation in the aforementioned benchmarking software in section 2.6, revealing the amounts of reads validated by calls to the routine, in a broad spectrum of workloads, on four popular STMs.

2.1 GPU architectures

Heterogeneous applications execute on a set of different architectures consisting of host code - executed on a general purpose CPUs, and kernel code - executed on parallel devices such as GPUs (Figure A.4 and Section 3.4.1 for a detailed OpenCL kernel execution in the context of STM). 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.
This Section presents, and later summarizes (subsection 2.1.4), the alternatives considered for this project - a proprietary NVIDIA CUDA framework for its discrete GPU products, and an open-source OpenCL framework executing on the Integrated Intel Gen9 and AMD Vega graphics.

2.1.1 Nvidia® CUDA

This Section briefly summarizes the fundamental concepts covered in Professional CUDA C Programming [26] written by Cheng et al.

Discrete graphics processing units (GPUs) represent powerful and independent devices (with their own physical memory) that are connected to the CPU over the PCI-Express interface. They are many-core architectures that handle many types of parallelism, namely, multithreading, MIMD, SIMD and instruction level parallelism (page 8, [26]).

Nvidia® designed the Compute Unified Device Architecture (CUDA) framework, which can be used to employ their devices for general-purpose computations (GPGPU). Programs that are to run on NVIDIA GPUs are written in kernels and can be expressed as sequential programs, which are scheduled to run simultaneously on thousands of threads using CUDA. Note that NVIDIA CUDA defines threads as being software threads. GPU cores favor a much simpler control logic than CPU cores, that is why data parallel tasks run best on the GPU, while task parallelism is preferably left for the CPU.

At first, let us consider the hardware and then move on onto the logical memory structure of CUDA discussed in the following paragraphs.

The key part of the Nvidia GPU architecture is the Streaming Multiprocessor (SM) - a highly parallel multiprocessor, which consists of CUDA Cores, load/store unites, special function units (e.g. sine, cosine, square root, and interpolation), warp schedulers, registers, a fast shared memory, a scratchpad memory (L1 which is typically re-purposed across different generations of architectures), and texture units. There is one L1 cache per-SM and one L2 cache shared by all SMs. The GPU’s L1 and L2 cache are examples of non-programmable memory which can cache global and shared memory data. Nvidia has a large assortment of GPUs. Generally, the higher tier - more expensive - graphics cards have a larger number of SMs and can run more simultaneous blocks (described next).

The Single Instruction, Multiple Threads (SIMT) programming model was introduced by Nvidia with a principle of scheduling fixed groups of 32 threads called warps to be executed simultaneously. Unlike SIMD, the SIMT execution model allows each thread in a kernel to have a private local memory, keep its own register state and have an independent execution path. Each thread in a warp issues an address it wants to access and together, the 32 threads, access the device memory which can be coalesced into reading a chunk of memory at a time. Warps are bundled together into logical structured called blocks that are scheduled onto SMs, which in turn can schedule multiple such blocks simultaneously. Conversely, the amount of parallelism deteriorates with warp execution divergence - by having threads within a warp execute a divergent branch of code - serializing the two executing halves of the warp.

Logically, the CUDA framework exposes a variety of memory types to the programmer, namely, registers, shared, local, constant, texture and global memories. Unlike the CPU’s non-programmable
caches, CUDA allows shared and local memories to be controlled from within a kernel programmatically. Each have different scopes, lifetimes, and caching behavior. Kernel threads have their own private local memory and blocks of threads access an on-chip shared memory to reuse data and reduces off-chip traffic. Shared memory is partitioned among thread blocks, and access to it must be synchronized with a barrier. The texture and constant memories are a read-only global memory accessible from all thread blocks, and like any global memory have the same lifetime as the application. With respect to registers, they are a scarce resource that is divided between all active warps on a SM. By using fewer registers in a kernel, it is possible for more thread blocks to reside on an SM, thus improving concurrency and performance. When more registers are used than available in hardware, excess registers spill over to local memory.

Memory model The host and the device each have their own physically separate memory. Traditionally data copying occurs between the CPU memory and the GPU memory through the PCI-Express interconnect, so batching many small transfers into one is considered good practice. A large improvement in the way memory is handled between the host and the device was introduced with the concept of a unified memory. Starting from CUDA 6.0, this feature simplified memory management by creating a pool of memory where each allocation could be accessed with the same virtual memory address on the host and the device, and memory contents would be automatically migrated between the two. It is also possible to have unified memory scattered and used across multiple devices and the host.

On Pascal and later Nvidia® GPU architectures, the CPU and the GPU can simultaneously access memory allocated inside the unified memory pool and support system-wide atomic memory operations. It is still up to the application developer, however, to make sure that there are no race conditions caused by simultaneous accesses.

Nvidia GPUs implement a relaxed consistency model meaning there is no ordering between memory accesses and threads may not see writes in the same order. For example, in a CPU-GPU producer-consumer model, using a CUDA device that supports unified memory and system wide atomics, explicit memory synchronization are required in the form of a system wide (CPU-GPU) kernel thread fence. Frequent system wide synchronizations introduce large bus-wide traffic and are not very efficient in heterogeneous architectures that use the PCI-Express interface for communicating.

2.1.2 The Intel® graphics processor compute engine

An Intel® graphics processor provides graphics, media, compute and display capabilities for many of Intel's SoC products [27]. 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 [41].

The Intel Gen9 graphics processor is a component within the Intel SoC architecture interconnected in a ring-based topology through the on-die bus between the CPU cores and caches. This interconnect facilitates access to the unified DRAM memory controller by each individual CPU core and the graphics processor. Each CPU core and the graphics processor is allocated a slice of the Last Level Cache (LLC)
through an address hashing scheme, however it is still considered as a single cache. Additionally, global memory coherency is supported between Intel Gen9 processor graphics and the CPU cores through snooping mechanisms and updated cache protocols ([27], Section 5.7.2).

The foundational building block of the Intel Gen9 GPU architecture compute engine is the Execution Unit (EU). All Intel Processor Gen9 graphics are composed of slices and subslices which house the EUs. The more powerful graphics are composed of a higher number of slices. Each subslice contains sampler L1 and L2 caches (used for 3D applications), and connects to a shared local memory structure within the GPU’s own L3 complex, shared among subslices. 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.

Figure 2.1: Intel® Gen9 Slice, Subslice, and EU diagram of the GT2 graphics chip configuration
A single slice, and three subslices of 8 execution units each - for a total of 24 EUs. Aside from grouping subslices, the slice integrates additional logic for thread dispatch routing, a banked level-3 cache, a smaller but highly banked shared local memory structure, and fixed function logic for atomics and barriers. ([27]).

An Intel hardware thread is a Simultaneous Multi-threading (SMT) scheduling unit. Each EU is able to simultaneously run 7 hardware threads on a Intel Gen9 GPU architecture, and the global thread dispatcher is responsible for balancing thread distribution across the entire device. Threads are scheduled in a round robin manner within the EU.

Intel graphics compute capabilities are accessed through the OpenCL [24] portable standard (maintained by the Khronos¹ 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 EU, all work-items within a HW-thread should be executing the same instruction. Divergent work-items are masked-off and executed serially in separate cycles (Figure 2.2).

A kernel enqueue creates an N-dimensional abstract index space of work-items, called the NDRange, consisting of global and local dimensions. Similarly to CUDA, OpenCL has multiple logical memory

¹https://www.khronos.org/opencl/
spaces - private memory per work-item (stored in hardware registers that spills to local memory), local memory shared by a group of work-items, and a global memory visible by all work-groups. 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.

Performance Characterization and Simulation of Intel’s Integrated GPU Architecture [14]

![Figure 2.3: Memory hierarchy latency Intel CPU (i7-6700k) vs. iGPU (HD 530 GT2)](image)

Memory hierarchy latency access with a single-threaded micro-benchmark. Data obtained from joining Fig. 4 and 5 in work by Gera et al. [14].

Access times for the entire memory hierarchy are generally higher for the iGPUs compared to the CPUs. The GPUs have a lower operating frequency. For example, our testbed CPUs operate at 3.6GHz (2400g APU) and 4.2GHz(i7-6700k), and their integrated GPUs at 0.95GHz and 1.3GHz respectively. Hence the latency gap.

The latency for the first general purpose L3 cache in Intel HD530 is 125ns (Figure 2.3). Next in hierarchy is the 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 on which 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. It is also noted 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.

2.1.3 The AMD APU

Most recently AMD has launched a series of consumer chips into the market that have an integrated GPU based on the new Vega architecture, similarly to Intel's HD Graphics, but with more performance.

In Vega 10, the graphics core and the other main logic blocks on the chip, including the memory controller, the PCI Express controller, the display engine, and the video acceleration blocks and all linked together (through something AMD coined the Infinity Fabric).

The AMD GCN Vega series processor architecture is shown in Figure A.1. At the heart of the architecture lies the DPP Array which is organized as a set of independent compute unit pipelines that operate in parallel on streams of floating point or integer data. As kernels are running, the hardware fetches instructions from memory into on-chip caches and registers automatically. The memory controller has direct access to all device memory and the host-specified areas of system memory. Just as Intel's architecture, this shared virtual memory design is beneficial to reduce data access latency required in our work. The GCN processor pipeline hides memory latency by keeping track of potentially hundreds of work-items in different stages of execution, and by overlapping compute operations with memory-access operations.

Conceptually, the kernel program is executed independently on every work-item, just as with Intel and NVIDIA, but in reality the GCN processor groups 64 work-items into a wavefront, which executes the kernel on all 64 work-items in one pass.

Within the DPP Array, each CU dispatches wavefronts to be executed on vector ALUs which operate on unique values per work-item or scalar ALUs which operate on a single value per wavefront. Each CU has 64kB of memory acting as a local data storage which allows for work-items within a workgroup to communicate and share data. This memory can be tapped as a software cache for predictable re-use of data, data exchange for work-items in a single workgroup for state control in software or as an efficient way to read off-chip memory required by every work-item in a workgroup exactly once [3].

To achieve peak throughput, GPUs require a tremendous amount of data movement given their massive parallel architecture. Typically, they rely on advanced memory devices arranged and a multi-level cache arrangement. The registers of each Compute Unit pull data from a set of L1 caches, which access a unified, on-chip L2 cache. And it is this L2 cache that then provides high-bandwidth, low-latency access to the GPU's dedicated video RAM. To keep the device running at maximum performance, it is crucial to keep the entire working set of data in the local video memory.

In contrast, the Vega architecture allowing its local video memory to behave like a last-level cache. If the GPU tries to access a piece of data not currently stored in local memory, it can pull just the necessary memory pages across the PCIe bus and store them in the high-bandwidth cache, rather than forcing the
GPU to stall while the entire missing resource is copied over (a typical resource is a texture). Since pages are typically much smaller than entire textures or other resources, they can be copied over much more quickly. Once the transfer is complete, any subsequent accesses of these memory pages will benefit from lower latency since they are now resident in the cache.

### 2.1.4 Comparison Summary

The way a discrete GPU communicates with the CPU memory would introduce an unfavorable latency for frequent transactional memory validation calls from many simultaneous transactions. Masking the DMA delay is challenging because individual validation processing calls in STM typically do not require lots of computation or coalesced memory accesses. PCIe bandwidth is typically an order of magnitude smaller than the memory bandwidth of a GPU, hence data transfer limits performance benefits of GPU accelerated applications.

Compared to NVIDIA's discrete GPUs, Intel and AMD's integrated GPUs can compute some tasks much more efficiently because they share a portion of the processor's L3 cache and accesses the same physical and virtual memory as the CPU. NVIDIA's discrete GPU can use the same virtual memory as the CPU, but only hides the memory transfer latency in much larger data sets, i.e., when computations take up much more time than the initialization of a call.

To seize the performance benefits of using a GPU without the data transfer overhead on the PCIe bus we turn to integrated GPUs for STM validation processing.

### 2.2 The APU in various workloads

**Packet processing** There have been research efforts focused on boosting packet processing performance by offloading it to discrete graphics processing units (GPUs). Integrated GPUs placed on the same physical die with a CPU, however, offer many advantages such as an on-chip interconnect communication, and shared physical - and most recently a virtual memory. Tseng Et al. [40] conduct in-depth profiling and analysis to determine the integrated GPU's capabilities, and performance potential for packet processing workloads.

The authors introduce a GPU accelerated network packet processing framework that fully utilizes integrated GPU's massive parallel processing capability without the need for large numbers of packet batching, which might cause a significant processing delay - the same kind of instantaneous processing/response requirements imposed by STM validation offloading on an integrated GPU.

STM read-set validation share some common characteristics with packet processing:

- Highly parallelizable at the transaction's read-log level.
- Being very memory intensive, usually involving frequent indirect memory accesses for ORec data loading for each read-set entry. Thus, communication latency between the host and device is critical.
- Very little data locality for most packet processing workloads. For example, in TinySTM a single ORec/lock covers four contiguous memory addresses (default), however data that maps to those ORecs is often not contiguous.

The network packet analyzer consists of the following workload types:

<table>
<thead>
<tr>
<th>Workload Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Layer 2 switching</td>
<td>Uses the Cuckoo open addressing hashing scheme [47], which requires, on average, 1.5 hash computations and 1.5 memory accesses per lookup.</td>
</tr>
<tr>
<td>IPv4/IPv6 packet/flow classification</td>
<td>Both addressing schemes evaluated based on hash table look-ups. Different sizes of the flow table were evaluated to understand the impact of table size on performance.</td>
</tr>
<tr>
<td>IPv4 forwarding</td>
<td>The algorithm is essentially an indexing scheme, the computation is negligible and on average it requires up to two memory accesses per lookup. A memory-intensive workload; forwarding table does not fit into CPU cache.</td>
</tr>
<tr>
<td>IPv6 forwarding</td>
<td>Similar to IPv4, but with 4-6 memory accesses per flow/packet due to having a longer address.</td>
</tr>
</tbody>
</table>

In comparison, the author’s test-bed iGPU contained double the core count as our Intel platform, slightly weaker CPU, and a a workload that performs hash computations and hash table look-ups. In contrast, our STM validation workload corresponds to following pointers to arbitrary memory locations to a corresponding read-set ORec/lock. By offloading the packet flow analysis to the accelerator they managed to outperform the CPU pipeline model by up to 2.5x, as seen in Figure 2.4.

**APUNet** Go Et al. [15] find that many network applications with computations offloaded to the iGPU, regardless of being memory or compute-intensive, outperform the CPU baseline as well as optimized implementations. The authors demonstrate that that popular cryptographic algorithms in SSL and IPsec such as RSA, SHA-1/SHA-2, and ChaCha20 highly benefit from parallel computation cycles rather than transparent memory access hiding usually achieved when using a GPU. They outperform optimized CPU code by a factor of 1.3 to 4.8.

APUNet has an improvement on G-Opt’s [28] (packet processing on a dedicated PCIe GPU) performance by up to 1.6x in hashing-based IPv6 forwarding, 2.2x in IPsec gateway (Figure 2.5.a), 2.8x in Secure Socket Layer (SSL) proxy (Figure 2.5.b), and up to 4.0x in Network Intrusion Detection System (NIDS).

In their work, it is noted that APUNet does not improve the performance of simple applications like IPv4 forwarding as the offloaded GPU kernel task is too small to overcome the CPU-GPU communication.
overhead. Kalia et al. [28] have argued that the key performance contributor of GPU is not its high computation power, but its fast hardware context switch that hides memory access 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 (TX) 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. Generally transactions that abort must retry until they are successful. Due to their speculative execution nature, transactions perform better than traditional locking schemes under a low level of contention [23]. Under high contention TMs suffer from a high number of of repetitive aborts [46].

Correctness Besides atomicity, a transaction must enforce isolation by ensuring that another concurrent transaction does not affect its result. When multiple transactions are executed concurrently the result is serializable - it appears that they happen in a one at a time order. In other words the statements and memory accesses inside each individual transaction do not get interleaved with statements from other transactions. Besides their result being serializable, TMs offer an even stronger isolation level, which goes under the name of opacity [17].

The guidelines to building correct STM implementations under the opacity property have been formalized as multiple correctness criterion and these requirements have been intuitively described by many TM designers. In TinySTM (2.3.3), opacity is achieved by accessing memory region versions from
snapshots in isolation and executing transactions speculatively - by means of a Lazy Snapshot Algorithm (subsection 2.3.3).

These correctness criterion extend the classical database definition of the serializability property by (1) ensuring that even non-committed transactions are prevented from accessing inconsistent state, (2) every operation performed by a committed transaction should appear as if it occurred at some single and indivisible point in time, and (3) aborted transactions must not leave any trace of their updates in shared memory [17].

Programming languages like C++ and Java have their fundamental memory model tightly integrated into their core. But when it comes to transactional memory there is no consensus on how it fits in. Unlike databases where all data is treated transactionally, there are times when it is needed to access non transactional data from within transactions. This corresponds to the implementation choice of weak or strong atomicity [16]. The former guarantees that an atomic block only is isolated from other atomic blocks, and the latter - that atomic blocks are isolated from all other memory accesses.

2.3.1 Hardware transactional memory

To reduce cost and complexity of integrating transactional memory in modern CPUs, all existing commercial HTM implementations are based on an extension of a cache coherence protocol already employed by today’s multi-core processors [22]. The protocol works at the level of individual loads and stores and permits all processors to agree on the state of shared memory. Each load and store issues a broadcast to all other processors that snoop in on the bus. The main idea of the protocol is to buffer tentative changes to cache instead of updating main memory directly, which mimics transactional behavior.

HTM has been implemented by various vendors, namely, IBM’s Blue Gene/Q (2011) [43], IBM’s zEC12 (2012) [25] and IBM Power8 (2014) [2]. Intel has also began implementing transactional microcode extensions starting from their Haswell architecture in 2012 and named it TSX (Transactional Synchronization Extensions).

Intel's approach to HTM yielded outstanding performance in workloads with small transactions, such as concurrent data structures and Memcached, but only a couple of STAMP benchmarking programs. Intel's HTM performance is strongly dependent on the access patterns to L1 cache, and long running transactions can lead to frequent cache capacity exceptions and spurious aborts [10].

Maurice Herlihy once stated that when Intel adds hardware features to their CPUs, they stay for good and become part of all future products - meaning the addition of transactional memory as a hardware feature is great for a broader appeal to TM amongst developers.

2.3.2 Software transactional memory

Software transactional memory 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 (Figure A.2).

Generally encountered in STM implementations are a number design choices which trade-off between some form of consistency and performance. Visible reads require taking ownership of a resource by adding their transactions descriptor to a list of readers, which enables writing transactions to detect conflicts with reading transactions, but if multiple transactions read the same object their performance will suffer from contention. In the analysis of contention management policies by Scherer and Scott [38] it was shown that to maintain performance under sufficiently high write contention it is necessary to use invisible reads. Additionally, any STM that has frequent read-write conflicts can alternate between the two schemes. For example using invisible reads regularly and switching to visible reads when a transaction aborts consecutively.

Using invisible reads requires checking that no other update interfered with the atomicity of a transaction's snapshot by keeping a consistent view of its private data using some validation scheme, which either checks the private read set every time a shared object is speculatively read (incremental validation), or like ASTM [31] perform a single validation at the end of each transaction (no guarantees of consistency during execution - only after commit). Using versioned locks [37] allows speculative reads to remain invisible to other transactions while writers still remain visible.

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. SwissTM [12] performs validation using the Lazy Snapshot Algorithm (subsection 2.3.3), and a variation of TinySTM's [13] encounter-time locking, while NOrec [6] uses time based validation and resorts to value based validation if the former fails (a reader sees the global time advance). The concept of transaction validation is one of the largest remaining STM design questions [6] in TM development. It is a costly operation that ensures the consistency of a transactional memory system, and is what this work aims to improve.

A STM system typically acquires locks to the locations its transactions write to at encounter time (“eager acquire” in TinySTM [13] and McRT-STM [37]) or at commit time (“lazy acquire” as in TL2 [9]). Transactional writes and their versions can be written to memory in two ways. The first approach is called eager version management where updates are directly written to shared memory (also called write-through), but must keep an undo-log in case the transactions is aborted. The second is a lazy version management, where updates are buffered locally into a redo-log and are written to main memory on commit (also called write-back).

The commit-time locking scheme is effective for short transactions, but might waste significant work in longer ones that eventually abort due to write/write conflicts. This is because write/write conflicts, which usually lead to transaction aborts, are detected too late.
The *encounter-time locking* scheme, used by most STMs, e.g., TinySTM, McRT-STM [37], and Bartok-STM immediately aborts a transaction that tries to read a memory location locked by another transaction. Hence, read/write conflicts, which can often be handled without aborts, are detected very early and resolved by aborting readers. Long transactions that write memory locations commonly read by other transactions might thus end up blocking many other transactions, and for a long time, thus slowing down the system overall [12].

**TL2** [9] - This STM has an *ownership record* for each memory word - which is just a simple versioned write locks. It also uses a single global time to stamp the memory words upon each write. Every transaction caches locally the value of the global time when it begins. On every read and write the algorithm makes sure that the timestamp of the orec that the memory word belongs to is lesser or equal to the global time cached at the beginning of the transaction, and that the write lock inside the orec is free (the orec corresponding to the memory location has not been written to since the transaction began, and is not being written to at the moment). Alternatively, the transaction aborts (TinySTM [section 2.3.3] gives the transaction a second chance with the LSA algorithm [subsection 2.3.3]). Upon commit, the transaction tries to acquire the orecs of every item in its write-set. Next, it atomically increments the global clock using a *compare-and-swap* operation, then re-validates the read set. The new values of the write-set are then buffered to memory before releasing the orecs while updating their versions to the new value of the global clock.

**SwissTM** [12] - is a lock and word based STM that uses optimistic (commit-time) conflict detection for read-write conflicts and pessimistic (encounter-time) conflict detection for write/write conflicts, as well as a new two-phase contention manager that ensures the progress of long transactions while inducing no overhead on short ones.

**NOrec** [6] - Most recently, STMs such as NOrec, have departed from local metadata (ownership records) to adopt global metadata for conflict detection (using a single global sequence lock), which essentially simplified them but makes them less scalable. This global sequence lock is used to protect the commit protocol - it essentially works as a global mutex for writing. Before committing, a transaction obtains a unique global timestamp that is larger than any existent transaction's start or commit timestamp. Because of this global time, updates can be ordered and compared, and validation has to be performed upon every commit.

NOrec uses value-based validation as fallback when version-based (metadata based) validation fails, i.e., the global sequence lock does not correspond to the one recorded locally at the beginning of a transaction - meaning another writer has recently committed. In this case, value-based validation must be performed to make sure that if the previous reads were to be performed right now, would return the same values as previously seen. This STM implementation is very efficient at a low number of threads, and when there aren't many concurrently committing transactions.

NOrec implements a naïve value-based validation that requires a quadratic number of checks on every previously read value's validity - performed after each read (in the worst case when version-based validation fails after each read).
2.3.3 The Lazy Snapshot Algorithm

Overall, time-based validation is being used by most STMs that use invisible reads and require their transactions maintain consistent snapshots. For example, SwissTM [12] consists of LSA and a variation of TinySTM's [13] encounter-time locking. Likewise, the STM in Intel's TM stack [34, 44] uses LSA with a write-through design. LSA is also an optional mode of operation for transactions in TinySTM.

The algorithm reduces abort rates in transactions by extending their validity range and keeping their snapshot consistent. Having a snapshot to read from and write to allows for an optimistic concurrency control during transactional reads and concurrent but non-conflicting update transactions to commit in parallel. The performance of taking atomic snapshots of data accessed in a transaction is essential for good overall performance of a TM system.

As opposed to traditional snapshot isolation where a snapshot is taken at the start of the transaction, the lazy snapshot algorithm (LSA) computes the validity interval on the fly by looking at the already read local copies and their available versions (available as a local copy or a globally committed version by other transactions). A version of an object is added to the snapshot whenever it is first accessed.

This validation algorithm is based around monotonically increasing integer values that are used to time-stamp commits. This allows access to a versioned history of each object. For example, a version $o^v_i$ is valid within the range $[v_i, v_{i+1} - 1]$, which means that only one transaction updated object $o$ within this range, and created its new version during commit - at timestamp $v_i$.

$V_T = [\min_T, \max_T]$ is the intersection of each transaction's local object's validity interval. It is certain that each object's version in this snapshot was committed no later than $\min_T$, and there is not a transaction that committed another version within $V_T$.

Writes must always access (read and write to) the latest existing version of an object and write to it at commit. The most recent object's validity range must intersect with the transaction's validity range. If it does not, we can opt for an extension of our snapshot. If the validity range does not overlap with our snapshot even after extension, the transaction must abort. In case of a write-write conflict, the contention manager is asked to arbitrate (tries to delay one of the transactions) instead of going straight to abort. This way, only one instantaneous active writer is allowed to exist.

When performing a read on new object that is not yet in the private read-set of the transaction we have to try and select the most recent version whose validity range overlaps our transaction's snapshot $V_T$, and then update the snapshot's range accordingly. It is important to note that this design allows an active reader to read while there is an active writer. The writing transactions defer their read-write conflict until commit time, and reading transactions run unobstructed for a longer time period because of extensions.

TinySTM [13] is a portable and efficient lock based implementation of software transactional memory which is part of the Dresden TM Compiler (DTMC), and is popular in academic research. It is one of many existing STMs that implements the Lazy Snapshot Algorithm.
TinySTM uses ownership records (Orecs, which are stored together in a large global lock table) to synchronize transactions. Before a transaction can access a memory location it must hash its address into an Orec and use it for synchronization. Orecs are kept separate from the memory that can be accessed by a transaction. In TinySTM, an Orec is a word-sized structure whose most significant bit is a lock bit - essentially it is a timestamp carrying lock. When the lock bit is set, the remaining bits store an identifier of the current locking transaction, otherwise a timestamp of the last update remains engraved in an Orec (Figure 2.7). If a transaction cannot obtain the lock (resource owned by another transaction), by default it passively self-aborts (may use a different contention management scheme). False conflicts based on shared locks can be minimized by having each memory address map to a unique lock but requires more memory, so instead Orecs are mapped to strips of memory.

TinySTM offers various contention management strategies (conflict resolution on Orec acquisition), allows to choose when locks are acquired (commit or encounter time), to pick a memory update scheme (write-through or write-back), and to choose the hash function that maps addresses to Orecs. By default it is configured with write-back encounter-time locking, LSA off, and a mapping of 4 addresses to a single Orec. Figure A.2 shows how TinySTM executes transactions in its threads.

**Single version LSA in TinySTM** - TinySTM implements a reduced version of the LSA algorithm that was previously described in subsection 2.3.3, having a single value instead of a snapshot validity range, which is set to the global time when the TX begins. Throughout its execution, the algorithm is only focused on the upper bound of the interval and extends the snapshot after having re-validated the entire read-set. We expect that by enabling the single version LSA algorithm in TinySTM we will observe more time spent in validation as transactional snapshots get extended in certain workloads with large read-sets. More time spent in validation may have a greater impact on the overall transactional throughput introduced by our iGPU validation tool.

The single version Lazy Snapshot Algorithm implemented in TinySTM is disabled by default because of its narrow use case: there is a small overhead with non-contended workloads but it may significantly reduce the abort rate, especially with transactions that read much data.\(^2\)

---

\(^2\)Author’s commentary from the TinySTM read-me file.
2.3.4 Hybrid transactional memory

Hybrid transactional memory is probably the most promising implementation of TM. In theory it can perfectly combine the high efficiency of hardware and the robustness and flexibility of a software transactional system. Transactions are run in hardware and use software as fallback when hardware’s limitations begin to arise. Unfortunately, hybrid systems suffer from high overheads when guaranteeing the correctness of synchronization in between hardware and software transactions.

HyNOrec [7] uses NOrec [6] as its STM fallback and avoids instrumenting reads and writes, however, requires instrumentation of hardware in order increment its global sequence lock used by NOrec, because HTM commits must respect NOrec's single-writer commit protocol and cannot commit during software write-back.

DMP-TM [36] is the first hybrid TM system that relies on a scalable ownership record (Orec) based STM implementation and at the same time avoids the instrumentation cost of using HTM. It detects transactional conflicts between HTM and STM by relying on the operating system level memory protection mechanisms. This hybrid TM maps the heap of the application in the virtual address space twice - to be accessed separately by the HTM and its fallback STM. DMP-TM is the first to use this approach for safe and concurrent execution between HTM and STM. Additionally, DMP-TM has a heuristic scheduling mechanism that tracks the number of capacity aborts (aborts due to cache being full and subsequently cache line invalidations). If their ratio, with regards to total transactions scheduled (aborts+commits), exceeds 90% - the transaction is labeled for STM execution.

2.3.5 GPU Transactional memory

Initial ideas of GPU STMs were influenced by the already existing CPU implementations. Their implementation changes due to differences in the GPU memory model: memory access coalescing, massive multithreading and the SIMT execution model (section 2.1).

Early GPU STMs, such as the one proposed by D. Cederman et al. [5], underutilized the massive multithreading of GPUs by only supporting transactional execution at the thread block granularity (running individual transactions on thread blocks) instead of individual threads (TX per thread).

GPU-STM by Rui Wang et al. [45] was the first software transactional memory system to run at a fine granularity. In STMs like NOrec [6], the idea of value based validation that uses a single global sequence lock runs fast on the CPU because it does not need to access any shared metadata, but does not scale well in GPU systems that have thousands of transactional threads that need to access a single resource simultaneously. GPU-STM introduced three novel ideas for correct implementation of a fine grained STM on a GPU, simplifying data synchronization, allowing scalability to thousands of transactions, and ensuring livelock-freedom:

- A hierarchical validation that uses Time Based Validation (TBV) and VBV as fallback (taken from NOrec [6]). TBV by itself reduces memory comparisons and off-chip memory traffic and uses global version locks that cover memory slices similarly to TinySTM. It is invoked after each read
to ensure a consistent snapshot, and only if the snapshot is out of date that the STM falls back to value based validation (VBV).

- **Encounter time lock-sorting** - the back off strategy used on CPUs to deal with lock contention cannot work at a GPU thread granularity. Threads that belong to the same warp share the same instruction counter and cannot wait for different delays because of their lockstep execution. The authors chose to abdicate from contention management by maintaining a global order for lock acquisition among all transactions.

- **Coalesced read-/write-set organization** merges read and write sets of all transactions within each warp to allow access to consecutive memory locations. Each thread within a warp uses its index to access the merged read-write set.

PR-STM [39] made it possible to remove lock sorting during contention management by assigning a static priority to individual threads inside warps (this effectively pre-sorts them by priority). Static priorities can be assigned by having the lowest thread id have the highest possible priority. Deadlock and livelock inside warps can be prevented by using a lock stealing algorithm where threads with a higher priority can steal locks from lower priority threads. Deadlocks are prevented because any thread can determine their next action when encountering locked data, while the livelock problem is addressed by having threads never enter a state of perpetual lock stealing from one another.

Besides the idea of implementing TM on discrete GPUs, there has been recent work done by Villegas et al. [42] in the field of transactional memory and its acceleration on HSA compliant hardware, namely the integrated GPU. The challenges in their work arise due to the CPU and GPU having essentially different programming models.

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 time-stamp based conflict detection mechanism. The STM that runs on the GPU (inspired by GPU-STM [45]) 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.
2.4 Transactional Memory Benchmarks

Table 2.1: Quick overview of benchmarks and some of their characteristics (Table 2.3 in [10])

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Lines of Code</th>
<th>Atomic Blocks</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TX array traversal</td>
<td>320</td>
<td>1</td>
<td>Our simple transactional array traversal program that uses both strong and weak scaling, where elements can be picked at random or traversed sequentially. Useful for quick prototyping because of its simplicity.</td>
</tr>
<tr>
<td>Data-structures</td>
<td>3702</td>
<td>12</td>
<td>Concurrent Red-Black Tree, Skip-List, Linked-List and Hash-Map with workloads varying contention and update ratio.</td>
</tr>
<tr>
<td>STAMP [32]</td>
<td>28803</td>
<td>35</td>
<td>Suite of 8 heterogeneous benchmarks with a variety of workloads (genomics, graphs, databases).</td>
</tr>
<tr>
<td>STMBench7 [18]</td>
<td>8623</td>
<td>45</td>
<td>Based on OO7 [4] with many heterogeneous transactions over a large and complex graph of objects.</td>
</tr>
<tr>
<td>TPC-C [33]</td>
<td>6690</td>
<td>5</td>
<td>OLTP workload with in-memory storage adapted to use one atomic block encompassing each transaction.</td>
</tr>
</tbody>
</table>

This Section describes benchmarks used to evaluate the efficiency of the solution proposed in this work, and briefly summarizes them in Table 2.1. The primary metrics for benchmarking transactional memory come down to transactional throughput, their abort rate and the energy efficiency, and can be seen listed in Figure A.2.

Transactional Array Traversal We’ve developed this micro-benchmark (to be as simple as possible) to exclude any caveats, possibly inconspicuous bugs or any other influences arising from complex benchmarks that might surface in the early stages of developing our tool. This benchmark is purely load/store with variable TX size and no computations in between. We expect validation time to achieve greater proportions (from total execution time) compared to other benchmarks. This benchmark does a minimum of one write into TX’s own read-set in order to advance the global time and trigger validation of other TXs.

This simple array traversal benchmark pre-fills every entry with pseudo-random long integers and
accepts multiple values as parameters: the desired read-set size, element access pattern (sequential, random) and array shape (disjoint, conjoint) to scale with parallelism. When the array is shaped as a disjoint set, it’s size is a multiple of the number of STM threads (weak scaling, each thread has a private array of read-set size). Alternatively, a conjoint set shape reduces the problem space to the read-set size parameter. Figure 2.9 has an example with four threads.

There is no contention among threads in any disjoint set access pattern, and in the random access pattern (column one), element index randomness is achieved with stdlib’s erand48.

A moderate amount of contention exists in Conjoint arrays+random access mode. There is however, total contention in Conjoint-sequential, because every thread tramples each other by reading and writing the same consecutive elements simultaneously.

In an attempt to increase the time spent validating, relative to total program execution (and most importantly number of elements validated per call to validation), we devised an alternative version of the array traversal program. There is a version with long running transactions where each thread interferes with its neighboring thread once every epoch in order to trigger validation (explained in detail in subsection 2.6.1).

**Concurrent data structures** STM systems’s performance is usually evaluated using operations performed on concurrent data structures, like hashmaps, linkedlists, skiplists and red-black trees (Table 2.2). These operations are normally look-ups, inserts and deletes. The ratio of each operation, along with the size of the data-structures allow to control the degree of contention and the size of the transaction.

**STMBench7 [18]** is a complex synthetic benchmark that adapts a well known OO7 object oriented database benchmark into a TM environment. OO7 was designed to measure the latency of isolated operations issued to an object oriented database management system called OO-DBMS. It operates

---

**Table 2.2: Concurrent datastructures benchmark**

<table>
<thead>
<tr>
<th>Application</th>
<th>Arguments</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>redblacktree</td>
<td>-d 15000000 -i 1048576 -r 1000000 -u 90 -n$th</td>
<td>Programs run for (d) milliseconds with (i) elements inserted before the test, with range (r) of integer values inserted in the set, with a percentage (u) of update transactions, using (n) threads over the same data set size (strong scaling).</td>
</tr>
<tr>
<td>hashmap</td>
<td>-d 50000000 -i 1048576 -r 1000000 -u 90 -n$th</td>
<td></td>
</tr>
<tr>
<td>skiplist</td>
<td>-d 200000 -i 10485 -r 1000000 -u 90 -n$th</td>
<td></td>
</tr>
<tr>
<td>linkedlist</td>
<td>-d 25000000 -i 1048576 -r 1000000 -u 90 -n$th</td>
<td></td>
</tr>
</tbody>
</table>
Table 2.3: STMBench7 workload variation

<table>
<thead>
<tr>
<th>SB7</th>
<th>Arguments Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sb7-r-w-f-f</td>
<td><code>false -s b -d 5000 -w r -t false -m false -n $th</code> false; write root key</td>
</tr>
<tr>
<td>sb7-r-f-f</td>
<td><code>false -s b -d 5000 -w rw -t false -m false -n $th</code> false; write root key</td>
</tr>
<tr>
<td>sb7-w-f-f</td>
<td><code>false -s b -d 5000 -w w -t false -m false -n $th</code> false; write root key</td>
</tr>
<tr>
<td>sb7-r-t-f</td>
<td><code>false -s b -d 5000 -w r -t true -m false -n $th</code> m true/false; enable/disable structural</td>
</tr>
<tr>
<td>sb7-rw-t-f</td>
<td><code>false -s b -d 5000 -w rw -t true -m false -n $th</code> m true/false; enable/disable structural</td>
</tr>
<tr>
<td>sb7-rw-f-t</td>
<td><code>false -s b -d 5000 -w rw -t false -m true -n $th</code> m true/false; enable/disable structural</td>
</tr>
<tr>
<td>sb7-w-t-f</td>
<td><code>false -s b -d 5000 -w w -t true -m false -n $th</code> false; write root key</td>
</tr>
<tr>
<td>sb7-r-f-t</td>
<td><code>false -s b -d 5000 -w r -t true -m true -n $th</code> m true/false; enable/disable structural</td>
</tr>
<tr>
<td>sb7-rw-f-t</td>
<td><code>false -s b -d 5000 -w rw -t true -m true -n $th</code> m true/false; enable/disable structural</td>
</tr>
<tr>
<td>sb7-w-t-t</td>
<td><code>false -s b -d 5000 -w w -t true -m true -n $th</code> m true/false; enable/disable structural</td>
</tr>
<tr>
<td>sb7-rw-t-t</td>
<td><code>false -s b -d 5000 -w rw -t true -m true -n $th</code> m true/false; enable/disable structural</td>
</tr>
<tr>
<td>sb7-w-f-t</td>
<td><code>false -s b -d 5000 -w w -t false -m true -n $th</code> m true/false; enable/disable structural</td>
</tr>
<tr>
<td>sb7-r-t-t</td>
<td><code>false -s b -d 5000 -w r -t true -m true -n $th</code> m true/false; enable/disable structural</td>
</tr>
<tr>
<td>sb7-rw-t-t</td>
<td><code>false -s b -d 5000 -w rw -t true -m true -n $th</code> m true/false; enable/disable structural</td>
</tr>
<tr>
<td>sb7-w-t-t</td>
<td><code>false -s b -d 5000 -w w -t true -m true -n $th</code> m true/false; enable/disable structural</td>
</tr>
</tbody>
</table>

on a variety of graphs composed of data-structures of graphs and indexes mimicking CAD applications. It is parameterizable to control the workload type (which can be read-dominated, read-write or write-dominated), the number of concurrent threads doing the traversals, and the length of graph traversals. STMBench7 does not exploit concurrency patterns in which a thread must wait for results of operations performed by other threads.

The Stanford transactional applications for multi-processing (STAMP) [32] is a suite of benchmarking software whose behaviors resemble a variety of real world applications. It been used to benchmark transactional memory implementations (hardware and software) and was conceived mainly because, at the time, most transactional memory implementations showed results tested with micro-benchmarks that did not represent real world application behavior. STAMP’s benchmarks span various computing domains and vary transaction lengths, read and write set size as well as the amount of contention.

In perspective, the suite consists of dependency graph traversals, string matching, a network packet scanner that uses shared queues and dictionaries, a clustering algorithm, a maze path finding algorithm, graph traversal, tree traversal, and a job processing algorithm. Several of these applications exhibit different behavior depending on the size and type of the input data set (Table 2.4).

There are six variants of kmeans and vacation which target different levels of contention and working set sizes. They denoted by appending -low and -high to the application name to indicate the relative amount of contention. Additionally, the usage of a larger data set is indicated by adding a ‘+’ to the end of the application name. For the remaining benchmarks, there are only three variants, because increasing the data set size also affects the level of contention [32].

TPC-C TPC-C is a well known benchmark that evaluates online transaction processing in databases and is well known in the database community [33]. It represents a wholesale supplier benchmark for relational databases. It consists of 5 different types of transactions, two of which are read-only. Our work uses the TPC-C variant that was ported to operate on in-memory database and, straightforwardly adapted to support TM. The workload variation of TPC-C can be seen in Table 2.5.
Table 2.4: Recommended configurations and data sets for STAMP

<table>
<thead>
<tr>
<th>Application</th>
<th>Arguments</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>genome</td>
<td>-g256 -s16 -n16384 -t$th</td>
<td>Dependencies for ( v ) variables are learned from ( r ) records, which have ( n \times p ) parents per variable on average. Edge insertion has a penalty of ( i ), and up to ( e ) edges are learned per variable.</td>
</tr>
<tr>
<td>genome+</td>
<td>-g512 -s32 -n32768 -t$th</td>
<td></td>
</tr>
<tr>
<td>genome++</td>
<td>-g16384 -s64 -n16777216 -t$th</td>
<td></td>
</tr>
<tr>
<td>intruder</td>
<td>-a10 -i4 -n2048 -s1 -t$th</td>
<td>( n ) traffic flows are analyzed, ( a ) of which have attacks injected. Each flow has a max of ( i ) packets, and the random seed ( s ) is used.</td>
</tr>
<tr>
<td>intruder+</td>
<td>-a10 -i16 -n4096 -s1 -t$th</td>
<td></td>
</tr>
<tr>
<td>intruder++</td>
<td>-a10 -i128 -n262144 -s1 -t$th</td>
<td></td>
</tr>
<tr>
<td>kmeans-high</td>
<td>-m15 -n15 -t0.05 -i random-n2048-d16-c16 -p$th</td>
<td>The number of cluster centers used is varied from ( m ) to ( n ). A convergence threshold of ( t ) is used, and analysis is performed on input ( i ). The input consists of ( n ) points of ( d ) dimensions generated about ( c ) centers.</td>
</tr>
<tr>
<td>kmeans-high+</td>
<td>-m15 -n15 -t0.05 -i random-n16384-d24-c16 -p$th</td>
<td></td>
</tr>
<tr>
<td>kmeans-low</td>
<td>-m40 -n40 -t0.05 -i random-n2048-d16-c16 -p$th</td>
<td></td>
</tr>
<tr>
<td>kmeans-low+</td>
<td>-m40 -n40 -t0.05 -i random-n16384-d24-c16 -p$th</td>
<td></td>
</tr>
<tr>
<td>kmeans-low++</td>
<td>-m40 -n40 -t0.00001 -i random-n65536-d32-c16 -p$th</td>
<td></td>
</tr>
<tr>
<td>labyrinth</td>
<td>-i random-x32-y32-z3-n96 -t$th</td>
<td>The input ( i ) consists of a maze of dimensions ( x ), ( y ), ( z ). ( n ) paths are routed.</td>
</tr>
<tr>
<td>labyrinth+</td>
<td>-i random-x48-y48-z3-n64 -t$th</td>
<td></td>
</tr>
<tr>
<td>labyrinth++</td>
<td>-i random-x512-y512-z7-n512 -t$th</td>
<td></td>
</tr>
<tr>
<td>sscas-high</td>
<td>-s13 -i1.0 -u1.0 -l3 -p3 -t$th</td>
<td>There are ( 2^x ) nodes in the graph.</td>
</tr>
<tr>
<td>sscas-high+</td>
<td>-s14 -i1.0 -u1.0 -l9 -p9 -t$th</td>
<td>The probability of inter-clique edges and unidirectional edges are ( i ) and ( u ), respectively. The max path length is ( l ), and the max number of parallel edges is ( p ).</td>
</tr>
<tr>
<td>sscas-high++</td>
<td>-s20 -i1.0 -u1.0s -l3 -p3 -t$th</td>
<td></td>
</tr>
<tr>
<td>vacation-high</td>
<td>-n4 -q60 -u90 -r16384 -t4096 -c$th</td>
<td>The database has ( r ) records of each reservation item, and clients perform ( t ) sessions. Of these sessions, ( u )% reserve or cancel items and the remainder create or destroy items. Sessions operate on up to ( n ) items and are performed on ( q )% of the total records.</td>
</tr>
<tr>
<td>vacation-high+</td>
<td>-n4 -q60 -u90 -r1048576 -t4096 -c$th</td>
<td></td>
</tr>
<tr>
<td>vacation-high++</td>
<td>-n4 -q60 -u90 -r1048576 -t4194304 -c$th</td>
<td></td>
</tr>
<tr>
<td>vacation-low</td>
<td>-n2 -q90 -u98 -r16384 -t4096 -c$th</td>
<td></td>
</tr>
<tr>
<td>vacation-low+</td>
<td>-n2 -q90 -u98 -r1048576 -t4096 -c$th</td>
<td></td>
</tr>
<tr>
<td>vacation-low++</td>
<td>-n2 -q90 -u98 -r1048576 -t4194304 -c$th</td>
<td></td>
</tr>
<tr>
<td>yada</td>
<td>-a20 -i 633.2 -t$th</td>
<td>The input mesh ( i ) is refined so that it has a minimum angle of ( a ). The input 633.2 consists of 1264 elements; \texttt{timeu10000.2}, 19998 elements; and \texttt{timeu100000.2}, 1999998 elements.</td>
</tr>
<tr>
<td>yada+</td>
<td>-a10 -i timeu10000.2 -t$th</td>
<td></td>
</tr>
<tr>
<td>yada++</td>
<td>-a15 -i timeu100000.2 -t$th</td>
<td></td>
</tr>
</tbody>
</table>

Suffixes of -low and -high indicate the relative amount of contention, and appended ‘+’ symbols indicate larger input sizes.
Table 2.5: TPC-C workload variation

<table>
<thead>
<tr>
<th>TPC-C Arguments</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-t 60 -s 96 -d 1 -o 1 -p 1 -r 1 -n $th</td>
<td>stockLevel transactions ratio - s</td>
</tr>
<tr>
<td>-t 60 -s 1 -d 96 -o 1 -p 1 -r 1 -n $th</td>
<td>delivery transactions ratio - d</td>
</tr>
<tr>
<td>-t 60 -s 1 -d 1 -o 96 -p 1 -r 1 -n $th</td>
<td>order status transactions ratio - o</td>
</tr>
<tr>
<td>-t 60 -s 1 -d 1 -o 1 -p 96 -r 1 -n $th</td>
<td>payment transactions ratio - p</td>
</tr>
<tr>
<td>-t 60 -s 1 -d 1 -o 1 -p 1 -r 96 -n $th</td>
<td>new order TXs ratio - r</td>
</tr>
<tr>
<td>-t 60 -s 20 -d 20 -o 20 -p 20 -r 20 -n $th</td>
<td>number warehouses - w</td>
</tr>
<tr>
<td>-t 60 -s 4 -d 4 -o 4 -p 43 -r 45 -n $th</td>
<td>duration in seconds - t</td>
</tr>
<tr>
<td>-t 60 -s 4 -d 4 -o 4 -p 43 -r 45 -n $th</td>
<td>number of clients - n</td>
</tr>
<tr>
<td>-t 60 -s 4 -d 4 -o 4 -p 43 -r 45 -n $th</td>
<td>maximum number of warehouses - m</td>
</tr>
</tbody>
</table>

Workload scaling in benchmarks

When benchmarking multi-threaded/STM systems, the addition of concurrent threads/transactions influences the overall amount of work to be performed by the environment.

Measuring speedups, where an increasing number of processors are applied to fixed-size problem, is called strong scaling. In contrast, weak scaling measures speedups assuming a fixed problem size per processor. Demonstrating strong scaling requires solving a fixed problem size faster and faster, while demonstrating weak scaling requires solving an increasing problem size within a fixed time budget [29]. Notably, our transactional micro-benchmark includes both levels of scaling.

2.5 Methodology and Testbed

To evaluate the proposed augmentation to software transactional memory systems we have set-up existing benchmarking applications (described in section 2.4) which execute transactions on a configurable number of threads. These applications are set-up with a modest amount of configurable parameters that conduct the behavior of the application.

Four of the most performant, and referenced in academia, word based STM systems were selected to evaluate the solution proposed in this work, and instrumented to hold (and update) relevant performance counters. Namely, they are SwissTM, TinySTM, TL2 and NOrec. Amongst the STM systems we conducted a study on the validation call frequency, which is described in section 2.6, and ultimately settled for a single STM system without compromising the applicability of our proposed solution - TinySTM.

All 12 benchmarking applications (described in detail in Section section 2.4) 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. This produced precisely 66 different programs. Each of the programs was executed at least 20 times, with each of the STM systems bootstrapped with 1, 2, 4, 8, 16 and 32 threads (as well as 1 threaded execution modified to always run validation at commit [note, however, in Figure 2.14, a single threaded STM program never calls validation]). This methodology counted at least 31680 individual program executions of variable lengths.
Figure 2.10: 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. The focused transaction to the right has two possible branches: reads validated until Commit (color-coded green), and reads validated until Abort (color-coded red).

When the number of threads exceeds the number of cores, threads will be pre-empted while executing transactions. This increases the likelihood of abort by increasing the transaction’s duration. All our machines have 8 cores through hyper-threading, thus we omit results above 16 STM threads and focus on the physical limitations of our testbeds.

The benchmarks that we used to verify our work were manually instrumented to make them invoke STM-based synchronization (atomic blocks of code) for reads and writes. The four STM systems that we evaluated for validation frequency were also manually instrumented to gather required performance counters, and depend on the version of the tool we developed (for example, some GPU/CPU specific counters only make sense in a co-operative CPU-GPU validation). The fundamental metrics extracted from every application run are detailed in Table A.2.

Transaction’s counters are collected by their threads and stored in memory, during execution, in their respective transactional descriptors, and are gathered and aggregated at the end of the program run. We store an aggregate of all thread’s counters after termination because transactions mostly run a homogeneous workload in these benchmarks. Each program version is ran at least 20 times to make sure that the standard deviation stays within a reasonable 10%. Figure 2.10 visually represents how counters are collected from transactions running on STM threads.

The reported power consumption measurements are gathered through the Intel RAPL [8] utility. Studies [20, 21] show that the model used by Intel RAPL is accurate at getting the power consumption when compared to a power meter attached to the machine.

In the initial stages of development we focused mainly on the \textit{reads-validated/second} metric. This metric paints a clearer picture on any (possibly slightest) differences and improvements among different 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. \textit{Commits/second} was only considered in later stages of development of the tool, as we approached our most performant kernel and synchronization algorithm.

This work has a very specific set of hardware/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 and drive commercial progress. The only hardware/software configuration that permitted us to work on Intel were the 4th and 6th generation...
Table 2.6: Characteristics of the Intel and AMD machines used in the study

<table>
<thead>
<tr>
<th>Resource</th>
<th>Intel i7-6700k</th>
<th>Ryzen 5 2400g</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor</td>
<td>Intel i7-6700k</td>
<td>Ryzen 5 2400g</td>
</tr>
<tr>
<td>Die Size (mm$^2$)</td>
<td>122</td>
<td>210</td>
</tr>
<tr>
<td>TDP</td>
<td>95W</td>
<td>65W</td>
</tr>
<tr>
<td>Max Clock Frequency (MHz)</td>
<td>4.2GHz</td>
<td>3.75 GHz</td>
</tr>
<tr>
<td>Cores</td>
<td>4 (each with hyperthreading)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Memory</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cacheline size</td>
<td>64B</td>
<td>64B</td>
</tr>
<tr>
<td>L1-D (per-core)</td>
<td>32kB</td>
<td>32kB</td>
</tr>
<tr>
<td>L2 (per-core)</td>
<td>256kB</td>
<td>512kB</td>
</tr>
<tr>
<td>L3 (shared)</td>
<td>8MB</td>
<td>4MB</td>
</tr>
<tr>
<td>RAM</td>
<td>32GB 2400Mhz</td>
<td>16GB 2400Mhz</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Software</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Operating System</td>
<td>Ubuntu 16.04</td>
<td>Ubuntu 18.04</td>
</tr>
<tr>
<td>OS kernel version</td>
<td>4.7 with custom Intel patch(^4) that enables fine-grained SVM + atomics</td>
<td>4.20</td>
</tr>
<tr>
<td>OpenCL 2.0</td>
<td>intel-opencl-r5.0 legacy driver</td>
<td>OpenCL 2.1 AMD-APP (v3098.0) ROCm v 1.9</td>
</tr>
<tr>
<td>Compiler</td>
<td>GCC 5.3.2</td>
<td></td>
</tr>
</tbody>
</table>

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. The changes required to the Linux kernel - which would enable us to use more recent and more powerful processors, such as the Intel i9-9900k - never made it "upstream" into the main kernel branch, and do not have the required functionalities enabled on Linux.

Additionally, all benchmarks that use STM have their threads pinned to physical cores in a round robin fashion. E.g., 4 threads are allocated uniformly, one per core. Hyperthreading is only used when 5 or more threads are used simultaneously.

Notes  It was cumbersome to deal with the Intel OpenCL driver for Linux for a number of reasons. The most prominent problem arose in Instant Kernel executions that took "too long" between the launch/enqueue of the kernel, and any work being dispatched (first validation request).

There is a GPU watchdog service in the Intel i915 driver stack which kills off the kernel when this pattern occurs - leaving the program blocking indefinitely, while unable to communicate with the no longer resident kernel. This issue required some modification to the location of the kernel launch. From the TM API point of view, the kernel setup was moved from TM_STARTUP to TM_THREAD_ENTER, and the first thread to arrive launches the kernel (exactly once), thus displacing it a short time behind its peers. This service has the purpose of preventing long-running, and unresponsive kernels from occupying the GPU and preventing screen rendering. The same programs ran on the AMD platform with no issues.

\(^3\)https://www.spinics.net/lists/intel-gfx/msg160963.html
2.6 Benchmark validation analysis

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, through an evaluation of validation requests, by manually instrumenting well known benchmarks with state-of-the-art STM systems and gathering their performance counters.

![Figure 2.11: Validation study of cutting edge STM systems](image)

Validation study of four STM systems in numerous benchmarks (X axis). Average number of read-set entries validated with varied thread count over at least 20 executions. Thread count is row-stacked in each histogram. The number displayed on top is a geometric average of reads validated across the thread count. Horizontal red line indicates the arithmetic average of reads validated across all benchmarks on that specific STM system.

After having gathered a multitude of counters (list of counters in Table A.2) from benchmarks and TM systems with varied degree of parallelism and input parameters, we begun analysing the results, to attempt and find the lengthiest transactions.

Looking solely at the proportion of time spent in validation can be erroneous, as the transactions may be small, but abort most of the time. This results in a high proportion of execution time spent validating small read-sets. We are looking for STM/Benchmark combinations with the largest transactions. They possess a notably larger read-set, and complete validating its majority without being invalidated too early.
The parameter variations and titles of each benchmark configuration have been listed in section 2.4, in Tables 2.4, 2.2 and 2.5 of the previous Chapter.

This experiment observes the execution of benchmarking programs with a variable degree of parallelism (and in turn contention). Contention influences the number of reads actually checked before returning because of an invalidation - naturally reducing the utility of our proposed GPU validation tool, as the number of executing STM threads increases.

Figure 2.11 shows a good estimate, because it represents the number of reads actually validated (until stopped/invalidated), as opposed to recording what the read-set size was upon entry to validation. The sought after metric is the average read-set size of a transaction just as it exits validation. We are studying the best case scenario to apply our work, by looking for the largest work size to offload to the GPU. There is less use in a transaction with a large read-set if it exits validation too early.

Figure 2.11 row-stacks the total number of reads being validated during the entire execution of the program (aggregate of all thread data as seen in Figure 2.10 of Section 2.5). The Y axis is the accumulated reads-validated during program execution, but it is only facultative, and is displayed only for scale of proportions. The upwards moving gradient visually represents the reduction in the number of reads validated as parallelism increases. A fast reduction hints at a higher contention and abort rate due to invalidation. The two topmost (lightest shades) stacks are a special case of execution with more STM threads than physically available hardware threads. The Intel Skylake machine has 4 physical cores and 8 threads (through hyper-threading).

Because all workloads in these benchmarks are homogeneous, anything that happens to a thread can be reduced to events affecting an individual transaction of that thread, i.e., since we aggregate the total counters by all threads during a program execution we can divide the total over the number of threads used in that execution (the summation of Equation 2.1) to extract the **per thread** performance estimates. The following Figure 2.12 shows a practical example of what has already been discussed in Figure 2.10 of section 2.5, where we visually represented the stages of each counter’s collection.

In Figure 2.11, the numbers on top of each row-stacked bar represent an average (from all thread counts) of **reads validated per validation call**. This is achieved by by dividing how many reads were validated in total over the number of validation function calls, as seen in the following Equation 2.1:

$$
\sum_{th=2}^{16} \left( \frac{\text{Reads Validated}}{\text{Validation success} + \text{Validation fail}} \right) \frac{\text{th}}{4}
$$

(2.1)
From Figure 2.11, TinySTM executing STMBench7 appears to show the largest reads validated per transaction, among any other benchmark/STM. Enabling long structural traversals in STMBench7 lead to larger transactions, namely the following SB7 configurations: `sb7-r-t-f`, `sb7-rw-t-f`, `sb7-w-t-f`, `sb7-r-t-t`, `sb7-rw-t-t` and `sb7-w-t-t` - regardless of Read, Write or ReadWrite dominated workloads.

Following the results observed in Figure 2.11, we focus our research on the STMBench7 benchmark under TinySTM. We will attempt to describe the frequency of and circumstances under which TinySTM calls validation, by closely studying its validation algorithm and datastructures.

![Commit procedure in TinySTM - validation invocation](image)

In Figure 2.14, we present an example of the commit algorithm implemented in TinySTM. Every event that is timestamped with a discrete global time is described immediately to the right of the vertical red line, perpendicular to the global time flow of every transaction (horizontal blue arrows).

All transactions record their Snapshot Time when they initialize at ST=0 (in the example). When two concurrent transactions read memory addresses that map to the same Orec/lock, and attempt to commit simultaneously: they atomically fetch-and-increment the global time and store a private copy. If ST and GT differ, some transaction committed since ST, meaning it might have interfered with "our" reads - must validate own read-set. In case of TX1, validation is skipped because time has not advanced since TX1
started, the transaction commits successfully, and then writes its ST into all of its corresponding oreces and releases them.

In this example, TX2 was the last to complete it's reads. ST will differ from T-1 and validate-rset procedure will be activated. Depending on the result of the validation sub-routine the transaction will be able to commit or will have to abort. It is easy to see why in a single threaded environment validation is never called.

There is however another opportunity for a transaction to call validate-rset. Upon each read, the ownership record's time is compared against the ST of the reading transaction. If some TX wrote to the Orec since ST, the commit will later fail. To prevent this we may extend (move forward) the transaction and its starting timestamp as if it had just started after this Orec mismatch occurred. For this to work seamlessly - validation must succeed. This is the single-value, word based Lazy Snapshot Algorithm.

However, this only works under certain conditions, which may add some additional overhead comparing the benchmarks of the default parameters in TinySTM: a) irrevocability must be enabled; b) locked oreces must be readable; c) epoch garbage collection must be used.

We present and discuss the read-set size's influence on overall time spent in validation during the analysis of the results in section 4.3.4.

### 2.6.1 LSA benchmark requirements

Simple array traversal with disjoint sets (section 2.4) would not benefit from enabling LSA (subsection 2.3.3) - the reads do not overlap with other TX's writes - they are performed into TX's own chunks. The Lazy Snapshot Algorithm attempts an extension through an additional validation only when some other TX writes into another transaction's read-set (more correctly - the Orec that maps to the read in the read-set).

We devised a slightly modified version of the array walk program with disjoint sets (section 2.4). In this version all transactions write once into their immediate neighbour's read-sets (Figure 2.15). This way, more or less, on every round there would be an additional validation call, times the number of active STM threads. The detection of this collision occurs during the read (LSA logic), so that writes must happen very early on in the transaction's lifetime.

An alternative to this approach was having a single, small transaction writing to every other large transaction's read set. This approach wastes a hardware thread dedicated to an interfering transaction, as opposed to performing validation, and we ultimately went with a more heterogeneous micro-benchmark.

The relationship between the read-set size and time spent in validation is an application/workload specific relationship. We can have much clearer results, by constructing a very simple application, that performs mostly reads next to no writes, and has no interference from computations performed by the application in between TM calls. This way we can better reason about where in the system the overall application time is spent, while it is not validating or issuing reads and writes.

The goal of this benchmark is to provide meaningful data to correlate the growth of the read-set,
thread count, and the time spent performing validation (as a proportion of the program execution time).

![Diagram of TX LENGTH with dashed arrows representing transactions performing a single write-after-read into a memory address always read by its immediate neighboring transaction. This scheme kicks-off validation in all transactions every round, for 100 rounds.]

**Figure 2.15: Maximizing LSA snapshot extension**

Dashed arrows represent transactions performing a single *write-after-read* into an memory address *always read* by its immediate neighboring transaction. This scheme kicks-off validation in all transactions every round, for 100 rounds.

### 2.7 Overview

This Chapter presented a lengthy background to some concepts that made this research an opportunity, and sets-up a foundation for a first ever, to the author’s knowledge, tool that synchronizes STM read-log validation between the CPU and the integrated GPU, through GPGPU techniques (such as the *Persistent Threads* and *Instant kernel* programming paradigms) later presented in detail while describing our algorithm in the following Chapter 3.

We have provided an overview of GPU architectures and their programming models (section 2.1), gave a general overview of the design choices for popular STM systems (subsection 2.3.2), that were paramount for a cautious development of the iGPU validation tool, and followed up on *validation* in certain STM architectures, until finally giving an example of such an algorithm and its implementation in TinySTM (section 2.3.3).

The Chapter also introduced some STM systems that were ported to run on conventional GPU and APU architectures (subsection 2.3.5). We also presented similar applications of the APU for a wide variety of computations and their performance characterization on an integrated GPU (subsection 2.2).

Then, we listed and described some popular benchmarking methods, and the input parameters used for measurements of STM system performance in section 2.4, and explained our methodology for ubiquitously instrumenting them for our study in section 2.5.

Finally, the Chapter concluded with a broad study of the behaviour of validation in the aforementioned benchmarking software in section 2.6, revealing the amount of reads validated by calls to the routine, in a broad spectrum of workloads on four popular STMs. The study, into numerous TM benchmarks, highlighted a good starting position to evaluate the quality of the results produced throughout our work.
Chapter 3

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 the GPU [42], 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.

This work proposes the usage of a powerful and underutilized integrated GPU (reasoning behind the choice in a summary of GPU architectures [subsection 2.1.4]), that takes full advantage of the shared memory and coherent caches available on the same chip to completely (or partially) offload the validation step invoked by STM's transactions, thus freeing the CPU and increasing the STM's transactional throughput.

This chapter describes an iterative process of developing various versions of the validation tool, as well as a set of specific hardware and software requirements which are becoming more and more prominent among recent CPUs. We start with a naive approach of enqueueing a compute kernel on every validation request and convey into a more modern OpenCL 2.0 enabled programming model.

Building on the research presented in the previous Chapters, we present a bootstrappable validation tool that interacts with the otherwise idle integrated GPU through the cache coherent domain to aid the CPU in performing expensive read-set validations prominent in popular STMs.

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 [14]. However, recent developments and software/hardware support, as well as some unconventional new programming models [19] have interesting characteristics able to tackle these limitations.
3.1 The OpenCL 1.x computing model

There is an inherent race condition in the design of OpenCL that occurs between setting a kernel argument and using the kernel with `clEnqueueNDRangeKernel`. Another host thread might change the kernel arguments between when a host thread sets the kernel arguments and then enqueues the kernel, causing the wrong kernel arguments to be en-queued. Rather than attempt to share `cl_kernel` objects among multiple host threads, applications are strongly encouraged to make additional `cl_kernel` objects for kernel functions for each host thread [24].

When a memory buffer is accessed on an OpenCL device it has to be explicitly mapped and unmapped after kernel termination. Mapping is an application wrapper around providing memory access exclusivity to that memory region. Modifying the memory on the host and the device simultaneously results in undefined behavior. Preventing other threads from accessing the global lock table (Ownership records) by mapping it into an OpenCL context, would mean halting the entire STM, when a single thread performs lock look-up while validating its read-set on the GPU.

Alternatively, we experimented with mixing OpenCL 1.2/2.0 features. The global ownership records array can be pre-shared using the OpenCL2.0 `clSVMAlloc` allocation construct, enabled with fine-grained atomic support for concurrent access. The memory allocation is tied to a single OpenCL context. Consequently, it would not be possible to enqueue multiple kernels simultaneously on the iGPU because of the race condition, between the setting of the arguments, and their usage in the kernel. A simple, yet inefficient solution would be to serialize access to the iGPU with a lock or a CAS resource acquisition. This ultimately leaves regular STM threads awaiting the un-mapping of the memory region with the locks...
to resume execution. Figure 3.1 describes the naive algorithm that issues a new compute kernel with `clEnqueueNDRangeKernel` on every call to validation, and OpenCL limitations.

### 3.2 Modern OpenCL 2.0 computing

OpenCL2.0 Shared Virtual Memory (SVM) feature in the *fine-grained system/buffer + atomics* [1] mode, executed on an Intel APU allows for pointer-rich data structures, like the TinySTM's concurrently accessed global lock table, to be shared seamlessly in real-time between the host application’s transactions and the 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 [Section 3.1]). Additionally, on the AMD APU, it is not even necessary to use OpenCL constructs *for memory allocation, as SVM fine-grained system memory sharing capabilities are available*. Meaning regular pointers to word-aligned memory allocations can be accessed on the AMD’s integrated GPU. An alternative solution, to the naive approach explained in the previous Section (section 3.1), using the latest OpenCL version can be seen in Figure 3.5 of section 3.3, where we explain basic *Instant Kernel* validation instructions.

#### 3.2.1 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 when finished. It is even possible for the GPU to finish executing and not notify the application immediately. Kernel submissions have a notoriously high latency but work great when the computation time significantly outlasts the submission time. However, low latency is critical for orec/lock access in transactional validation.

The Persistent Threads (PT) GPU programming method has been concisely defined by Gupta et al. [19]. According to the authors, PT can achieve an order-of-magnitude speedup over some non-PT kernels by reducing kernel launch latencies. It is the main tool behind the proposed acceleration of the VBV process, and is described in detail in the following sections.

Using the OpenCL2.0 SVM feature with *fine-grained system/buffer memory + 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.

The architecture requires the first kernel to be en-queued normally (at program initialization), to spawn all hardware threads on the device and wait for dispatches from the host. Thread persistence is
Figure 3.2: Initializing of the iGPU Validation tool in TinySTM on an Intel 6700k CPU; 530HD GPU (GT2)

initialized by spinning at each hardware thread. This scheme effectively represents a GPU daemon and provides a zero-cost submission latency.

### 3.2.2 Zero-cost submission validation

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.

The approach of having all hardware threads spin and wait for a signal is somewhat energy inefficient. However, it is possible to have only a single thread spin-wait on the GPU device using the OpenCL2.0 device-side enqueue feature. The thread's task is to enqueue the device-side kernel that spawns all necessary hardware threads to perform computations. This is faster than enqueuing the kernel from the host and it is possible to perform multiple device-side enqueues that serve distinct purposes or validate different transactions.

### 3.2.3 Hardware characterization

In OpenCL, the mapping of logical work-groups to physical Execution Units and work-items to hardware threads is manufacturer dependant. It is opaque to the programmer using OpenCL. It is not possible,
nor necessary, to schedule a work-group onto a specific execution unit. As long as all execution units are occupied with a maximum of in flight software kernel instances, there will be simultaneous execution of work in the whole device (maximum parallelism), and no serialization of work-groups on an EU (over-subscription of the device with spare work-groups).

Software kernel instances that use SLM to synchronize, or reuse resources in a work-group, are typically mapped (by the driver runtime) to EU hardware threads on the same sub-slice. Thus all kernel instances within a work-group will share access to the same 64 Kbyte SLM partition (subslice in Gen9).

Memory Level Parallelism (MLP) [14] is a key factor for attaining performance on a device such as the GPU. It is related to the memory pressure directed from a higher memory hierarchy. If there is not enough work coming into the GPU by the time it finishes computations, EUs are going to starve for work. We have made a dedicated analysis of this issue, for our specific use case of the GPU (TinySTM data-structures/metadata), in subsection 4.2.2. It comes down to maximizing the number of read-set entries that each work-item validates.

Identifying the best work-group size for the initial Instant Kernel launch required some empirical analysis (subsection 4.1.1). Figure 3.4 shows how work-items of the same work-group can span across different execution units in an Intel Gen9 architecture.

An Instant Kernel requires no more than full device occupancy. As demonstrated in [14] (Fig. 2), it is not recommended to have more work-groups than maximum hardware threads, because the kernel invocations for the remaining work groups are stacked at the end of existing hardware threads.

Currently, GPU specifications do not provide forward progress properties between threads in different work-groups, disallowing popular idioms from multi-core CPU systems to be ported to GPUs in a reliable manner. However, GPU programmers have discovered that by exploiting quirks in today’s GPUs, certain blocking idioms can be made to execute as expected. This can be achieved pragmatically by determining (via trial and error) the number of work-groups for which forward progress appears to be guaranteed, and hard coding this number into the GPU kernel [11]. These numbers are generally program dependant. i.e., for our validation kernel the preferred SIMD size/ work-group multiple is 32 work-items (a hardware thread will execute 32 instances of the OpenCL kernel program simultaneously [while waiting on MLP]).

An important property allows for persistent threads to have forward progress: once a work-group begins executing a kernel (i.e. the work-group becomes occupant on a hardware resource), it will con-
Table 3.1: Maximum occupancy guide for Intel GEN9 HD530 (GT2).

<table>
<thead>
<tr>
<th>Desired metric</th>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>ExecutionUnits</td>
<td>clGetDeviceInfo + CL_DEVICE_MAX_COMPUTE_UNITS</td>
</tr>
<tr>
<td>HWthreads/EU</td>
<td>Device spec sheet</td>
</tr>
<tr>
<td>HardwareThreads</td>
<td>24 * (ExecutionUnits) * 7 (HWthreads/EU) = 168</td>
</tr>
<tr>
<td>SIMDsize</td>
<td>(Compiler heuristic) clGetKernelWorkGroupInfo + CL_KERNEL_PREFERRED_WORK_GROUP_SIZE_MULTIPLE = 32</td>
</tr>
<tr>
<td>Global Work Size</td>
<td>SIMDsize * HardwareThreads = 32 * 168 = 5376</td>
</tr>
<tr>
<td>HWThreadsPerWKGP</td>
<td>7 (empirically discovered).</td>
</tr>
<tr>
<td>Local Work Size</td>
<td>SIMDsize * HWThreadsPerWKGP</td>
</tr>
<tr>
<td>N Work-groups</td>
<td>Global Work Size / Local Work Size</td>
</tr>
</tbody>
</table>

Make sure LWS is a multiple of SIMDsize
Make sure GWS is multiple of LWS

Table 3.2: Instant Kernel configurations for AMD and Intel

<table>
<thead>
<tr>
<th></th>
<th>AMD</th>
<th>Intel</th>
</tr>
</thead>
<tbody>
<tr>
<td>Global Size</td>
<td>11264</td>
<td>5376</td>
</tr>
<tr>
<td>Work size</td>
<td>256</td>
<td>224</td>
</tr>
<tr>
<td>Local size</td>
<td>64</td>
<td>112</td>
</tr>
<tr>
<td>work-group size</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>N of work-groups</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>44</td>
<td>24</td>
</tr>
<tr>
<td></td>
<td>176</td>
<td>48</td>
</tr>
</tbody>
</table>

continue executing until it terminates [11]. This indicates that in a persistent kernel, work-groups shall not hop around EUs during the kernel execution, and will maintain their assigned GRF memory space and instruction registers (ARF).

There is an abundance of heterogeneous systems available on the market. We chose commodity devices from two competing manufacturers. In order to make full use of this types of device, we needed to determine the best combination of global and work-group sizes. Intel’s documentation provided clearer indications on how to achieve maximum occupancy of the device. Their architecture has a transparent definition of a hardware thread. AMD’s, however - does not. Due to certain uncertainties about the AMD architecture, we attempted a couple of combinations of work-group size and global number of work-items for maximum occupancy of the device (Table 3.2). The combinations we experimented with are dependant on the hardware configuration (number of execution units, vALUs/EU, Processing elements/vALU on AMD and number of EUs per slice, hardware threads/EU on Intel).

Some tests were performed using a transactional array walk micro-benchmark to record the best performance among Instant Kernel configurations on both AMD and Intel testbeds. The configurations were obtained programmatically using the steps discriminated in Table 3.1, and are listed in Table 3.2. The results are later reported in chapter 4, Figures 4.3 and 4.4.
Memory Level Parallelism  One of our goals is to understand the number of *in-flight* memory requests that can be serviced concurrently by the GPU. Latency for memory requests can be hidden by multiple outstanding requests, also known as the MLP. If the system has an MLP of $K$, majority of the memory service time for up to $K$ in-flight requests can be hidden. However, if the memory pressure increases beyond the available MLP, we would see the execution time increase. To measure this empirically, we use kernels with different number of read-entries per work-item, and increase $K$ one at a time, while always having work assigned to each work-item.

In subsection 4.2.2 we present some experiments to characterise the effect of memory level parallelism (MLP) for our validation kernel that uses the *Persistent threads* and *Instant Kernel* programming models.

### 3.3 Integrating the validation tool into STM

Changes to the underlying STM, i.e., TinySTM, are a consequence of having to share its datastructures with the GPU through special OpenCL wrappers, for an allocation of SVM memory, used for simultaneous and atomic access between the host and the device.

The simplified chain of dependencies for OpenCL abstractions, method calls and TinySTM datastructures can be viewed in Figure 3.5. Once compiled, this setup is part of the TM library that is used by the application using the STM.

---

**Figure 3.5: Initialization of OpenCL structures and memory pre-allocation**
For faster initialization, we directly jump to the OpenCL platform and device (iGPU) without the platform/device discovery stage (CPU1,2). We pre-compile the OCL kernel into a binary format, to further reduce set-up time, and build the program from it at CPU4 (Figure 3.5).

The global lock table (ownership records) has been moved away from its default place on the stack, and dynamically allocated for OpenCL fine-grained atomic memory sharing (line CPU6). The global lock table, once a statically allocated array of \(2^{20}\) elements, has been increased to \(2^{26}\), because of insufficient bits to address every memory word while experimenting with very large read-sets (in the extremely large cases of 130M entries with our transactional array micro-benchmark [section 2.4]).

Lines CPU9-CPU12 allocate substantial chunks of memory with a known upper-bound to fit the largest read-set possibly occurring during the application’s execution. In TinySTM, each thread’s transactional descriptor commences with a read-set capacity of 4096 (8192 byte) elements, and doubles this capacity as more reads arrive. With OpenCL it would not be possible to re-set the kernel argument, to accommodate a larger read-set, without tearing down and restarting the kernel. We presumptively begin the program with sufficiently large pre-allocated memory for read entries in every STM thread. This simple static assignment consumes more memory but does not require additional performance degrading kernel re-enqueue/re-allocation logic. Upon STM thread initialization, every thread atomically selects their permanent read-set chunk out of a pre-created pool with a simple compare-and-swap operation. We also statically specify how many threads are going to be executed before compiling the STM.

CPU7 and CPU8 are SVM structures for the communication protocol (Figure 3.3) between the CPU and the GPU. threadComm incorporates fields used in the sequential validation, such as: TX’s write-set address bounds in memory, work-item work assignment count, a number of elements to be validated, the read-set id, number of blocks for the cooperative validation, a validity flag and performance counters used on the GPU. The data-structure is padded with at least (each CPU’s) CACHE_LINE_SIZE bytes.

### 3.4 Implementation road-map

The validation tools created during our research and development 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. Figure 3.6 presents an ordered chronology of the types of validation schemes we have developed.

We began by fully offloading validation to the GPU in (1), and contemplated on the complexity of beginning new transactions on the CPU in the meantime (until the GPU returns). Ultimately, we went with a simpler approach on the CPU: (2) simultaneously co-validate a chunk of the read-set (initially, with a static partition or work).

The static partition became dynamic in (3), but the design brought on a more complicated synchronization logic, due to the chunks no longer being predetermined. The CPU inverted the direction in which it validates, and the GPU traversed the read-set in blocks in order to reduce synchronization time (Figure 3.16) between the kernel work-items and the CPU.
At this point, 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 (4). We applied custom pinning of threads to cores, considering the data locality of offsetted chunks of the read-set, and the fact that hyper-threads have shared L1 and L2 data caches.

The comparison between (5) and (6) helped us determine the optimal memory access pattern within the GPU, which was ultimately applied to the dynamic partition scheme in (8).

Figure 3.6: Summary of implemented versions

1) Full GPU validation; 2) Actuation of both devices (CPU+GPU); 3) Dynamic CPU+GPU (multiple STM threads); 4) Side-step into validator threads; 5-6) Alternative memory access patterns; work-item affiliation with multiple read-entries; 7) Full GPU validation - strided memory access; 8) Multi-threaded CPU+GPU with strided memory.

3.4.1 Overview of basic instant validation kernel instructions

In this Subsection we present a run-down of a simplified version of the program used to make full GPU validation with a coalesced memory access.

After the kernel enqueue in Figure 3.5 (line CPU17), the CPU blocks until all GPU hardware threads (OCL sub_groups) notify their readiness in the global SVM commBuffer (line GPU3 in Figure 3.7).

By utilizing Shared Local Memory (GPU4,5), a substantial reduction in the global memory traffic is attributed to every work-item having “cached” communication variables in the SLM (denoted with the _local qualifier). The state is polled by a single element in the work-group (GPU9), and a (fast) local memory barrier is issued to synchronize (GPU14). This is a more efficient alternative to having each work-item poll variables in GPU10-13.

Following a validation request from an STM thread, a threadComm buffer is filled with metadata necessary to perform validation (Figure 3.8), the global phase counter is advanced and the CPU blocks while awaiting all 24 (best Intel HD530 configuration) work-groups to complete (CPU8A).
Figure 3.7: Kernel instructions - GPU initialization and polling

GPU initialization: the CPU has initialized the pre-compiled kernel from a binary file, and awaits every thread on the GPU device to respond and enter polling mode (Action 2 from Figure 3.3)

CPU 1A: int stmt_validate(stmt tx_t *tx){
CPU 2A: int valid = 1;
CPU 3A: threadComm[tx.id].n_per_wi = (tx->r_set.nb_entries + global_dim[0] - 1) / global_dim[0];
CPU 4A: threadComm[tx.id].nentries = tx->r_set.nb_entries;
CPU 5A: threadComm[tx.id].w_base = tx->r_set.w_set_base; threadComm[tx.id].w_set_end = tx->w_set.w_set_end;
CPU 6A: threadComm[tx.id].valid = 1;
CPU 7A: atomic_store_explicit(&CommBuff[PHASE], ++cl_global_phase, memory_order_seq_cst);
CPU 8A: while(CommBuff[COMPLETE] < n_workgroups);

/* Synchronizes with GPU. Wait for all workgroups (line GPU24). */

CPU 9A: CommBuff[COMPLETE] = 0;
CPU 10A: return threadComm[tx.idx].valid;

Figure 3.8: Instructions - CPU signalling the GPU for work

CPU-GPU delegate thread kicks-off validation on behalf of the thread executing a transaction by incrementing the round counter.

Finally, Figure 3.9 shows the instructions that are executed by every work-item in the kernel. In a persistent threads kernel with attained maximum occupancy, each work-item in a work-group will equally branch into GPU16. When there is work to be done by the whole group - there is no thread divergence.

Next, the read-set offsets are calculated by every work-item (GPU17). The total number of elements per work-item (henceforth denoted as K), has been calculated once by the CPU, based on the size of the read-set of the calling transaction and the maximum device occupancy (CPU3A, Figure 3.8). Before advancing into expensive pointer dereferencing (loading the lock), every work-item checks whether the transaction has been invalidated by some other work-item in the kernel. (GPU18) is a potential location for thread divergence, but as our kernel deconstruction analysis shows (Figure 4.2), it is cheaper than...
wasting the GPU by loading locks when we should have relinquished validation and aborted the TX.

At this point we have reached instructions resembling sequential validation on the CPU, stripped of a multitude of macro substitutions. In the termination stage (GPU31), the work-items advance their _privately held phase counters, and the notification of completed work is done by the work-group leader.

![Figure 3.9: Kernel instructions - validation logic on every work-item](image)

Figure A.4 provides a visual cue to understanding work-item/work-group/hardware-thread relationships in this persistent threads kernel.

### 3.4.2 GPU validation correctness

To verify that OpenCL 2.0 devices are observing transactional memory and it's ownership records correctly (Figure 3.5 GPU19, 20, 24), we've devised a basic smoke-test - graphically represented in Figure A.3. After the GPU has read and copied over the read-set and lock values into the three special SVM debug arrays, we compare these values with the ones read by the CPU.

**Performance counters**  In a many core, massively multi-threaded GPU environment with scarce, yet very fast memory, the presence and heavy usage of performance counters significantly degrades performance when updating statistical data numerous times from every thread. For more reliable estimates (due to branching and TX invalidation), we update reads validated count within every work-item - at the (cheap) cost of an atomic SLM access traffic. Instead of accessing a single global counter, work-items of a single work-group update counters in the SLM, which are aggregated at the end of the validation round into global SVM memory (instructions not shown in on any previous figures).
3.4.3 GPU only - Instant Kernel - Coalesced memory access

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.

Using the basic kernel presented in the previous Section, we follow the most sought after access pattern in GPGPU programming - the coalesced memory access pattern. It is a well studied phenomenon in GPU architectures, and an important optimization technique in high performance applications. It combines multiple memory accesses into a single transaction. In it, 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 core.

The baseline STM is configured to map 4 consecutive memory addresses into a single ownership record. This means four consecutive reads will map to the same lock. Lock caching effects in the GPU’s L3-data cache may be unpredictable - if reads were preformed consecutively then some lock loads would result in a cache hit on the following read-entry validity check - thus resulting in better performance. The unchanged, default CPU version of the STM has an advantage of having some locks already cached from having previously worked on them (stm_read/stm_write operations).

We suspect that mapping a work-item to a single read-set entry \((K=1)\) may not be sufficient to saturate the GPU with memory accesses for an optimal MLP. At the same time, having a large \(K\) will most likely spill GRF (register) memory into the L3-data cache. Figure 3.10 shows the amount of GRF memory used per work-item with \(K=1\), and Figure 3.11 with \(K=2\). The figures show the register space occupied with loaded locks and read-set entries, and ignores persistent kernel state variables (a fixed overhead). We discuss the ideal \(K\) assignment with an empirical analysis in subsection 4.2.2 (data in Tables A.6 and A.7).
3.4.4 GPU only - Instant Kernel - Strided memory access

Besides the simplest and most intuitive assignment of elements (coalesced memory access in subsection 3.4.3), we experimented with alternative memory access patterns. The reasoning behind this strided access comes from the memory level parallelism that the GPUs depend on, which in turn influences the scheduling of work-items on the hardware threads of an EU as new memory addresses get fetched (subsection 2.1.2).

In an Instant Kernel, where exactly full device occupancy is achieved, hardware threads across all Execution Units schedule their first work-item simultaneously (vertical gradient in Figure 3.12). Work-items get executed as memory fetch instructions are complete (pipelined execution). The first column in Figure 3.12 gets truly executed in parallel, because memory for those work-items is the first to arrive (empirical analysis in Figure 4.3). It is imperative that the memory these (vertical) work-items are going to be requesting gets loaded into the L3-data cache in single transaction/ together.

This is truly a coalesced memory access, from the perspective of work-items scheduled to be executed at the same time. We termed this memory access as strided, because of way offsets are calculated (Figure 3.12 (element j calculation).

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

For this example, consider the mapping of a tuple \((EU, hw-thread, work-item, k) \rightarrow \text{(read-set-entry)}\), where the lower case \(k\) represents the index of \(K\) within the work-item. The three examples in Figure A.5 have exactly 5376 elements of a read-set to process. Based on these identifiers within the Instant Kernel, and applying the formula used in Figures 3.12 and A.5, we have:
Note We were not able to replicate this strided memory access on the AMD Ryzen Vega 11 iGPU, because there is no clearly defined notion of a hardware thread in the AMD architecture documentation. We attempted a straightforward replica of the memory access pattern discovered for Intel, but the the kernel did not fetch the entire read-set (failed the smoke-test). The same access pattern may not be directly applicable because of major architectural differences among these two platforms. Further investigation is required in this small grey area, although not imperative, because both platforms (Intel & AMD) performed best in the iterative blocks architecture (subsection 3.4.5).

### 3.4.5 GPU only - Instant Kernel - Iterative validation in blocks of MAX occupancy

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 will trigger 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 (as shown in Figure 3.13), until the entire device re-submerges and moves to the next block. It augmented in later versions to perform more work per work-item.

This mode of operation is shown with a coalesced memory access pattern, however it was also modified to work with a strided memory access presented in the previous subsection 3.4.4. We present
an analysis of the effects of varying the number of elements per work-item and various memory access patterns in Figure 4.3, (chapter 4) and in detail in Tables A.6 and A.6.

Blocks iteration performs better than having each work-item load multiple consecutive (or strided) read-set entries. The overhead of triggering the Instant Kernel iteration number of times apparently outweighs the computations of the offsets and the banking conflicts, possibly occurring when thousands of work-items iterate individually over the read-set (GPU17, Figure 3.9). Additionally, the synchronization on the stop/invalidated condition no longer happens on global (high latency) memory within each work-item in the kernel (GPU18, Figure 3.9) but once, in the CPU delegate thread, between iterations.

![Figure 3.13: Validating in blocks of maximum occupancy - Intel](image)

### 3.4.6 CPU-GPU - Instant Kernel - cooperative validation with blocks

As empirical evaluation will show (Figure 4.7), the GPU by itself was not performing better than the single-threaded CPU validation in either version of the validation tool we have presented. With this new 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.

At first, we statically varied the proportion of the read-set (in percentage) to be validated by the CPU and the GPU. We attached the GPU to a single STM thread (we later refer to it as sticky thread mode). Variation of work attribution, while varying 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 array walk (with sequential and random element accesses [section 2.4]), with no computations in between loads and stores - making it as transparent as possible, and the results (subsection 4.2.3) approximately applicable to other benchmarks (with similar hardware configuration).

The CPU validated the same way as it would in an unaltered, default version of the STM, albeit with less elements. There is a clear picture the number of element assigned to each device. The starting
address and work-item offsets are all predefined. Thus, full GPU validation can be done on the GPU assigned read-set portion, using any of the previously presented (coalesced or strided) memory access patterns, and most importantly without CPU synchronization.

However, static work assignment is not at all practical. It requires to know the volatile read-set size of the application’s transactions during compile time. Not knowing when either device must stop poses an issue in the absence of a static assignment of work. The (computationally) cheapest way of solving the given synchronization issue is to begin validation from opposite ends, and develop towards the middle - stopping once intersected or invalid. However, the GPU works at the granularity of a work-group, and the Instant Kernel running on Intel has 24 work-groups which must now 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 (a method introduced in subsection 3.4.5). The concept is simple (represented in figure Figure 3.15): the device submerges into work by validating a block and re-surfaces to synchronize. The GPU notifies the CPU whether the transactions should be aborted because it found a discrepancy in the read-set, and queries the CPU for the same information.

It was much simpler to handle offsets inside on the GPU sequentially, so we made it begin at position 0 of the read-set. Letting the CPU start at r_set.size Entries, and working its way downwards. Most modern compilers optimize pointer decrements the same way they do increments.

In Figure 3.14 we show the worst case opportunity for GPU contribution, according to our analysis of time spent in validation (Figure A.11). In our observations, the average time spent inside validation was around 15% of execution time (array traversal micro-benchmark). The overall GPU’s contribution is a 1.875% lower bound, assuming the GPU to be always attached to a single thread. However, if we assume the best case where validation can occur at non-intersecting time across all STM threads, then the GPU can co-validate each request and its utility becomes 7.5% * 4threads = 30% of total execution time.

For a more dynamic 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. We did, in fact, experiment with the sticky thread approach, and it produced worse performance than having to compete for the acquisition of the GPU. This is because threads do not always validate their read-sets simultaneously and there is a period of quiescence in the sticky thread where no validation occurs. It has a perpetual hold on the device even when not using it, preventing other threads from its acquisition.
Figure 3.15 shows the CPU and the GPU moving in opposite directions until they meet in the middle. Synchronization for the stop condition occurs in between blocks. Since work-groups complete more or less simultaneously, it makes more sense to synchronize only once in the GPU delegate thread, right before issuing a new block, than 24 times (once inside every work-group). Figure 3.16 shows a more immediate stopping procedure, where each work-item performs a costly synchronization with the CPU in every round. It can be useful when it is known that transactions frequently abort. We leave the trade-off analysis for future work.

Figure 3.15: CPU-GPU cooperative validation - **block** level synchronization

Figure 3.16: CPU-GPU cooperative validation - **work-item** level synchronization
3.4.7 CPU only - multi-threaded validation - CPU validator threads

In this Section we present an alternative validation scheme (Figure 3.17). It 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 \textit{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. Synchronization on abort information is shared by a \textit{mostly-read;written-once} counter, which is written to only if the read-set is invalid in some thread’s chunk. False sharing between threads is avoided because a \textit{read-only} variable’s cache-line is only invalidated once written to.

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

The target platform for these tests was our Intel testbed machine. AMD’s CPU is weaker, and is only useful for its more powerful GPU. Since the GPU is not used in this version of the validation tool, we leave out the performance analysis with CPU validator threads on the AMD platform for future work.

3.5 Overview

In this chapter we discussed modern software abstraction for GPGPU programming using an unconventional Persistent Threads kernel with \textit{instant submission} capabilities, that are tightly coupled to the underlying hardware configuration. These capabilities allowed us to break the boundaries of the traditional high latency kernel enqueue process, in which computations must overtake kernel launch times by an order of magnitude for work offloading to the GPU to be justifiable.

We have presented various modes of operation of our validation tool (in section 3.4), and discussed the various iterations of the development process. In each step, new discoveries were iterated upon throughout the development pipeline, and helped us - within the best of our efforts - arrive at the final solution. The end result became a CPU-GPU cooperative validation tool that proved to be the best performant mode (results in chapter 4) out of all that were discussed. This novel validation model allowed us to augment transactional memory capabilities beyond the traditional, single-threaded sequential validation.
Chapter 4

Experimental Evaluation

The previous Chapter presented the various architectures/modes of the developed validation tool, and described their differences. We shall attempt to verify the concept of validation offloading/co-validation, orchestrated with some existing benchmarks, as well as some micro-benchmarks that we have created for specific use cases.

We also reason about the most beneficial conditions for the use of our validation tool, when coupled with a state-of-the-art, word-based, single-version LSA algorithm such as the one used in TinySTM. 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 degrees of parallelism.

In this Chapter, we compare the various versions of the validation tool that we described previously (in chapter 3, section 3.4), and present our main contributions with a final, most performant architecture. The Chapter starts in subsection 4.1.1 with a brief analysis on the effects of a strict (sequentially consistent), and a more relaxed memory synchronization order, applied to load/store operations within our most basic kernel. Figure 4.1 in subsection 4.1.2 deconstructs and analyses the various validation kernel stages, to discover any potential bottlenecks and performance opportunities.

Benchmarking of the tool begins in a single-threaded environment, under which we created its first versions, and evaluate the number of reads-validated/s, the GPU is able to validate in comparison to the CPU. TinySTM is modified to perform validation at commit, even though there is no contention. This presents the simplest sandbox environment from which we obtained estimations from our early work.

Section 4.2 reintroduces our own transactional array walk micro-benchmark, with its large variety of configurable read-set sizes. It was heavily relied upon during development, because of its simplicity and transparency (i.e., we were able to easily provoke LSA snapshot extensions with small adaptations). In subsection 4.2.1 we discuss early developed GPU-only validation strategies, then move onto an analysis on the optimal choice of the per work-item work assignment in the blocks mode of operation in subsection 4.2.2. Delving closer to the final solution, we present subsection 4.2.3, which shows the results obtained from empirically identifying the optimal work partition between the CPU and the iGPU.
through static assignment, before subsequently arriving at the final - dynamically split alternative in subsection 4.2.4.

Starting at section 4.3, we cross over into the multi-threaded realm, for a closer look at performance in real-world transactional applications that have parallelism. In subsection 4.3.4 we show the effects of enabling the LSA algorithm, and finally arrive at the most important transactional throughput metric in subsection 4.3.6, with the transactional array walk micro-benchmark (introduced and described earlier, in section 2.4).

Finally, section 4.4 evaluates our work in a more realistic, high caliber benchmark - STMBench7, with long traversals enabled to produce long running transactions. The selection of this precise subset of program configurations was guided by the analysis produced from section 2.6.

The tests in this Chapter have been executed on two machines (unless stated otherwise) from competing manufacturers with entirely different OpenCL2.0 enabled APU architectures and shared virtual memory capabilities: Intel Skylake i7-6700k CPU with 8 threads through hyper-threading with an integrated HD530 graphics in a GT2 configuration, and the AMD Ryzen 2400g CPU with Vega 11 graphics.

### 4.1 Instant Kernel evaluation

#### 4.1.1 Memory ordering semantics and work-group optimality

In GPU programming, there sometimes occurs a need to execute as fast as possible, and possibly ignore consistency in the process. In this case, synchronisation between work-items inside work-groups may not achieve forward progress with relaxed memory consistency when there is divergence among work-items. But when there is no such divergence, everything appears to work flawlessly. However, in parallel programming, such scenarios are very unrealistic. The workload size is never precisely a multiple of maximum occupancy and there is always bound to be divergence among threads in validation schemes (e.g., a lock being taken for some read entries and not for others - for work-items in the same work-group).

The correct execution of our kernel relies on atomic access with stronger consistency - with at least acquire-release semantics. In a transactional environment there is always bound to be branching divergence (e.g., many elements in the read-set will not correspond to their assigned global lock). Nevertheless, we presented the effects of using a more relaxed C11 memory model as if these constrains had not existed. With specific test cases, that have no branch divergence on the GPU, it was possible to
obtain communication overheads for various memory orders. According to the communication analysis in Figure 4.1, the best *Instant Kernel* configuration is to have 24 work-groups with 224 elements each: $24 \times 224 = 5376$ (full device utilization for Intel). In conclusion, having a stricter memory order does not result in a substantially large decrease in *Instant Kernel* polling latency ($\approx 2 \mu s$). Compared to the overall execution time of validation kernels, having correctness is not expensive.

### 4.1.2 *Instant Kernel* deconstruction

In Figure 4.2 we present an *Instant Kernel* kernel deconstruction analysis. It executes a simple transactional array benchmark with systematically subtracted features. This allows us to measure the overhead of an *Instant Kernel* with *Persistent Threads* in the validation algorithm. The underlying STM behaves regularly, as if validation had been executed, but we remove some major portions of the program. The following operations are performed by each work-item of the persistent kernel:

- 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 quite trivial with regard to the branching logic (last step). In a single-threaded STM environment, every work-item descends into the same branch - they never load the write-set address of the lock’s owner. In a multi-threaded STM environment, this branching will have a performance degrading effect on the GPU. It is difficult to estimate how much, without a more sophisticated set of experiments with contention.

From analysing the kernel deconstruction in Figure 4.2, we see that the gap from calculating offsets to loading read-set entries is almost negligible in terms of reads-validated/s (the read-entry structures are all packed together in a contiguous array).

Clearly, the most expensive set of operations is the de-referencing of numerous arbitrary addresses from the read-set entries into the lock table. We expect CPU threads to have an advantage in this case, as they access and cache these locks in a much larger LLC, with a much smaller memory access latency, and cache them prior to performing validation (e.g., performing writes, retrieving timestamps).
4.2 Transactional array traversal

Assessment of the memory order semantics and the complete kernel deconstruction, which were reported in the previous sections, allowed us to have a better understanding of the communication overhead introduced in validation in a Persistent Kernel on an integrated GPU device.

Before moving onto other benchmarks, we reintroduce the transactional array walk in this Section, and soon after, compare a full GPU validation (with different memory access patterns) against and the iterative blocks kernel in in subsection 4.2.1. In subsection 4.2.2 we encounter the optimal read-set distribution \((K)\) among work-items in every block. Soon after, in subsection 4.2.3, we show our discoveries with regard to a cooperative static assignment of work between the CPU and the GPU, and analyze a subsequent dynamic assignment algorithm in subsection 4.2.4. Finally, we summarize our findings in subsection 4.2.5.

For simplicity, early versions of our algorithm do not account for multiple STM threads. They were developed to evaluate the GPU’s operating capabilities over TinySTM’s data structures through the SVM, and to analyze the communication overhead. However, in a single-threaded environment, read-set validation does not get triggered because it serves no purpose. A single thread has no contention, and automatically commits through metadata based validation by looking at the standstill of the global time (commit algorithm of TinySTM in Figure 2.14). The behavior of TinySTM’s commit protocol had to be slightly modified to always preform validation of the full read-set, for the sole purpose of single-threaded benchmarking.

**TX array traversal** is a simple micro-benchmark that does not perform any expensive computations in between loads and stores. Throughout this micro-benchmark we vary the read-set size in powers of 2 - starting at 512, and moving up to 130M elements. We made the read-set size as large as possible to consider benefits from using the GPU on any possible occasion.

TinySTM’s ownership records (locks) array was increased from \(2^{20} \approx 1M\) locks addressing \(4M\) words to \(2^{26} \approx 67M\) locks addressing \(268M\) words to address very large work sets (4 addresses map to a single lock). Our benchmarks evaluate up to \(130M\) elements. This is an upper bound of stdlib posix::memalign’s function for an aligned memory allocation, without recurring to external libraries. Moreover, in a multi-threaded environment the working set only grows to a maximum of \(16M\) elements to address every element in the disjoint arrays \(\approx 130M/8\) threads \(\approx 16M\).

4.2.1 GPU-only validation

This subsection focuses on the early benchmarks used during the development of the validation tool on the Intel platform, which were later revisited to compare with AMD. Figure 4.3 shows the reads-validated/s performance of a “full GPU” validation for the Intel platform, and Figure 4.4 for AMD. All of the following Figures are in a semi-logarithmic scale, unless stated otherwise.

**Top left** (Figure 4.3) varies the number of work-groups, and consequently, the number of work-items per work-group. This choice indicates the mapping of work-groups to hardware threads (Figure 3.4)
and dictates the occupation of one whole Execution Unit with a single group (in the current case of 24WKGPs, 224 work-items/group). Alternatively, an execution unit is partially divided by two pieces of a work-group as the Persistent Kernel is launched with 48 work-groups. The global work size remains fixed because of the requirements set by the Persistent Kernel.

Synchronization between work-items on the GPU (through atomics) takes on the acquire-release memory order, paired across loads and stores to the same memory location. We gather that the work-group size assignment does not play a significant role in performance.

There is a slight dip in performance after the GPU’s L3 is saturated at the ≈65k element mark. A further decrease in slope at the 2MB mark may indicate that the integrated GPU has at most that chunk of the LLC attributed to it. At the ≈1M read-set size mark (8MB) we see a drop in performance of the GPU kernel because the read-set no longer fits in the LLC. Performance drops at this point, as the kernel begins systematically accessing the DRAM.

**Top right** (Figure 4.3) shows the difference between the coalesced (subsection 3.4.3) and strided (subsection 3.4.4) memory access patterns. Strided memory access has an improvement of up to 2.46x (at 1M entries) over its coalesced counterpart.

**Bottom** two sub-figures (Figure 4.3) present an empirical discovery of an optimal number of elements to be processed by every kernel-instance (work-item) of a Persistent Kernel. There is, however, a major difference in the Blocks kernel operation. In the top two sub-figures (coalesced and strided), every one of the 5376 GPU work-items would iterate over a series of read-set elements they have to process. With the Blocks mode of operation, a single iteration is performed on the CPU, while it awaits for the GPU to process a block of 5376 read-set entries (none of which have cycles or offset calculations). Additionally, the metrics extracted from the coalesced and strided memory access kernels no longer have to be collected within the kernel, but can be extrapolated with every new block on the CPU.

The best K-value (read-entries/work-item) we are looking for is only valid for a Persistent Kernel on this hardware. It is a direct influence of the MLP (section 3.2.3), and and carries over to other STMs on this hardware. K will increase (better performance) with a more compact STM read-entry data structure (16 byte in TinySTM), and decrease otherwise. **Bottom left** (Figure 4.3) shows the exhaustive search for the optimal K-value with a coalesced memory access, and **bottom right** (Figure 4.3) - with strided.

Overall, the Blocks mode beats kernels with offset calculations by as much as 1.5x. It seems that loading 2-4 read-entries/work-item leads to best performance throughout the entire read-set range. We take a closer look at the different K-values in Figure A.6, and compare them with the baseline TinySTM.

In Figure 4.4, we show the same metric as previously (reads-validated/s), but for the AMD platform. With an exception that we could not replicate a strided memory access on the AMD Ryzen Vega 11 iGPU. There is no clearly defined notion of hardware thread in the AMD architecture documentation. By attempting a straightforward replica of the memory access pattern for Intel, we could not get the kernel to progress. 11246 work-items with 44 work-groups of 256 work-items and a coalesced memory access is the most performant AMD configuration across the read-set size spectrum. It peaks with 360M reads-validated/s at a ≈130k read-set size - which beats Intel’s coalesced but not strided memory access kernel.
Figure 4.3: Single-threaded array traversal, random element access - Intel

Figure 4.4: Single-threaded array traversal, random element access - AMD
4.2.2 Optimal number of read-set entries per work-item (K)

In this section we would like to give a more detailed comparison of the full GPU validation in a Blocks mode of operation, shown in Figures 4.3 and 4.4. Table A.6 shows the amount of reads validated per second when varying K (on the y axis) on the Intel platform, and Table A.7 on AMD. K represents the number of elements from the read-set which are validated by each individual work-item.

From a theoretical point of view, this number depends on three things. First, the memory space left on each thread’s GRF after Persistent Kernel state variables, second, the device’s L3-data cache size, and finally third, register pressure from Memory Level Parallelism (section 3.2.3) which is based on the number of memory requests being issued by the kernel.

This experiment is set-up to have every work-item doing work. This is why we are only interested in K values that would employ the whole GPU - values of \( K < \frac{\text{rset-size}}{5376} \). Anything above the diagonal (in Tables A.6 and A.7) would result in some work-items to idle.

We have highlighted the fastest K value of being around 2 for Intel coalesced and 1-3 for strided in Figure A.6. We only showcase a random array traversal, because GPU sequential array element access never surpasses the baseline TinySTM CPU validation for any value of K.

4.2.3 CPU-GPU cooperative validation - static work partition

We present an analysis of a static partitioning of the read-set among the CPU and the GPU, for simultaneous validation, as described in detail in subsection 3.4.6.

Since we have a static assignment of work, the GPU validates with a coalesced memory access at precisely known bounds. In this kernel, every work-item loops over neighbouring read-entries when the read-set is larger than maximum occupancy.

Regular TinySTM beats this kernel mode by as much as 2.75x in a sequential element traversal in the transactional array micro-benchmark Figure 4.5 (right). This is quite expected because the CPU’s prefetching capabilities are highly optimized, and it has a much lower memory access latency. Because the read-set is accessed sequentially, and locks map to four sequential addresses, the access to the lock table is also going to be sequential and consequently makes use of prefetching and caching.

The random array traversal Figure 4.5 (left), however, shows an increase in performance over the baseline TinySTM’s reads-validated/s, beginning at approximately 250k elements (with a higher CPU work proportion), and maintains superiority over the baseline throughout the larger read-set range - until reaching \( \approx 130M \) elements. The speed-ups for each intersection of the read-set and static workload partition are discriminated in Figure A.8. The ideal partition of work is close to 50% at very large read-sets. In smaller read-sets (\( \approx 250k \) elements) it is closer to 75% for the CPU.

Figure A.8 displays an irregular "bow-tie" shape for the discriminated speedup data. The GPU’s contribution is noticeable starting from a read-set of \( \approx 500k \), and around \( \approx 500 – 1000k \) the CPU (main contributor) hits the limit of its LLC capacity. After that, the right half of the shape broadens, and we begin seeing bigger improvements due to the presence of the GPU. The best improvement appears to be focused around the 50% mark with a speedup of up to 1.7x.
4.2.4 CPU-GPU cooperation validation - dynamic work partition

In Figure 4.6, we present a validation volume comparison between the CPU and the GPU, in an array traversal program, with random element access performed on our Intel platform. The setting is a single-threaded, dynamic work split between the CPU and the GPU (algorithm from subsection 3.4.6), where the two devices move in opposite directions and towards each other. We compare the number of elements validated by each one, throughout the numerous executions of the same program in Figure 4.6.

With small read-set sizes, data from the GPU is omitted. We can conclude that the GPU takes more time to react, complete an iteration of a block of 5376 elements and synchronize, than the CPU to completely validate small read-sets on its own.

From Figure 4.6, we observe that the GPU catches-up to the CPU in the amount of reads validated around 500K elements. The biggest leap occurs in between 2M and 4M elements, followed by a steady increase in the GPU's contribution as the CPU begins systematically hitting the DRAM access latency.

In contrast to the static-partitioning in Table A.8, the benefits of the GPU in a dynamic setting begin at the same 500K elements. However, in Figure 4.6, the GPU reaches larger increases in raw validation volume 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 has $\lceil \text{readset}/5376 \rceil$ elements per work-item. Finally, it is noted that the relative cost of synchronization between the devices is a systematic loss of $\approx 2K$ read-set elements, represented as as wasted val reads in Figure 4.6

4.2.5 Single-threaded summary

We would like to present the best kernel validation programs for each platform:

- Intel i7-6700k, HD530: cooperative validation in blocks of $21504 = 5376 \times (K = 4)$; strided memory access;
- AMD 2400G, Vega11: cooperative validation in blocks of $45056 = 11264 \times (K = 4)$; coalesced memory access;
- Intel Skylake i7-6700k: CPU validator threads; 4 workers;
Figure 4.6: CPU-GPU cooperation performance comparison - random array traversal
GPU's superiority over the CPU, in the number of \textit{reads-validated/s} in a random array traversal, speedup is represented as a multiplier on the xtics, under the bars.

Figure 4.7: CPU-GPU-cooperative, CPU validator threads - speedup over TinySTM-untouched in \textit{reads-validated/s}
Recommended GPU employment for cooperative validation when using the Intel HD530 GPU - at the \(\approx 250\)k mark.

Figures 4.7: Cyber-physical systems, cooperative validation - speedup over TinySTM-untouched in \textit{reads-validated/s}

Despite a weaker performance in smaller datasets, the GPU validation tool enhances the system in \textit{reads-validated/s} on the Intel testbed, starting from \(\approx 250\)k elements in a read-set, according to the relative speed-up to the baseline TinySTM-untouched (Figure 4.7), with an up-to 2.25x increase in performance. From the 250k mark, there is some noticeable improvement. However, peak performance is attained when the CPU reaches its L3 cache capacity with \(\approx 2 - 4\)M elements. After that point, the GPU's contribution is very noticeable.

There is however, a small enough difference in terms of \textit{reads-validated/s} between the values of \(K\) to consider the following. As \(K\) increases, so does the block size. In a cooperative validation, the greater the block, the longer it takes the GPU to sync-up with the CPU after re-emerging (subsection 3.4.6) from
the block, in between iterations. This latency can cause the algorithm to spend too much time in between sync with the CPU, and on a potentially already invalidated (by the CPU) read-set, thus preventing the GPU from being relinquished as early as possible and subsequently captured by a new thread.

4.3 Multi-threaded evaluation

After having analysed the various validation tool modes in a single-threaded STM validation, and having applied some optimizations (notably to the amount of work done per GPU work-item), we begin the evaluation of our modified STM system in a more realistic multi-threaded environment.

4.3.1 Alternative design - CPU validator threads

The employment of the GPU incorporates all available computational resources in an APU for transactional validation. Analogously, and while parallelism remains low, we would like to consider the potentially idle CPU hardware threads for the same purpose intended in our work with the GPU. As it was shown that in a single-threaded environment, where hardware threads are not occupied with transactions, the CPU validator threads pool was able to outperform any CPU-GPU-cooperative validation (Figure 4.7 and Figure 4.8). We expect the utility of validators to decrease as the amount of parallelism in an STM environment increases.

We also ponder the downsides of having a CPU thread pool solely dedicated to validating the read-set interfering with the scheduling of regular transactional threads. This approach is presented as an alternative to the originally proposed work. It poses an interesting design choice in situations with very high contention, where too many STM threads frequently abort. For better performance, this exemplified setting could incur a reduction in the degree of parallelism of the STM system, and can exclusively dedicate resources to validate the read-set.

For this test, 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/s with the array micro-benchmark, in Figures A.9 and A.10, the optimal thread count is between 4 and 8 threads. From the figures, the trade-off seems to be apparent - for the same overall performance, employing half the threads (4) is a better choice.
4.3.2 Multi-threaded array traversal

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 (Lines GPU21-GPU29 Figure 3.9). A multi-threaded environment brings us one step closer to real-world STM applications. It was expected that the presence of contention among STM threads would negatively affect GPU performance due to kernel thread divergence during lock availability verification. The cost of branching on a GPU is much higher because of a lockstep execution of work-items.

The array traverse program was modified to include strong and weak scaling (section 2.4), to allow each thread to have its own (largest possible) read-set without having contention. Another notable change in the program (from single-threaded to multi-threaded) is the addition of many more transactions during each execution. Previously we would run a single-transaction, single-threaded program 20 times for statistical significance. In the new, multi-threaded version, we execute each program with 100 transactions 20 times on each thread, for each read-set size.

In a multi-threaded array traversal with disjoint sets (Figure 2.9), the threshold for the recommended GPU utilization might become lower, and more favourable, than the \( \approx 250k \) elements, as seen in single-thread results (Figure 4.7, subsection 4.2.5). 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 4.9 shows the results of our validation tool (in \( \text{reads}-\text{validated}/s \)) being applied to multiple STM threads (speedup table in Figure 4.10). It includes executions of a cooperative validation on both Intel and AMD platforms, and CPU validator threads normalized to TinySTM-untouched. The figure also demonstrates the LSA-enabled counterparts of the aforementioned programs, which are represented as dashed lines of the same color as their regular executions.

The experiments performed with disjoint array sets (Figure 4.9) do not have a single abort, because transactions operated isolated from each other. As a consequence, during commit, each transaction will validate the entirety of its read-set. TinySTM does, in fact, trigger validation when concurrent transactions incremented the global clock by committing their own transactions.

A noticeable difference from a single-threaded AMD-cooperative execution to multi-threaded is a much earlier improvement over the baseline TinySTM (\( \approx 4000k \) [Figure 4.8] down to \( \approx 250k \) [Figure 4.9]). We ponder that, as the number of transactions increased, so did the cache locality of read-set elements in the AMD architecture. Further investigation is required into this unusual phenomenon, only having affected AMD. Overall, AMD performs the worst of all augmented/cooperative validation schemes.

There is no significant change in \( \text{reads}-\text{validated}/s \) when moving up from 1 to 2 threads with large transactions (Figure 4.10). From 2 to 4, however, the reduction is much greater because the GPU can only co-validate a single STM thread at a time, so its utility is diminished slightly with an increase in parallelism, however still remains quite noticeable.
**Figure 4.9:** Disjoint set array traversal - reads-validated/s

Threads operate on isolated data sets (no contention). GPU’s ability to aid validation scaled with parallelism.

(speedup over TinySTM-untouched in reads-validated/s).

**Figure 4.10:** Disjoint array set traversal micro-benchmark - CPU-GPU-cooperative, CPU validator threads - speedup over TinySTM-untouched in reads-validated/s
4.3.3 Effects of contention

The simplest way of introducing contention into the benchmark was to depart from disjoint array traversals and make every STM thread operate within the same array of read-set size (conjoint array traversal). Unsurprisingly, there is a decrease in performance (in reads-validated/s) with the addition of contention (Figure 4.11). The overhead of initializing the request (capturing the GPU), synchronization, to only be invalidated by another transaction while validating, makes the GPU co-validation perform worse in a workload with contention. The GPU does not have the opportunity to reach an ideal work-size and assignment balance. This effect becomes more prominent as the degree of parallelism and contention increase, thus reducing the actual number of reads validated in Figure 4.11.

![Figure 4.11: Intel-cooperative validation - Introduction of contention into the benchmark](image)

Data normalized to TinySTM-untouched-Intel (speed-up in reads-validated/s).

4.3.4 LSA favorable workload

This experiment consists a heterogeneous workload of large, long running transactions being interfered with (once during their lifetime), and instead of aborting would have a chance to commit, by attempting to extend their snapshot into the current global time (requires re-validation). It’s design was previously explained in subsection 2.6.1, Figure 2.15.

This experiment has next to no contention, as transactions write to their neighbours once per round, for 100 rounds. It was performed on the Intel machine only, as we find that a single test-bed to be sufficient for a relative comparison. Its goal does not depend on the speed of execution, but aims to show a ratio between the read-set size and the proportion of time spent in validation.

According to the obtained results in Figure 4.12, enabling the LSA algorithm succeeded in reducing the number of aborts (dashed lines represent LSA algorithm versions) throughout the entire read-set.
size variation and thread count, on all programs with LSA enabled. However, LSA reduces the overall
time spent in validation, thus reducing the usefulness of our tool. Having a lower number of re-starting
transactions incurs less time spent validating read-sets.

And finally, transactional throughput (commits/s) increases with LSA in this workload, as seen in
Figure 4.13. However it fluctuates in an unpredictable manner with parallelism, possibly because with 4
threads, the timing of LSA trigger was unfavorable. Meaning extensions were performed too early in the
transaction and read-sets were too short when extension applied, because of a lack of synchronization
among STM threads on the timing of the neighbouring writes.

![Disjoint round-robin write - number of aborts](image1)

Figure 4.12: Disjoint round-robin write - number of aborts
Data normalized to TinySTM-untouched-Intel. LSA program executions (all dashed lines) reduce the number of
aborts throughout the entire read-set size spectrum.

![Disjoint round-robin write - TX throughput](image2)

Figure 4.13: Disjoint round-robin write - TX throughput (speed-up over TinySTM-untouched)

Read-set size influence on LSA vs. non-LSA designs

The time proportion validating is lower in a conjoint array workload (Figure A.12) because contention
systematically invalidates transaction in a multi-threaded environment, and does not allow them to finish
validating the entirety of their read-sets (abort/restart). In Figure A.12 we see a reduction of time spent
in validation, with the enabling of the Lazy Snapshot Algorithm which provides snapshot extensions in
the underlying STM. Every dashed line in the figures is an LSA-enabled counterpart of the programs represented with solid lines.

This is a relative comparison with the enablement of a multitude of STM design choices to provide LSA support. The tests were performed on the Intel platform only. As expected, the overhead of enabling epoch garbage collection, a modular contention manager and the reading of locked data to produce LSA, has too large of an overhead with non-contended workloads. It is only justifiable in large transactions with a precisely timed abort (nontrivial to replicate as shown in section 4.3.4).

4.3.5 Aborts

Besides accelerating validation, we must also keep the number of aborts from growing past the unaltered version of the underlying STM, because the throughput of the STM will decrease, and the utility of our validation tool hinder.

Figure 4.14 shows a comparison as to how many more transactional aborts were added to the STM’s operation because of our GPU validation module. By varying the read-set size range from a very low number of elements, we can further evaluate our validation tool, and pin point the least amount of work necessary to have the GPU provide utility.

The number of aborts is up to 3x as high at \(\approx 16k\) elements with 2 threads, and \(\approx 32k\) with 4. At 8 threads there is systematically more aborts than in the default TinySTM across the entire read-set range. This inability to scale is attributed to the 8 threads being hyper-threaded on four cores (shared L2).

![Figure 4.14: Conjoint array - total aborts normalized to baseline](image)

Data normalized to TinySTM-untouched-Intel. We choose conjoint array because disjoint array sets do not contend for the same memory words and do not have any aborts.

4.3.6 Transactional throughput analysis

The most important metric to consider in an application that uses an STM in a multi-threaded environment is the transactional throughput. It accounts for the number of transactions committed during the execution time of the program.

Figures 4.15 and 4.16 show the number of commits/s, normalized to the baseline version of TinySTM, produced by the various versions of our validation tool on both testbed platforms, CPU validator threads,
and the same programs with the LSA module enabled.

Unfortunately, in comparison to the reads-validated/s metric, the results are not that promising. There is some improvement over the baseline version of TinySTM in every thread count which appears to be insignificant enough to warrant the application of the iGPU at this point. Some analysis is required to further study the cause of such a discrepancy in a large increase in validation throughput, and such a small increase in the transactional throughput with this array-traversal micro-benchmark.

Figure 4.15: **Disjoint** arrays: TX throughput

Figure 4.16: **Conjoint** array: TX throughput

![Graph 1](Image)

No contention among threads

![Graph 2](Image)

Moderate contention among threads

*tx/s normalized to TinySTM-untouched-Intel*
4.3.7 Array micro-benchmark - summary

In this Section we applied our specialized array traversal micro-benchmark for an in-depth analysis of the STM's performance with various read-set sizes. We have presented various angles of perspective into our STM validation tool using the following metrics: reads-validated/s, validation-time/execution-time, total-aborts, and commits/s, both in a single and multi-threaded environments.

The disjoint array benchmark spends more time inside validation Figure A.11 for a greater opportunity for the validation tool to be helpful. The best version of our tool, i.e., CPU-GPU-cooperative (with blocks of $K=1$) produce the highest number of reads-validated/s out of all GPU versions (Figure 4.10), and does not systematically increase aborts when attached to TinySTM (Figure 4.14). It beats the CPU validator threads when all hardware threads are oversubscribed to the CPU - at 8 threads - in a small window: between 500k and 4M read-log elements (Figure 4.9).

We now move onto the next Section (section 4.4) where we describe our attempt at larger transactions with lengthier execution times, where the start-up overhead of our validation tool’s Instant Kernel gets possibly hidden away in proportion to the total execution time.

4.4 STMBench7 analysis

Earlier in this Chapter, we performed a meticulous study into the number of aborts and reads-validated/s, in order to fine-tune our solution. 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 now depart from our micro-benchmarks into popular TM benchmarking software that was presented in section 2.4.

Previously, in section 2.6 we performed a study on the amount of reads-validated in these transactional benchmarks and their various input parameters. Out of the most popular benchmarks, some STMBench7 variants produced the largest number of reads-validated/validation-call, thus delivering the best potential utility for our cooperative 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 section 2.6, with long traversals enabled.

TinySTM - STMBench7 We expected the GPU to vastly under-perform, as it was shown in Figure 2.3, 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 (commits/s in Figure 4.17 vs. 4.18), our validation tool outperformed the baseline TinySTM by up to 2.1x in commits/s, as shown in Figure 4.19, 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.
STMBench7 commits/s - 5 second runs vs. 20 second runs

The program names stem from their sequence of input parameters that have been previously listed in Table 2.3.
Chapter 5

Conclusions and Future Work

From the *Simulation of Intel’s Integrated GPU Architecture* [14] (section 2.1.2) and *The APU in various workloads* (section 2.2) studies, it was clear that our test-bed integrated GPU is exceptional at performing floating point SIMD computations, but slower than the CPU in tasks with a high memory access latency. The most computationally intensive task our algorithms perform are read-set offset calculation. The rest are arbitrary memory locations loads, and address content and branch comparisons.

An STM system is a highly optimized software environment. It is built with an utmost regard for performance, as it competes with popular locking mechanisms such as *mutexes* and *futexes* that have relatively low overhead. It is susceptible to rapid and unpredictable change. As such, even measurements may occasionally affect the performance of the STM. The implications of our changes to the testbed STM became quite apparent, and negatively affected the overall performance, even though caution has been taken as to not interfere with the STM’s regular state of operations. Given some discrepancy in performance in terms of reads-validated/s and commits/s in our transactional array micro-benchmark (Figure 4.9 and Figure 4.15), we ponder the possibility that the inclusion of OpenCL abstractions into the highly optimized environment such as TinySTM undone some crucial performance optimizations. However, given enough execution time to hide the *Instant Kernel* initialization, the system performance straightened itself out and managed to overcome the baseline, untouched TinySTM in long running, large transactions in the STMBench7 benchmark (section 4.4).

Based on the reads-validated/s estimate alone, we determined an approximate threshold starting from which GPU cooperation is justifiable - a read-log volume of \( \approx 250k \) elements (Figure 4.10). Additionally, and according to the results from the STMBench7 benchmark, we conclude that long running systems are additionally beneficial to our validation tool, as it is able to service more well timed validation requests (where the iGPU is not wasted on short, doomed [to be invalidated early] read-sets).
5.1 Main contributions

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

We provide evidence to suggest that a partial offloading of validation to another device has placement in transactional memory applications. All of our results that were measured in reads-validated/s demonstrated a significant improvement in performance compared to a baseline, unaltered STM (Figure 4.10). Our cooperative validation tool shows promise in raw number of elements validated, and given enough time to operate, achieves best utility by servicing more STM validation calls, and delivers an improvement in transactional throughput of up to 2.1x, with large read-sets in STMBench7 (Figure 4.19).

We have also applied a novel (strided) memory access pattern to an Instant Kernel/Persistent Threads programming paradigm, with offset calculations based on the permanent residency of workgroups in hardware threads, which showed an improvement over other access patterns (Figure 4.3).

5.2 Future Research

Write-set validation - We encountered another opportunity to employ the GPU in a similar manner as we do for the read-set. For instance, TinySTM has a section of code that validates the write-set, and some study is required into its frequency of use, as well as to look into the inclusion of this mechanism in the design of other similar STMs such as swissTM.

Continuous validation - A prospective algorithm could attach the GPU to a single STM thread and perpetually attempt to maintain the read-set valid at all time by guaranteeing the validity of the read-set until some discrete or possibly scalar time in the system.

NORec - The final version of the cooperative validation tool should be applied to another STM.

More powerful hardware - This year there has been significant breakthrough in integrated GPU development with the Intel GEN11 XE integrated GPU. This more powerful model packs a triple amount of execution units per subslice, an increased L3-data cache, as well as a larger SLM. This new configuration will most definitely increase the number of reads-validated/s.

Fine grained work-group validation - We experimented with, but ultimately abandoned for simplicity in development, the sharing of the iGPU on a finer-grained level than presented in this work. Future work should consider a finer grained/non compete employment of GPU resources through the Persistent threads programming model, 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.

Non blocking, GPU-only validation - finally, we would like to experiment with a validation architecture where a thread initializes GPU-only validation for a very large read-set and immediately begins another transaction until the GPU terminates to switch back and complete the rest of the commit/abort protocol.
Bibliography


Appendix A

Auxiliary Figures

A.1 Background

A.1.1 AMD Vega GPU architecture

*Discrete GPU – Physical Device Memory; APU – Region of system for GPU direct access

Figure A.1: AMD Vega 10 GCN architecture

AMD GCN VEGA architecture has four vALUs per Compute Unit. Four processing elements per vALU. Vega 11 iGPU has a total of: 11 Compute Unit * 4 vALUs * 16 Processing Elements = 704 Shading units [27]).
A.1.2 The TinySTM API

```
main()
    TM_STARTUP;
    Create 8 threads:
    TM_SHUTDOWN;
    }

void parallel_region(){
    each of 8 threads calls
    TM_THREAD_ENTER();
    when thread enters region
    subfunction();
    subfunction();
    TM_THREAD_EXIT();
    }

void subfunction()
    TM_START();
    do work to RW shared data
    multiple calls to TM_READ, TM_WRITE
    TM_END();

TM_START();
    do work to RW shared data
    multiple calls to TM_READ, TM_WRITE
    TM_END();


```

Figure A.2: Application interaction with TinySTM
A transaction reusing thread local storage transactional descriptors with every atomic block.

A.1.3 GPU debugging

```
struct stm_tx {
    stm_word_t status;
    start, end;
    {r_set, w_set}; -- alloc once
    void *data[MAX_SPECIFIC];
    10x stats counters, ...
}

Figure A.3: GPU debugging method (basic smoke-test)
```
A.1.4 Testbed iGPU characteristics

Table A.1: iGPU comparison (Intel & Amd)

<table>
<thead>
<tr>
<th>Spec</th>
<th>AMD Radeon RX Vega 10 Mobile</th>
<th>Intel HD 530 (GT2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Die Size (mm²)</td>
<td>123</td>
<td>210</td>
</tr>
<tr>
<td>Clock (MHz)</td>
<td>950</td>
<td>1300</td>
</tr>
</tbody>
</table>

**Render Config**

<table>
<thead>
<tr>
<th>Spec</th>
<th>AMD Radeon RX Vega 10 Mobile</th>
<th>Intel HD 530 (GT2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shading Units</td>
<td>640</td>
<td>192</td>
</tr>
<tr>
<td>TMUs</td>
<td>40</td>
<td>24</td>
</tr>
<tr>
<td>ROPs</td>
<td>8</td>
<td>3</td>
</tr>
<tr>
<td>Compute Units</td>
<td>10</td>
<td>24</td>
</tr>
</tbody>
</table>

**Theoretical Performance**

<table>
<thead>
<tr>
<th>Spec</th>
<th>AMD Radeon RX Vega 10 Mobile</th>
<th>Intel HD 530 (GT2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pixel Rate (GPixel/s)</td>
<td>10.41</td>
<td>2.850</td>
</tr>
<tr>
<td>Texture Rate (GTexel/s)</td>
<td>52.04</td>
<td>22.80</td>
</tr>
<tr>
<td>FP16 (half) performance (GFLOPS)</td>
<td>3331</td>
<td>729.6</td>
</tr>
<tr>
<td>FP32 (float) performance (GFLOPS)</td>
<td>1665</td>
<td>364.8</td>
</tr>
<tr>
<td>FP64 (double) performance (GFLOPS)</td>
<td>104.1</td>
<td>91.20</td>
</tr>
</tbody>
</table>

**Design**

<table>
<thead>
<tr>
<th>Spec</th>
<th>AMD Radeon RX Vega 10 Mobile</th>
<th>Intel HD 530 (GT2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>TDP</td>
<td>10W</td>
<td>15W</td>
</tr>
</tbody>
</table>

A.1.5 Methodology - Performance Counters

Table A.2: Counters collected during the analysis of STM execution

<table>
<thead>
<tr>
<th>Counter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Validation time (s)</td>
<td>Total time spent validation.</td>
</tr>
<tr>
<td>Validation time CPU (s)</td>
<td>Time spent validating by the CPU; cooperative validation only (Section 3.7.7).</td>
</tr>
<tr>
<td>Validation time GPU (s)</td>
<td>Time spent validating by the GPU; cooperative validation only.</td>
</tr>
<tr>
<td>Number of Commits</td>
<td>Number of commits</td>
</tr>
<tr>
<td>Number of Aborts</td>
<td>Aborts</td>
</tr>
<tr>
<td>Reads validated</td>
<td>Reads validated until read-set/Orec version miss-match provokes abort, or success.</td>
</tr>
<tr>
<td>Reads validated CPU</td>
<td>Same as Reads validated; cooperative validation only.</td>
</tr>
<tr>
<td>Reads validated GPU</td>
<td>Same as Reads validated; cooperative validation only.</td>
</tr>
<tr>
<td>Wasted reads validated</td>
<td>Because the GPU validates in blocks and synchronizes between iterations (for best performance), it may overlap the CPU's work and double validate; cooperative validation only.</td>
</tr>
<tr>
<td>GPU employment times</td>
<td>Number of times the GPU was employed by current thread.</td>
</tr>
<tr>
<td>Successful validation call</td>
<td>All elements in the read set validated, transaction may commit.</td>
</tr>
<tr>
<td>Failed validation call</td>
<td>Invalid element in transactional read-set, transaction must abort.</td>
</tr>
<tr>
<td>LSA snapshot extensions</td>
<td>(Section 2.2.4)</td>
</tr>
<tr>
<td>Energy (J)</td>
<td>Collected after all threads completed.</td>
</tr>
<tr>
<td>Program execution time (s)</td>
<td>Collected after all threads completed.</td>
</tr>
</tbody>
</table>

Manually instrumented counters gathered from all four STM systems used in this work. The counters are collected individually by all threads and stored in their respective transactional descriptors in memory, until they are aggregated at the end of program execution.
A.2 GPU kernel analysis

Figure A.4: OpenCL 2.0 work-item responsibilities in the Persistent Threads work model. Order of events (Left to right, top to bottom) referencing instructions in Figures 3.7-3.9.

Figure A.5: Strided memory - three examples
Work item 0, hw thread 6 in EU 23 loads element index 167 from the read-set.
A.3 Single-threaded blocks mode kernel - \( K \) elements/work-item

Figure A.6: Intel HD530 - Transactional array walk (random elements) - effects of varying number of elements per work-item

COALESCED kernel memory access

<table>
<thead>
<tr>
<th>( K ) per work-item</th>
<th>( K ) in blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5376</td>
</tr>
<tr>
<td>2</td>
<td>10712</td>
</tr>
<tr>
<td>4</td>
<td>21424</td>
</tr>
<tr>
<td>8</td>
<td>42888</td>
</tr>
<tr>
<td>16</td>
<td>85776</td>
</tr>
</tbody>
</table>

A heatmap of \( \text{reads validated} \times 10^8 / \text{s} \). 1 STM thread - persistent kernel full gpu validation in blocks of \( 5376 \times K \). Data from Blocks in Figure 4.3 and Figure 4.4 (bottom left and right).

Figure A.7: AMD Vega 11 - Transactional array walk (random elements) - effects of varying number of elements per work-item

COALESCED kernel memory access

<table>
<thead>
<tr>
<th>( K ) per work-item</th>
<th>( K ) in blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>11264</td>
</tr>
<tr>
<td>2</td>
<td>22528</td>
</tr>
<tr>
<td>4</td>
<td>45056</td>
</tr>
<tr>
<td>8</td>
<td>90112</td>
</tr>
<tr>
<td>16</td>
<td>180224</td>
</tr>
</tbody>
</table>

A heatmap of \( \text{reads validated} \times 10^8 / \text{s} \). 1 STM thread - persistent kernel full gpu validation in blocks of \( 11264 \times K \). Data from Blocks in Figure 4.4 (right).
A.4 Statically assigned, cooperative CPU-GPU validation

Figure A.8: Statically assigned (%), cooperative CPU-GPU validation (single-threaded) - speed-up over TinySTM in reads-validated/s

Data from the 3-d Figure 4.5 (left). CPU validates normally, GPU validates its chunk with coalesced memory access.
A.5 Intermediate results - multi-threaded environment

A.5.1 Best assignment of validator threads

Figure A.9: Validators, RND array walk

Figure A.10: Validators, SEQ array walk

reads-validated/s normalized to TinySTM-untouched
A.5.2 Array traversal program - time % spent in validation

Figure A.11: Disjoint arrays - validation time proportion

- 1 STM threads
- 2 STM threads
- 4 STM threads
- 8 STM threads

No contention among threads (validate whole read-set).

Figure A.12: Conjoint array - validation time proportion

- 1 STM threads
- 2 STM threads
- 4 STM threads
- 8 STM threads

Moderate contention among threads (occasional invalidations).