Enhancing GPU Programmability and Correctness Through Transactional Execution

A DISSERTATION

SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL

OF THE UNIVERSITY OF MINNESOTA

BY

Anup Purushottam Holey

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

Doctor of Philosophy

Antonia Zhai

January, 2015

Acknowledgements

I joined the U of M as a Master's student in Electrical Engineering six years back. My journey until here is been filled with memorable moments, and it would not have been possible without support from a number of people. First and foremost, I would like to sincerely thank my adviser, Professor Antonia Zhai, for giving me the opportunity to pursue research in the field of computer architecture. Professor Zhai's guidance has been instrumental in reaching this goal.

Next, I would like to thank my committee members, Professors Pen-Chung Yew, Kia Bazargan, and George Karypis, for their insightful comments, which have improved the quality of my dissertation. I have also learned from Professor Yew how to evaluate architecture problems from multiple perspectives. Discussions with him during our weekly group meeting were always thought-provoking.

Throughout the graduate studies, I took several courses and I have greatly enjoyed them. I would like to thank all the professors who have offered them and took efforts to make the courses challenging as well as engaging. In particular, I extend my sincerest thanks to Professors Keshab Parhi and Ravi Janardan for their guidance in and out of the classroom. I would also like to thank Professor Wei-Chung Hsu for his enthusiasm and inspiration during the architecture and compiler classes.

I am grateful to the Computer Science department and its office staff for their support all the way through this journey. The systems staff in the department has also been exceptional. Many times they have gone out of their way to solve the system related issues

and made sure that none of my deadlines were affected. Thank you everyone!

I am thankful to my manager, Steve Tsai, and the team at Intel Corporation, who have been cooperative during the last phase of my Ph.D. Special thanks to Michael Cole for his help in improving my dissertation draft.

I was fortunate to share the lab with encouraging labmates during my stay at the U. Thank you Vineeth Mekkat, Ragavendra Natrajan, and Jieming Yin for all the support. I would like to thank Adam Bailey for his contributions to GPU benchmark development for the transactional memory work. Also thanks to Sanyam Mehta, Guojin He, and Yangchun Luo for lending a hand on various occasions.

A few dear friends have been part of my life for the past six years. They have made this journey exciting and memorable — thank you Vinit Padhye, Pratap Tokekar, Vineet Bhatawadekar, Neha Kulkarni, Shruti Patil, and Ram Varma. Thanks to Jitendra Patil, Pankaj Patil, and Tejas Joshi for their support and encouragement.

Finally and most importantly, I am indebted to my wonderful and loving family — my parents and my sister — who always believed in me and stood by me in all the endeavors. Without their firm support and love, I could not have achieved this milestone. Thank you Aai, Baba, and Amruta for everything.

Dedication

To my beloved family — Amruta, Aai, and Baba.

Abstract

Graphics Processing Units (GPUs) are becoming increasingly popular not only across various scientific communities, but also as integrated data-parallel accelerators on existing multicore processors. Support for massive fine-grained parallelism in contemporary GPUs provides a tremendous amount of computing power. GPUs support thousands of lightweight threads to deliver high computational throughput. Popularity of GPUs is facilitated by easy-to-adopt programming models such as CUDA and OpenCL that aim to ease programmers' efforts while developing parallel GPU applications. However, designing and implementing correct and efficient GPU programs is still challenging since programmers must consider interaction between thousands of parallel threads. Therefore, addressing these challenges is essential for improving programmers' productivity as well as software reliability. Towards this end, this dissertation proposes mechanisms for improving programmability of irregular applications and ensuring correctness of compute kernels.

Some applications possess abundant data-level parallelism, but are unable to take advantage of GPU's parallelism. They exhibit irregular memory access patterns to the shared data structures. Programming such applications on GPUs requires synchronization mechanisms such as locks, which significantly increase the programming complexity. Coarse-grained locking, where a single lock controls all the shared resources, although reduces programming efforts, can substantially serialize GPU threads. On the other hand, fine-grained locking, where each data element is protected by an independent lock, although facilitates maximum parallelism, requires significant programming efforts. To overcome these challenges, we propose transactional memory (TM) on GPU that is able to achieve performance comparable to fine-grained locking, while requiring minimal programming efforts. Transactional execution can incur runtime overheads due to activities such as detecting conflicts across thousands of GPU threads and managing a consistent memory

state. Thus, in this dissertation we illustrate lightweight TM designs that are capable of scaling to a large number of GPU threads. In our system, programmers simply mark the critical sections in the applications, and the underlying TM support is able to achieve performance comparable to fine-grained locking.

Ensuring functional correctness on GPUs that are capable of supporting thousands of concurrent threads is crucial for achieving high performance. However, GPUs provide relatively little guarantee with respect to the coherence and consistency of the memory system. Thus, they are prone to a multitude of concurrency bugs related to inconsistent memory states. Many such bugs manifest as some form of data race condition at runtime. It is critical to identify such race conditions, and mechanisms that aid their detection at runtime can form the basis for powerful tools for enhancing GPU software correctness. However, relatively little attention has been given to explore such runtime monitors. Most prior works focus on the software-based approaches that incur significant overhead. We believe that minimal hardware support can enable efficient data race detection for GPUs. In this dissertation, we propose a hardware-accelerated data race detection mechanism for efficient and accurate data race detection in GPUs. Our evaluation shows that the proposed mechanism can accurately detect data race bugs in GPU programs with moderate runtime overheads.

Contents

Acknowledgements **i**

Dedication **iii**

Abstract **iv**

List of Tables **x**

List of Figures **xi**

1 Introduction **1**

1.1 Challenges Addressed in Dissertation 2

1.1.1 Improving Programmability of Irregular Applications 3

1.1.2 Improving Correctness of GPU Applications 4

1.2 Dissertation Contributions 5

1.3 Dissertation Outline 6

2 Fundamentals of GPU **8**

2.1 GPU Architecture 8

2.2 Workloads 10

3 Programming Challenges with Irregular Applications on GPUs **11**

3.1 Programming Complexities in GPUs . . . 12
3.1.1 Challenges with Locking in GPU . . . 12
3.1.2 Challenges with Memory Consistency in GPU . . . 14
3.2 Transactional Execution in GPU . . . 15

4 Software Transactional Memory on GPUs 18
4.1 STM Framework on GPU . . . 19
4.1.1 Metadata Management . . . 21
4.1.2 Eager Read-Write Conflict Detection STM (ESTM) . . . 23
4.1.3 Pessimistic Conflict Detection STM (PSTM) . . . 26
4.1.4 Invisible Read STM (ISTM) . . . 27
4.1.5 Ensuring Memory Consistency . . . 28
4.1.6 Correctness of Transactional Memory . . . 29
4.2 Experimental Setup . . . 29
4.2.1 Benchmarks . . . 30
4.3 Evaluation . . . 30
4.3.1 Performance Analysis . . . 31
4.3.2 STM Scalability . . . 34
4.3.3 Memory Space Overhead for STM . . . 37
4.3.4 Selecting an Optimal STM Design . . . 38
4.3.5 Comparison with CPU . . . 38
4.4 Related Works . . . 39
4.5 Summary . . . 41

5 Hardware Transactional Memory on GPUs 43
5.1 Transactional Execution on GPUs . . . 44
5.2 Hardware Support . . . 47
5.2.1 Conflict Detection . . . 48

5.2.2 Version Management . . . 51
5.2.3 Bandwidth Optimization . . . 54
5.2.4 Hardware Complexity . . . 55
5.3 Infrastructure . . . 56
5.3.1 Benchmarks . . . 57
5.4 Performance Evaluation . . . 60
5.4.1 M-bit Granularity Impact . . . 64
5.4.2 Scalability Study . . . 65
5.4.3 Bandwidth Utilization . . . 66
5.5 Related Works . . . 67
5.6 Summary . . . 69

6 Data Race Detection in GPUs 71
6.1 Data Races in GPUs: Case Studies . . . 73
6.1.1 Lack of Correct Synchronization . . . 73
6.1.2 Incorrect Use of Locks . . . 74
6.2 Data Race Detection Framework . . . 76
6.2.1 Races Between Barrier Synchronizations . . . 77
6.2.2 Races in Critical Sections . . . 80
6.2.3 Races Due to Improper Memory Fencing . . . 82
6.3 HAccRG Implementation . . . 83
6.3.1 Data Race Detection in Shared Memory . . . 84
6.3.2 Data Race Detection in Global Memory . . . 85
6.3.3 Accuracy Trade-offs in HAccRG . . . 89
6.4 Evaluation Methodology . . . 90
6.5 Experimental Results . . . 92
6.5.1 Effectiveness of Data Race Detection . . . 92
6.5.2 Performance Impact of Race Detection . . . 96

6.5.3 Overheads in HAccRG 100
6.6 Related Works 103
6.7 Summary 105

7 Conclusions and Future Directions **107**
7.1 Future Work 109
7.1.1 Hardware-Software Co-Design of Transactional Memory 109
7.1.2 Compiler Support for Transactional Memory 110
7.1.3 Unified Transactional Memory and Data Race Detection Support . . 110

References **111**

List of Tables

4.1 Benchmarks for STM evaluation 31

5.1 Architectural configuration of GPU for HTM evaluation 57

5.2 Architectural configuration of CPU for HTM evaluation 58

5.3 Benchmarks for HTM evaluation 58

6.1 Hardware configuration for evaluating HAccRG. 90

6.2 Benchmarks used for HAccRG evaluation 91

6.3 Effect of tracking granularity on data race detection 94

6.4 Global memory overhead of HAccRG 103

List of Figures

3.1 Accessing hash table data structure in GPU 13
3.2 Lock-based code in GPU 14
3.3 Effect of weak memory consistency in GPU 15
3.4 Transactional execution model in GPU 16
3.5 Transactional memory API 16
4.1 Transaction descriptor 22
4.2 Performance achieved through STM on GPU 31
4.3 Performance of ESTM, PSTM, and ISTM designs 32
4.4 Execution time breakdown of STMs 33
4.5 Abort/commit ratio of STM designs on GPU 34
4.6 Scalability study of HASH-S benchmark 35
4.7 Scalability study of BANK benchmark 36
4.8 Effect of tracking granularity on STM performance 37
4.9 Performance comparison of STM with CPU 39
4.10 Scalability of STMs with thread count 40
5.1 Hardware transactional execution in GPUs 46
5.2 Hardware support for HTM on GPUs 48
5.3 Software support for HTM on GPUs 49
5.4 IPC achieved for HTM on GPU 60
5.5 Transaction failure rate in HTM 62

5.6 Performance of HTM on GPU 63
5.7 Impact of M-bit granularity on HTM 64
5.8 HTM scalability on GPU 66
5.9 DRAM traffic distribution with HTM 68
6.1 Data races caused by lack of synchronization 73
6.2 Data races caused by incorrect use of locks 75
6.3 Data race detection process 79
6.4 Detecting memory fence races in HAccRG 82
6.5 Hardware support for shared memory data race detection 84
6.6 Hardware support for global memory data race detection 86
6.7 Impact of tracking only most recent accesses in HAccRG 95
6.8 Performance impact of HAccRG 97
6.9 Performance comparison of HAccRG with its software implementation . . . 98
6.10 Impact of software shared memory shadow entries on performance 100
6.11 DRAM bandwidth utilization in HAccRG 101

Chapter 1

Introduction

Computers have made their way into most aspects for our lives, ranging from air conditioning and security systems at our homes, driver assistance in cars to high performance smartphones and desktop computers. Scientists rely on supercomputers for solving complex problems in weather forecasting, remote sensing, fluid dynamics, and in many more fields. The big data centers of internet giants, such as google, facebook, and amazon, are powered by some of the most powerful microprocessors available in the world. Computing needs in these applications vary greatly based on the nature of task, the cost, and also, the energy budget. To cater to such diverse applications, microprocessors have evolved over the years. Specialized architectures for digital signal processing and motion sensing can be found in a large number of consumer products. PC gaming is one such domain which has peculiar computing needs for processing large amount data quickly. Graphics Processing Units (GPUs), designed for such applications, have gained significant popularity in the gaming industry as the demand for superior visual experience grew over the years.

Programmers have been hacking GPUs to perform non-graphics tasks through graphics application programming interfaces (APIs) by reshaping the tasks to resemble graphics workloads. However, exploiting this potential requires a comprehensive understanding of GPU pipelines and graphics APIs. This presents a steep learning curve, and hence, it is

not very popular since programmers are not able to benefit from GPU's computing power. To address this problem, Nvidia introduced CUDA [56], a parallel computing platform, in 2007 that exposes general-purpose compute capabilities of their GPUs to programmers. CUDA provides a set of APIs and a programming model that resembles C, which can be easily understood and adopted. As CUDA gained in popularity, a framework called OpenCL [36] emerged in 2009. OpenCL programs can execute on heterogeneous systems consisting of CPUs, GPUs, and other specialized processors. CUDA and OpenCL have evolved over the last few years, and both allow programming of data-parallel applications on GPUs with minimal efforts. These easy-to-adopt programming models have fueled the growth of general-purpose GPUs (GPGPU) in a wide range of applications, beyond just graphics. GPUs have made their way into today's data centers as well as supercomputers because of their energy efficiency and ability to process large data. With such growing popularity, it is crucial to identify and address the challenges in GPU programmability in order to reduce software development time as well as to improve programmers' productivity and software reliability.

1.1 Challenges Addressed in Dissertation

This dissertation explores the challenges in GPGPU programming from two perspectives: programmability and correctness.

- With the help of the CUDA or OpenCL programming models, GPUs have become easy to program; however, their scope has been limited to a certain class of data-parallel applications. There are still a large number of data-parallel applications that are difficult to program on GPUs. In this dissertation, we will investigate techniques for improving their programmability.

- With the ability to run thousands of concurrent threads, it is challenging to write

kernels while maintaining legitimate data accesses from all threads. Incorrect programming practices or human errors can introduce issues such as data races, which are difficult to debug and resolve. In this dissertation, we will identify elements affecting the correctness of GPU applications and propose solutions to detect them.

A brief overview of programmability and correctness challenges and their proposed solutions is given next.

1.1.1 Improving Programmability of Irregular Applications

GPUs have most commonly been adopted for extracting data-level parallelism from applications having regular memory access patterns. However, a large number of applications, although possessing data-level parallelism, are not able to exploit GPU's potential effectively. In these applications, threads share data dynamically and exhibit irregular memory access patterns. For example, in many graph-based applications [12, 38], where every node is connected to a few other nodes in the graph, threads exhibit dynamic data sharing while working in parallel on different nodes of the graph. These applications often require synchronization mechanisms between threads, thus requiring significant programming efforts and causing performance degradation during execution.

To ensure functional correctness in an application, shared data must be protected by locks. Coarse-grained locking (CGL) uses a single global lock to serialize accesses to the shared data, but also causes performance degradation. On the other hand, fine-grained locking (FGL) maximizes parallelism; however, it can increase programming complexity. These issues exacerbate on GPUs that are capable of running thousands of concurrent threads. Writing FGL code becomes more challenging, while performance degradation caused by CGL can become exorbitant, offsetting the speedup achieved by GPU.

Apart from the shared data protection, the weak memory-consistency models in GPUs also pose additional programming difficulties for shared data applications. Most modern processors support weak memory consistency because of the improved performance. In

such systems, memory accesses can complete in non-deterministic order and break the intuitive sequential consistency model often assumed by the programmer. Therefore, the programmers must manually insert memory fence instructions to impose ordering in memory accesses, which can be error prone. This makes writing and debugging shared data applications on GPUs even more difficult.

Transactional memory (TM) can alleviate the GPU programming challenges discussed above. In this dissertation, we explore both software and hardware implementations of transactional memory support for GPUs. Transactional execution abstracts away the complexities associated with the lock-based programming. In this execution model, atomic sections protected by locks are treated as transactions and are executed concurrently with other transactions. Such support can be implemented in software (STM) [26, 64, 67, 18, 17, 15, 49, 68, 28, 9, 75] as well as in hardware (HTM) [30, 62, 76, 22]. Although hardware support is faster, it requires changes in the GPU architecture. Software support, on the other hand, can be implemented easily on commodity hardware.

1.1.2 Improving Correctness of GPU Applications

Designing and implementing correct and efficient GPU programs remains a challenge since programmers must consider interaction between thousands of parallel threads. Improper synchronizations between a large number of threads can lead to data races during program execution. A data race occurs when more than one thread simultaneously accesses the same memory location, and at least one of the accesses is a write. Being able to detect these data races at runtime can facilitate the construction of powerful tools to improve the reliability of GPU applications.

While a large body of work exists on detecting data races between CPU threads [51, 52, 59, 60], these techniques cannot be directly extended to GPU threads for both correctness and performance reasons. From a correctness perspective, the causes of data races are multifaceted in contemporary GPUs. Not only do improperly placed synchronizations and

critical sections cause data races, but the lack of memory coherence support also introduces data races that otherwise would not occur. Most CPU-based multicore systems are coherent and are not subject to the latter. From a performance perspective, many proposed solutions require mechanisms for tracking memory accesses per-thread to detect data races between CPU threads. For modern GPUs that are capable of executing thousands of threads simultaneously, tracking per-thread accesses poses challenges both in terms of performance as well as hardware overhead. Thus, providing an efficient and scalable data race detection mechanism for the GPU becomes a significant challenge.

There have been several recent efforts addressing software correctness in GPUs [41, 44], including data race detection [7, 34, 42, 43, 77, 78]. These works detect data races statically or at runtime by instrumenting GPU applications. Instrumentation introduces significant performance degradation, rendering such techniques inefficient. Adequate hardware support can potentially improve the effectiveness and efficiency of runtime data race detection for GPUs. In this dissertation, we design a hardware-accelerated data race detection framework for modern GPUs, referred to as HAccRG. This hardware support is responsible for tracking and comparing the memory accesses from all threads in order to detect data races.

A straightforward implementation of data race detection requires pairwise comparisons of all memory traces from all threads. The number of comparisons is quadratic relative to the number of threads. Scaling this implementation on GPUs that typically support thousands of threads is impractical. Therefore, HAccRG employs a per-memory access tracking mechanism, instead of tracking per-thread accesses. This method allows us to design a fast hardware data race detector with moderate hardware overhead.

1.2 Dissertation Contributions

This dissertation makes the following key contributions in improving the programmability and correctness aspects in GPUs.

1. We propose transactional execution as an alternative programming paradigm for developing irregular applications on GPUs with minimal efforts. We propose both software and hardware techniques for facilitating the transactional semantics. For software transactions, we provide multiple flavors that suit applications with different characteristics. Furthermore, a scalable hardware acceleration for transactions is also proposed. We present designs of our techniques and demonstrate their feasibility by implementing a CUDA-based STM framework as well as by extending a cycle-accurate GPU simulator.

2. We propose a low overhead hardware-accelerated data race detection framework for GPUs, referred to as HAccRG. We present the design and implementation of HAccRG, and evaluate its performance through simulations on a cycle-accurate GPU simulator. Furthermore, we show that the software implementation of our proposed mechanism outperforms a previously proposed software data race detector for GPUs.

1.3 Dissertation Outline

The rest of this dissertation is organized as follows:

- Chapter 2 provides an overview of GPU architectural features which are unique to GPUs and have been considered while designing the transactional execution and data race detection supports. We also briefly discuss the workloads that can benefit from GPU's thread-level parallelism.

- Chapter 3 addresses the issues in parallelizing irregular applications on GPUs, and proposes transactional execution for making their programming easier and faster.

- Chapter 4 presents the design and evaluation of lightweight software transactional memory on GPUs.

- Chapter 5 presents the design and evaluation of hardware transactional memory on GPUs.
- Chapter 6 addresses the causes of concurrency bugs found in GPU kernels, and further presents the design and evaluation of HAccRG, a framework for detecting data races in GPUs.
- Chapter 7 concludes this dissertation and outlines future research directions.

Chapter 2

Fundamentals of GPU

In this chapter, we present the details of the GPU architecture and workloads that can take advantage of this architecture.

2.1 GPU Architecture

An application running on the CPU launches a highly multithreaded kernel onto the GPU. The kernels for the GPU are written using the CUDA programming model for Nvidia GPUs or using OpenCL for GPUs from other vendors. The GPU compiler translates CUDA or OpenCL code into GPU-specific instructions. The GPU has access to the device memory which resides on the GPU card. The device memory is stored in a specially designed high-bandwidth GDDR memory for GPUs. The communication between CPU's main memory and device memory is carried out through a PCI-Express interface. In processors where both the CPU and GPU are integrated onto the same die, the device memory is a part of the CPU's main memory.

GPUs support thousands of threads executing in parallel. Such computing power is provided by an array of computing cores, often referred to as streaming multiprocessors (SM) in CUDA [56] or compute units (CU) in OpenCL [36]. Each SM consists of an array

of simple in-order cores that are referred to as streaming processors (SP) or processing elements (PE). SPs located within a single SM execute the same instruction but operate on different data in a given cycle, an execution model known as single instruction multiple data (SIMD). Nvidia refers to their CUDA thread execution model as single instruction multiple threads (SIMT) [56] because of hardware threading.

Work is allocated to the GPU as kernels that contain a large number of threads. Threads within the same kernel are organized into blocks [56] or work-groups [36], and blocks are mapped onto different SMs. At execution time, threads within the same thread-block are further partitioned into warps [56] or wavefronts [36]. The size of the warp varies with GPU vendor. Nvidia GPUs have 32 threads in a warp while AMD GPUs have, typically, 64 threads in a wavefront. Threads within the same warp are scheduled to the SIMD computing engine simultaneously, and thus are executed in lockstep. Threads across different warps are executed asynchronously. The GPU contains an on-chip memory in each SM, explicitly managed by a programmer, referred to as the shared memory. The shared memory is banked [56] to enhance throughput and has very low access latency. If threads within a warp access different banks, all the accesses are served in parallel; otherwise, multiple accesses to the same bank are serialized. Global memory, a part of the off-chip device memory, is accessible to all threads in a GPU kernel. The device memory has high access latency, and hence its accesses are expensive. Local memory is private to individual threads, although, it also resides in the off-chip device memory [56, 36]. Consecutive accesses to both global and local memory from different threads in a warp are coalesced, i.e., combined into a single larger access to compensate for higher memory access latency [56]. The latest GPUs also support caching. Global and local memory are cached in per-SM non-coherent L1 data caches and a coherent shared unified L2 cache. Since global memory is accessible to all threads in a GPU and L1 caches are non-coherent, global memory writes to L1 data cache are written through to the corresponding L2 cache banks [56]. Furthermore, GPUs have read-only texture and constant L1 caches which can be

utilized for storing read-only data for faster access. GPUs also support atomic operations in hardware that are used to implement critical sections in GPU kernels. CUDA and OpenCL support atomic operations in the shared and global memory spaces [56, 36].

2.2 Workloads

GPUs are throughput-driven highly multithreaded processors with low single-threaded performance. Therefore, applications that have large data-level parallelism are suitable for achieving high performance on GPUs. Many scientific applications including image processing, data mining, biomolecular simulation, fluid dynamics, economics, astronomy, and graph solving can benefit from GPUs. Several GPU benchmark suites [71, 12, 24, 8], released recently for studying and evaluating GPU architectures, have collections of applications from many such fields. Nvidia and AMD have also released various prototypes of scientific applications in CUDA and OpenCL for benchmarking purposes. Many individual contributors have also converted numerous CPU applications to the GPU domain. All these benchmarks fall under the category of throughput computing, in which performance gains are achieved through massive fine-grained parallelism. The majority of the GPU workloads available to programmers today are fully-parallel applications which have regular memory access patterns. Such applications are ideal for running on GPUs while extracting maximum parallelism. However, recent works have also proposed techniques for improving performance of irregular applications on GPUs [54, 32].

Chapter 3

Addressing Programming Challenges with Irregular Applications on GPUs with Transactional Execution

Graphics Processing Units (GPUs) have been widely adopted by researchers for extracting data-level parallelism from diverse applications. However, an important class of applications, although possesses data-level parallelism, shares data dynamically and exhibits irregular access patterns. These applications often require synchronization mechanisms that increase the programming efforts. Since GPUs have traditionally been used for extracting parallelism from fully-parallel applications, utilizing them for applications with dynamic data sharing remains a challenge due to the sheer number of concurrent threads GPU supports. In this work, we investigate the techniques for improving the programmability of such applications on GPUs.

We begin with a discussion on challenges in GPU programming with examples of irregular applications, and also explain how the transactional memory execution model is able to alleviate the programming complexities.

3.1 Programming Complexities in GPUs

In order to avoid data races in GPU applications that share data dynamically, accesses from multiple threads must be protected by locks. The challenges with lock-based programming exacerbate with a large number of GPU threads and synchronous execution of threads in a warp. In the rest of this section, we will demonstrate these challenges on GPUs.

3.1.1 Challenges with Locking in GPU

Locking is a commonly used mechanism for ensuring atomic accesses to shared data, however, the improper usage of locks can cause livelocks/deadlocks and/or performance degradation. In GPU, due to the fact that threads allocated within the same warp/wavefront are executed in lock steps, improper usage of locks is more prone to livelocks or deadlocks.

Consider the hash table data structure shown in Figure 3.1, where all threads are attempting to insert and delete elements from the table as a unit. I.e., either both insert and delete operations are completed or none. When multiple threads are modifying the hash table simultaneously, each thread must lock the corresponding hash buckets before modifying it. However, if multiple threads attempt to access the same buckets in a different order, a deadlock can occur. Consider a scenario where thread T_1 is inserting an element to hash bucket B_P and removing an element from hash bucket B_Q, while thread T_2 is inserting to bucket B_Q and removing from bucket B_P. If T_1 locks the bucket B_P and T_2 locks the bucket B_Q, neither thread can make progress since they are both waiting for the other thread to release the resources. If they both choose to relinquish the ownership of the locks and retry, they can enter into a livelock by repeating the same sequence of lock acquisition. In particular, if threads T_1 and T_2 happen to be in the same warp, their lock acquire and

release operations will always be in sync, leading to a livelock. The simplest solution to this problem is using a single global lock to lock the entire hash table, thus serializing operations from all threads. In a GPU supporting thousands of threads, serializing accesses from all threads has non-trivial performance overheads. On the other hand, employing a separate lock for each bucket to facilitate forward progress and functional correctness can become a significant programming challenge.

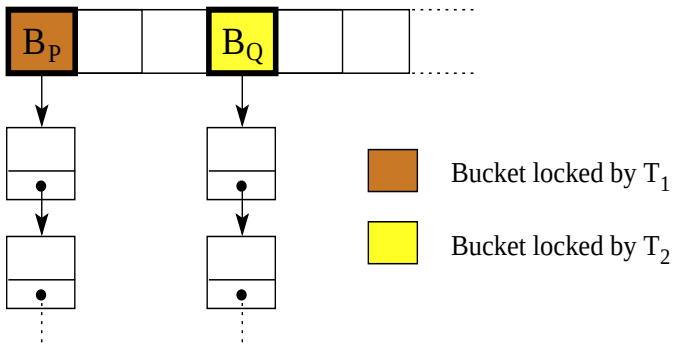


Figure 3.1: hash table data structure with B_P and B_Q hash buckets shown. Thread T_1 is trying to insert an element into bucket B_P and remove an element from bucket B_Q, while thread T_2 is trying to insert into bucket B_Q and remove from bucket B_P. A deadlock occurs when T_1 locks bucket B_P and T_2 locks bucket B_Q.

Lock-step execution of threads in a warp/wavefront can lead to deadlocks in a GPU that would not occur in CPU-like threads [63]. Consider the code snippet shown in Figure 3.2(a). When all threads within the same wrap/wavefront execute the same compare-and-swap (`CAS`) instruction in line `2`, only one thread will succeed. The successful thread waits for other threads to converge in line `3`, while others spin in line `2`. Thus, no thread is able to make progress and the execution enters a deadlock. This phenomenon would not have occurred on CPUs, because CPU threads can make progress asynchronously. To achieve the desired behavior, the `CAS` instruction must be moved into an *if* clause within the spin loop as shown in Figure 3.2(b).

```
1.  ...
2.  while(CAS(lock, 0, 1));
3.  /* critical section */
4.  lock = 0;
5.  ...
```

(a) Deadlock happens on GPU when threads in a warp attempt to acquire the same lock variable.

```
1.  bool leaveloop = false;
2.  while(!leaveloop)) {
3.    if(!CAS(lock, 0, 1) {
4.      /* critical section */
5.      leaveloop = true;
6.      lock = 0;
7.    }
8.  }
```

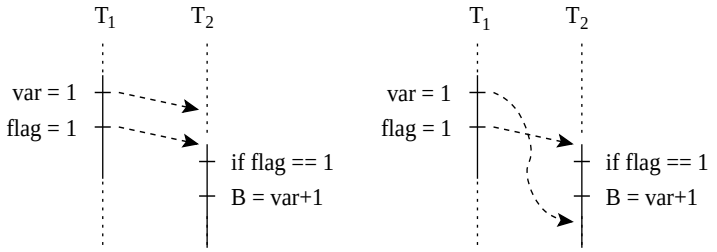
(b) Deadlock-free code on GPU.

Figure 3.2: Locking in GPU.

3.1.2 Challenges with Memory Consistency in GPU

The lack of support for a sequential memory consistency model in modern GPUs can create significant hurdles for programmers to achieve the desired behavior. Figure 3.3 demonstrates the discrepancies in memory access ordering due to the lack of sequential consistency in GPUs. Two threads T_1 and T_2 both access the `flag` and `var` variables. Both variables are initialized to 0 at the beginning of the program. Thread T_1 writes to `var` and then sets the `flag` to 1; thread T_2 tests the value of `flag` and accesses `var` when `flag` is set to 1. However, without support for a sequential consistency model, there is no guarantee that the two writes from thread T_1 will be seen by thread T_2 in the same order as the program order in T_1. In Figure 3.3(a), the two writes are seen by thread T_2 in the program order, thus program achieves the desired behavior. However, in Figure 3.3(b), the two writes complete out of order, and the program behaves incorrectly. This problem can be solved by inserting memory fence in the program before the `flag` is set to 1. A memory fence ensures that accesses prior to the fence instruction are complete before the program continues its execution. Therefore, when thread T_1 sets the `flag` to 1, the write to `var` is guaranteed to be complete.

The challenges discussed above in programming shared data applications on GPU can be abstracted away by transactional execution support. Ensuring forward progress and functional correctness of transactions is taken care by the transactional memory.



(a) Correct execution. (b) Incorrect execution.

Figure 3.3: Effect of weak memory consistency in GPU. Variable `var` and `flag` both are initially set to 0. Dotted arrows show when the memory accesses are complete.

3.2 Transactional Execution in GPU

In the GPU execution environment, we propose the transactional memory (TM) support that allows the programmers to convert segments of a thread into transactions. The underlying support for TM is responsible for tracking memory accesses from all threads to detect dependence violations and to support thread abort and retry. TM ensures atomicity (transactions either succeed or fail as an entity), consistency (transaction always moves from one consistent memory state to another consistent state), and isolation (modifications made by an uncommitted transaction are not visible outside that transaction) for all threads. Figure 3.4 illustrates the TM execution model with multiple threads executing transactions in parallel. As part of a transaction, thread T_x from warp 0 speculatively reads from the address A and writes to the address B; while thread T_y from warp 1 attempts to read address B. These concurrent memory accesses create a dependence violation, thus thread T_y aborts and retries. During the retry T_y succeeds as thread T_x has committed its transaction. Next, we will demonstrate how transactional memory support can facilitate the design and development of parallel programs in GPU, where concurrent threads must access shared data.

A programmer can exercise transactional execution support by simply marking the section of code that has to be executed atomically. A compiler then translates the marked

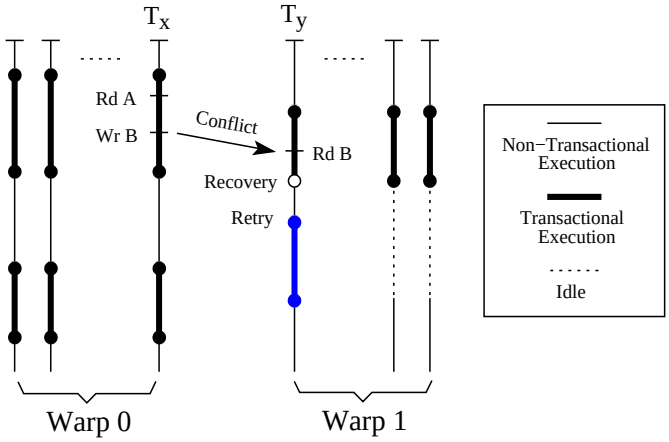


Figure 3.4: Transactional execution model in GPU. Thread T_y from warp 1 conflicts with thread T_x from warp 0. When thread T_y retries upon detecting a conflict, other threads in the warp 1 remain idle due to divergence. The idle threads have successfully committed their transactions.

code into transaction. For each memory access in the marked code, the compiler adds instructions to track the memory accesses, detect dependence violations between threads, and maintain a consistent memory state. In a lock-based program, such section of code would be protected by locks. In programs that do not use explicit locks for sharing data, such as the one shown in Figure 3.3, a programmer will have to identify and mark the critical sections. For the example shown in Figure 3.3, two transactions can be marked for instructions executed by threads T_1 and T_2, as shown in Figure 3.5.

```
TX_START          TX_START
  var = 1;          if flag == 1
  flag = 1;            B = var + 1;
TX_END            TX_END
```

Figure 3.5: Markers TX_START and TX_END define the transactions for execution shown in Figure 3.3.

Let us see how transactional execution solves the problems discussed in Figures 3.1 and 3.3. For the hash table example shown in Figure 3.1, the insert and remove operations

performed together can be formed as a transaction. Hence, when two threads attempt to access the same buckets simultaneously, one of the transaction fails while the other succeeds. The failed transaction then recovers and retries, thus serializing the conflicting accesses to the same buckets. For the example in Figure 3.3, two transactions are formed by combining accesses to variables `var` and `flag`, as shown in Figure 3.5. If the thread T_2 executes its transaction without a conflict with thread T_1, it is guaranteed that the two writes made by thread T_1 are complete, thus resulting in correct value of variable `B`.

Chapter 4

Software Transactional Memory on GPUs

This chapter presents the design and evaluation of software transactional memory (STM) support on GPUs. We propose this support as a compile-time library, which generates a binary with transactional constructs. To enable this support, a programmer identifies and marks critical sections in the code that must be executed atomically. The compiler then replaces the memory accesses within the marked critical sections with transactional reads and writes. It also inserts operations to start, commit, and retry transactions. The produced binary provides performance comparable to fined-grained locking (FGL), while limiting the programming efforts to that of coarse-grained locking (CGL).

Achieving high performance using transactional execution is a challenge due to overheads associated with transaction management. The challenges include overheads in tracking dependences and detecting dependence violations across thousands of GPU threads at runtime, buffering data before a transaction successfully commits, and re-executing aborted transactions. These challenges have been addressed by an extensive body of work in CPU-based systems that run a small number of powerful threads [1, 17, 26, 29, 48, 64, 69]. Many STMs on CPU take advantage of features of object-oriented programming languages, such

as Java, to improve their performance. On the other hand, GPU executes thousands of lightweight threads and has a C-like programming language support. Thus, transaction management can have non-trivial performance impact on the GPU applications.

Previous works [9, 75, 74] have shown that it is feasible to provide STM support on GPU, while reducing programming complexity. However, these techniques show less performance improvements with large number of GPU threads due to high overheads involved in the transaction management, particularly in conflict detection. In this work, we illustrate mechanisms for providing lightweight software transaction support that is able to scale to thousands of GPU threads. We also identify opportunities for optimizing conflict detection operations within transactions, thus reducing the overhead of transactional execution. To this end, we propose different flavors of transaction supports that are suitable for applications with different performance characteristics. The proposed techniques can even outperform FGL under high lock contention on GPU. Our comparison with earlier STM work on GPU reveals that low overhead transactions provide scalability on the GPU-based systems. Overall, we demonstrate that despite inherent overheads in the STM on GPU, it provides significantly higher performance than the CPU-based execution.

The rest of this chapter is organized as follows: In Section 4.1, we discuss the design of lightweight software transactions on GPUs. Section 4.2 presents our experimental infrastructure and the benchmarks used in evaluation. In Section 4.3, we evaluate the performance of the proposed STM designs. Finally, we compare the proposed STMs with previous works in Section 4.4.

4.1 STM Framework on GPU

Providing efficient software transactional memory support on GPU is challenging because of large number of lightweight threads GPU supports, unlike few powerful threads on CPU. Such lightweight threads could incur significant performance overheads with the transaction management tasks. Therefore, to scale STM support to thousands of GPU threads, the

transactions must be very lightweight. In the rest of this chapter, we discuss how to design low overhead transactions on GPUs.

Two key design issues for supporting STM are: managing speculative modifications to ensure a consistent view of the memory state from different threads and detecting dependence violations between different transactions. We refer to the former as *version management* and the latter as *conflict detection*. Both version management and conflict detection can be performed at the time of memory access or when the transaction commits. We refer to the former as *eager* and the latter as *lazy*. Eager version management performs well when conflict detection rate is low, while lazy version management is more effective in high contention workloads [27]. We choose eager scheme for version management, which benefits applications with lower conflict rates. In eager version management, a transaction backs up the old data and performs in-place modifications in the memory. We will later show that our techniques perform better than other STM designs, that specifically target high-contention workloads, on applications with higher conflict rates.

Conflict detection also has significant impact on performance based on its eager or lazy scheme. Eager conflict detection detects violation early, thus, is able to reduce redundant work by the threads and free up computation and memory resources for other threads. Furthermore, it allows aborted transaction to retry early. On the other hand, lazy conflict detection defers the conflict detection until end of the transaction, thus reduces the abort rate and increases concurrency. The choice of conflict detection scheme also depends upon the type of version management. In eager version management, conflicts on writes must be detected eagerly before the memory state is updated. However, conflicts on reads can be detected either eagerly or lazily.

In this work, we present three STM designs with different conflict detection strategies for transactional reads. The first STM is our baseline design, which detects read conflicts eagerly. The second STM design specifically targets the applications where the same memory locations are read first and then written in the transactions. This design also detects

read conflicts eagerly, but it does not differentiate between transactional reads and writes. We will later show how this approach reduces the conflict detection overhead in STM. Our third STM design detects read conflicts lazily, thus making transactional reads faster. Note that all three STM designs detect conflicts on writes eagerly.

Ensuring Forward Progress: In eager conflict detection, an aborted transaction can abort others. Thus, it can happen that two transactions keep on aborting each other, leading to a livelock [6]. A backoff mechanism can avoid livelocks by adding a delay before restarting aborted transactions, thus ensuring forward progress. In addition, the contention mechanisms that have been proposed for TM on CPUs to avoid livelocks can also be adopted for GPUs [6]. It should be noted that, however, the STM designs proposed in this work do not cause deadlocks since the shadow memory entries are never held indefinitely by GPU threads, as explained later in this section.

We now begin with the description of metadata structures required to support STM on GPU.

4.1.1 Metadata Management

To support transactional execution, a GPU system must be extended to be able to track dependence violations between different transactional threads and to create backups of data modified during the transactions. A dependence violation occurs when two transactions access the same memory location and at least one of the accesses is a write. To perform these tasks, the following metadata is maintained by the underlying STM support:

- **Shadow memory** tracks, for each memory location, the access information for that location. A section of the memory is reserved as the shadow memory, and there is a one-to-one mapping between all memory locations and their shadow memory entries. For each memory access, the corresponding shadow entry is accessed to detect a conflict. Each shadow entry in our STM is 32 bits in size.

- **Read log** tracks, for each thread, the set of memory locations read by the current transaction.

- **Undo log** tracks, for each thread, the set of memory locations modified as well as the old values that were stored in those memory locations. Undo logs are used to restore the memory state upon transaction failures.

- **Transaction descriptor** maintains the state of the current transaction for each thread. The descriptor, shown in Figure 4.1, is initialized before the transaction begins. The read_log and undo_log are pointers to the thread's logs. The rd_count and wr_count, initialized to 0, indicate the number of reads and writes performed by the transaction. The transaction state can be `ACTIVE`, `ABORTED`, or `COMMITTED`.

Among these metadata, only shadow memory needs to be accessed from all GPU threads for conflict detection. Thus, it is maintained in the global memory address space, which is accessible to all threads. On the other hand, the read log, undo log, and transaction descriptor are thread-private, and thus are stored in the per-thread local memory. Accesses to consecutive memory locations from threads in a warp to the thread-private metadata are coalesced. Additionally, the local metadata can take advantage of the L1 cache of the GPU, and thus can be updated efficiently.

Next, we describe utilization of the metadata to perform various STM operations in GPU.

```
struct tx_descr {
   readlog_t *read_log;
   undolog_t *undo_log;
   unsigned rd_count;
   unsigned wr_count;
   state_t state;
};
```

Figure 4.1: Transaction descriptor.

STM Operations: For every marked transaction as shown in Figure 3.5, the compiler inserts various operations. The TX_begin() operation starts the transaction, while TX_commit() ends the transaction. TX_read() is performed for each read within the transaction, while TX_write() is performed for each write. Details of these operations are listed below:

- **TX_begin()** resets the rd_count and wr_count in the transaction descriptor to zero, and sets the transaction state to `ACTIVE`.
- **TX_read()** detects if the read access is conflicting with other transactions. If no dependence violation is detected, it performs the read, adds an entry to the read log, and increments the rd_count by one.
- **TX_write()** detects if the write access is conflicting with other transactions. If no dependence violation is detected, it adds an entry to the undo log, increments the wr_count by one, and performs the write.
- **TX_commit()** clears the shadow entries updated during the transaction by scanning the thread's read log and undo log. For an aborted transaction, the memory state is first recovered using the undo log before releasing the shadow entries. The aborted transactions retry again.

All the three proposed STM designs perform the aforementioned operations, but they differ in the way conflicts are detected in TX_read() and TX_write() operations. To detect conflicts, different information is tracked in the shadow memory entries for each STM. We now begin by describing our first STM design that detects read and write conflicts eagerly.

4.1.2 Eager Read-Write Conflict Detection STM (ESTM)

In ESTM, we detect conflicts eagerly by tracking both read and write information in the shadow entries. During transactional execution, multiple threads are allowed to read a

location, but only a single thread can perform a write. The fields in each shadow entry are motivated below:

- **n_reads** field (14 bits) indicates the number of speculative reads made to the given memory location, either by a single thread or by multiple threads. For every speculative read, the n_reads field is incremented by one.
- **tid** field (15 bits) stores the id of the thread that has first accessed the location. The 0 value of tid field indicates the location has not been speculatively read or modified by any thread (thread ids are maintained as >= 1). The tid field is required to identify if the thread accessing the location has modified the same location earlier. By having this field in the shadow entry, scanning of the undo log is avoided for ownership checks.
- **modified** bit (M) of 1 indicates the location has been speculatively written by the thread indicated by the tid field, while the 0 value indicates the thread tid has speculatively read the location.
- **shared** bit (S) of 1 indicates more than one thread has speculatively read the location. During a speculative read, if the M bit is 0 and the tid field is different from the reader thread's id, the S bit is set to 1.
- **lock** bit (L) of 1 indicates some thread is updating the shadow entry. If the L bit is 1, a thread waits until it is set to 0 before reading or writing the shadow entry. Once the shadow entry is updated, the L bit is set to 0 so that other threads can access the shadow entry. Setting of the L bit to 1 is always performed atomically.

A conflict between transactions is detected when one of the following conditions is observed:

- **RAW violation:** When a thread tries to speculatively read a location, the M bit is 1 and the thread's id does not match with the tid field in the shadow entry. It means

that some other thread has speculatively written to the location.

- **WAW violation:** When a thread tries to speculatively write a location, the M bit is 1 and the thread's id does not match with the tid field in the shadow entry.
- **WAR violation:** (i) When a thread tries to speculatively write a location, the M bit is 0 and its thread id does not match with the tid field. It means that some other thread has speculatively read the location. (ii) When a thread tries to speculatively write a location, the S bit is 1. It means that more than one thread has speculatively read the location.

If no conflict is detected, the thread adds an entry to its read log or undo log depending upon the type of access. To avoid multiple entries in the undo log for the same location written by a transaction, an entry is added to the undo log only on the first write to the given location, i.e., when the M bit changes from 0 to 1.

Once a thread reaches end of a transaction, it must release all the shadow entries to retry (if aborted) or to commit its updates. This is achieved by first releasing the shadow entries for the data read, followed by releasing the entries for data written. In case of an aborted transaction, the data is recovered using the undo log before the shadow entries are released. To release the shadow entries for data read, the thread atomically updates the corresponding entry for each address in the read log. More specifically, it decrements the n_reads field in the shadow entry by one, and if the new value of n_reads is zero, the thread resets all fields in the shadow entry to zero, thus clearing the read information completely. However, if the M bit is 1 it does not take any action since the write information is cleared later. Once the reads are released, writes are released by simply clearing all fields in the shadow entries for entries in the transaction's write set.

The ESTM design we discussed above can potentially be optimized for two reasons: (i) the read operations update the shadow entries, and hence, are slow. It is possible to make reads invisible to other transactions and detecting read conflicts just before the transaction

commits; and (ii) not differentiating between reads and writes during transactions allows faster STM operations, thus improving the STM performance. Next, we describe the second optimization over baseline ESTM.

4.1.3 Pessimistic Conflict Detection STM (PSTM)

In this STM design, speculative reads and writes are not treated differently, i.e., we pessimistically assume that if reads are conflicting, writes must also be conflicting. Therefore, in addition to regular read-write conflicts, a conflict can also be detected even if two threads read the same memory location. Here, we take advantage of the fact that transactions in some applications read and write the same locations. The applications that do not exhibit such access patterns will report false conflicts. However, not differentiating between reads and writes in PSTM allows much simpler conflict detection mechanism than ESTM. Its benefits are twofold — first, it can detect conflicts early in transactions where same locations are first read and then written, and second, conflict detection and commit overheads are lower because of simpler mechanisms.

Similar to ESTM, the shadow entries track which thread has accessed a given memory location. However, they only contain the id of the thread that has accessed the given memory location, since we do not differentiate between reads and writes in PSTM. A conflict is detected by a single compare-and-swap (CAS) atomic operation on the metadata as shown below:

```
uint ret_value = CAS(maddr, 0, thread_id);
```

Here, the *maddr* is the address where the shadow entry for the current access is stored. In this operation, the shadow entry is always replaced with the thread id if its value is 0. (Note that thread id is always non-zero). If the ret_value is 0 *or* is equal to the thread id, there is no conflict; otherwise, the transaction is aborted.

Once a thread aborts or successfully reaches end of the transaction, all the shadow entries are released by setting to 0 for entries in the read and undo logs. Note how the

shadow entry acquire and release operations in this design are much simpler than those in ESTM as the shadow entries contain only the thread id. In addition, since shadow entries can be owned by only one thread at a time, transactions do not have to wait for others to acquire or release the shadow entries as in the ESTM design.

4.1.4 Invisible Read STM (ISTM)

In the third STM design we present, speculative reads are invisible to other transactions, and are validated at the end before the transaction commits. Therefore, unlike ESTM or PSTM, reads are performed faster in ISTM because they do not modify the shadow entries. The shadow entries used for detecting dependence violations in ISTM are versioned locks [64, 28]. Versioned locks allow speculative reads to remain invisible to other transactions, while writes still remain visible. As a result, conflicts on writes are always detected eagerly, while those on reads can be detected either eagerly or lazily. Such approach enables increased concurrency between transactions as potentially conflicting transactions can still continue their execution.

A versioned lock contains the following fields:

- **version** indicates the most recent version number of the location. Successfully committed transactions increment the version numbers for the speculatively modified locations.

- **lock** bit indicates if the shadow entry is locked by a transaction for a speculative write. A thread trying to speculatively write a location sets the lock bit in the shadow entry to 1, and resets it to 0 during the commit operation.

A versioned lock is a 32-bit word with its least significant bit acting as the lock bit, while the remaining bits representing the version. This implementation of versioned lock is similar to the one proposed in previous STM design [9].

For each access, the shadow entry is read and one of the following actions is taken:

- If the lock bit is 1 in the shadow entry *and* the address has not been written earlier in the same transaction, the transaction aborts. Here, the second if condition avoids an abort if the thread is accessing its own modifications. If the lock bit is 1 and the thread has previously written to the location, no further action is required.

- For a read access if the lock bit is 0, the version number along with the address of read is stored in the thread's read log, which stores <addr, version> tuples.

- For a write access if the lock bit is 0, an atomic CAS operation is performed to set the lock bit to 1 in the versioned lock. If the return value of CAS has the lock bit set, the transaction aborts as another transaction has acquired the lock. Otherwise, an <addr, data, version> entry is added to the undo log, where *version* is extracted from the return value of CAS.

At the commit time, reads are validated to check if concurrent writes happened to the locations read during the execution. This is achieved by comparing the version numbers recorded at the time of read to the current version numbers of those locations. A change in version number, or the lock bit in the shadow entry is set and the address has not been written earlier in the same transaction indicates concurrent write operation, which causes the transaction to abort. If read validation is successful, the transaction commits by releasing locks for entries in the transaction's write set. Specifically, it increments the version number by one and sets the lock bit to 0 for the corresponding shadow entries. By incrementing the version number, transactions notify concurrent reads about the concurrent write. Aborted transactions recover the modified data using the undo log and then reset the lock bit to 0 for entries in their write sets.

4.1.5 Ensuring Memory Consistency

We need to insert memory fences in STM because of GPU's weak memory consistency to achieve functionally correct execution. In CUDA, support for fence is provided by

a *__threadfence* [56] instruction that ensures memory accesses prior to __threadfence are complete before continuing the execution. In our STMs, a fence is inserted at the beginning of the commit operation, which guarantees that speculative modifications are visible to other threads after the shadow entries are released. For aborted transactions, a fence is also inserted between the recovery and shadow entry release operations. The fence instruction guarantees that the memory state is restored before the data can be consumed by other transactions.

4.1.6 Correctness of Transactional Memory

The correctness of transactional memory is critical. For ensuring correctness, a strong isolation between concurrent transactions must be maintained. A strong isolation ensures that active transactions never access an inconsistent memory state. The proposed STM designs conform to this property. By performing eager conflict detection on reads, we ensure that speculative modifications made by an uncommitted transaction do not propagate to other transactions. Furthermore, the failed transactions do not leave the memory state inconsistent. Such transactions restore the speculative modifications using their undo logs before releasing the shadow entries.

4.2 Experimental Setup

We base our experiments on Nvidia Fermi GPU, GeForce GTX 480, which has 15 SMs, operating at processor clock of 1215 MHz, and has access to 1.5GB of GDDR5 memory. The SIMD width of each SM is 32. The shadow memory is allocated in the global memory, while thread-private metadata are stored in the local memory. Caching of the shadow memory is disabled at L1 level, thus is cached only at the L2 level. The local memory is cached in both L1 and L2 levels. The CPU consists of four 8-core AMD Opetron 6220 processors, running at 3 GHz frequency, and 198 GB of DDR3 memory.

4.2.1 Benchmarks

We select CUDA kernels from diverse sources [70, 22, 8] to study the effectiveness of the proposed STM designs. The benchmarks have varying degree of contention, number of memory operations, and transaction sizes. HASH-S [22] is a microbenchmark which implements a hash table data structure. Each transaction inserts an element into a hash bucket. If two threads access the same bucket simultaneously, the accesses are serialized. HASH-M benchmark is similar to HASH-S, but each transaction inserts multiple elements into the hash table, instead of a single element. BANK [22] is another microbenchmark in which each transaction withdraws amount from one account and deposits into the another. Accesses from multiple threads to the same account are serialized. SPATH [8] is the single-source shortest path algorithm that works on an undirected weighted graph. EQUAKE is the GPU version of 183.equake benchmark from SPEC CPU2000 [70] suite, which processes the *ref.in* input. More specifically, we parallelized the sparse matrix-vector multiplication function on GPU, where each transaction updates the matrix with computed product values.

To evaluate coarse-grained locking (CGL) performance, we converted fine-grained locks (FGL) protecting each data element to a common global lock, thus serializing all critical sections in the benchmarks. All the GPU benchmarks are compiled using CUDA 4.1 toolkit [56]. The CPU versions of the benchmarks are compiled using *g++ 4.6* with -O3 optimization enabled. The benchmark characteristics are listed in Table 6.2.

4.3 Evaluation

In this section, we discuss in detail the performance of the proposed STM designs from the perspective of overheads, scalability, and sensitivity to conflict detection granularity.

Name	Blk Size	Grid Size	# of Cmt.	Rd/ Tx	Wr/ Tx	% Crit. Time	SD
HASH-S	160	120	32K	1	2	67.8	80MB
HASH-M	160	120	32K	4	8	81.3	80MB
BANK	128	128	18M	2	2	78.5	78MB
SPATH	128	128	120M	2	1	73.6	4.2MB
EQUAKE	128	128	600K	3	3	32.2	354KB

Table 4.1: Benchmark characteristics. *Blk Size* and *Grid Size* correspond to the number of threads in a block and the number of blocks in the kernel, respectively; *# of Cmt.* indicates the number of committed transactions; *Rd/Tx* and *Wr/Tx* indicate the number of reads and writes performed by each transaction; *% Crit. Time* indicates % of time spent in critical section by running the benchmark with 1 thread; and *SD* indicates the size of shared data.

4.3.1 Performance Analysis

We compare the performance of STM against that of FGL and CGL. For experiments presented in this section, the conflict detecting granularity of STM is set to 4 bytes, unless specified explicitly.

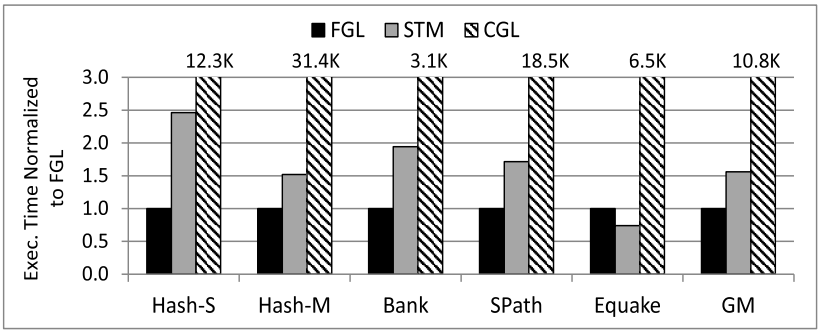


Figure 4.2: Performance achieved through STM on GPU.

Figure 4.2 shows the execution time of STM and CGL normalized to FGL on GPU for all the benchmarks evaluated. The values above one indicate slowdown, while those below one indicate the speedup compared to FGL. Overall performance is shown as geometric mean at the end of the figure. The STM execution bars represent the performance of the

best performing STM design among the three. For EQUAKE, the STM performs better than FGL (discussed later in the section), while for others the maximum slowdown is less than 2.5x. CGL, on the other hand, is three to four orders of magnitude slower than FGL, hence, is not a viable programming option on GPU. The overall slowdown caused by CGL is ∼10800x, while that of STM is only 56% compared to FGL. This shows that an efficient STM design can perform comparable to FGL at a low programming cost. However, we need to identify which of the three STM designs achieves the best performance.

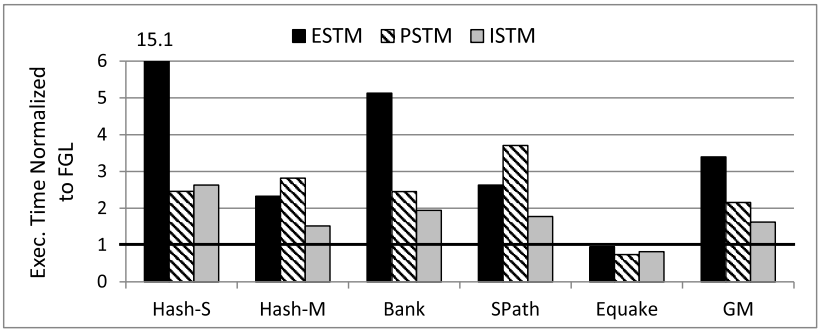


Figure 4.3: Performance of ESTM, PSTM, and ISTM normalized to FGL on GPU.

Figure 4.3 shows the execution time of ESTM, PSTM, and ISTM normalized to FGL. Our baseline design, ESTM, is 3.4x slower than FGL on average, while PSTM and ISTM designs are 2.16x and 62% slower than FGL, respectively. As shown in Figure 4.3, ESTM is never the fastest STM among the three. This is because of slower conflict detection in ESTM, which makes reads within the transaction visible to other threads. It requires ESTM to update the shadow memory during read as well as during commit operations. ISTM avoids this overhead by making reads invisible and validating them before the transaction commits, thus avoiding shadow memory updates for reads. PSTM does make reads visible, but unlike ESTM, it treats reads and writes same. Therefore, PSTM does not have to wait for other transactions to update the shadow entries.

Figure 4.4 shows the execution time breakdown of the three designs for each benchmark, running with a single thread. Each bar is divided into percentage of time spent

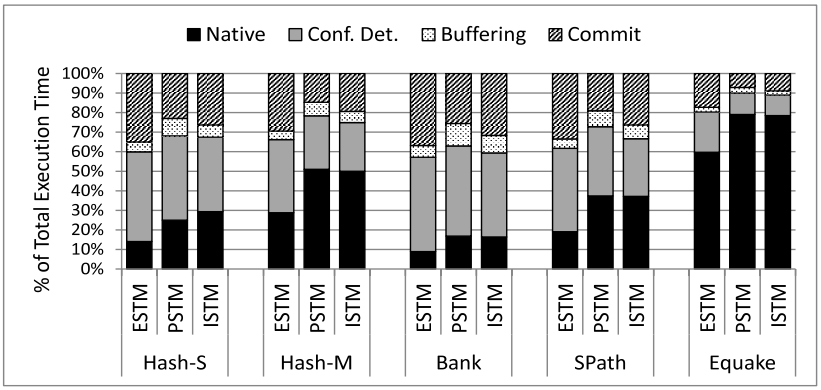


Figure 4.4: Execution time breakdown of a single thread execution for the three STMs.

in native code execution, detecting conflicts, buffering reads and writes, and committing the transactions. Notice that ESTM spends least amount of time executing native code compared to PSTM and ISTM, and it also spends more time in the commit operation. During commit, ESTM decrements the read count in shadow entries for every address read during the transaction. Conflict detection overheads of PSTM and ISTM are similar, but ISTM's commit overhead is higher because of the validation pass required for checking reads. Note that for EQUAKE, all three STMs spend significant time executing the native code because critical sections in EQUAKE contribute less in the total execution time as shown in Table 6.2. Consequently, all three STMs incur the least overhead for EQUAKE as shown in Figure 4.3. This shows that if the time spent in transactions is less, the STM overheads can be amortized by non-transactional execution in the application.

Among PSTM and ISTM, PSTM performs better under high contention (e.g. HASH-S — Figure 4.5) and when same data are read first and then written in the transaction (e.g. EQUAKE). In EQUAKE, each transaction first reads three elements in a matrix and then updates them. For EQUAKE, both PSTM and ISTM outperform FGL. In FGL, an atomic CAS operation is performed repeatedly until the lock is acquired. Many threads performing numerous atomic operations hampers the FGL performance of EQUAKE. On

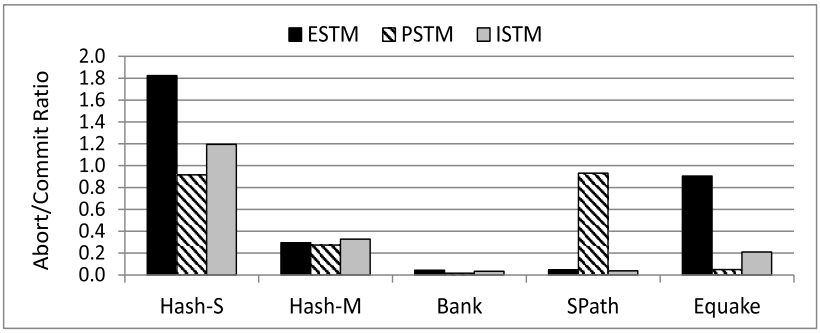


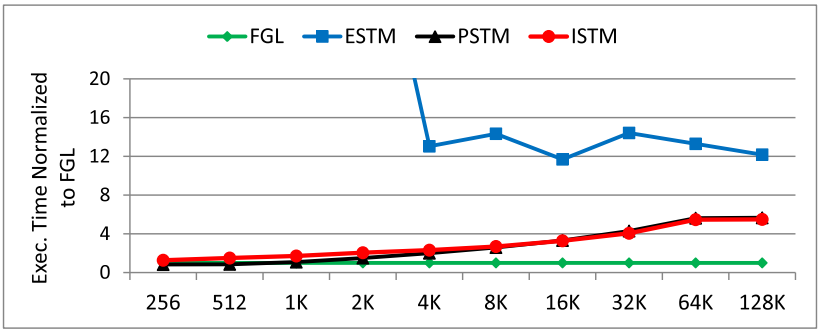
Figure 4.5: Abort/commit ratio of STM designs on GPU.

the other hand, in PSTM and ISTM, if a thread fails to acquire a lock, it aborts, recovers any modifications made, and retries rather than continuously spinning and waiting for the lock to be released. This minimizes the overall atomic operations performed in PSTM and ISTM, thus improving their performance. In EQUAKE, average number of retries per thread for FGL is 2.2, while abort/commit ratio for PSTM and ISTM are 0.05 and 0.21 as shown in Figure 4.5, respectively.

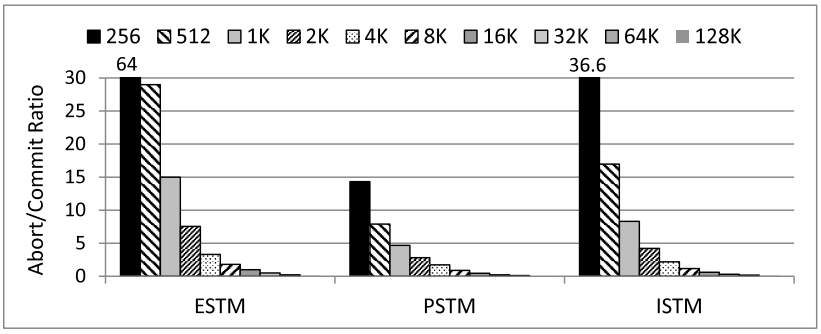
Performance of PSTM could suffer when an application has more number of reads than writes in the transactions. For example, in SPATH, when a thread processes a node, it reads its neighboring nodes for calculating their minimum distance from the source node. If the minimum distance is found, the node is updated; otherwise, no action is taken. In SPATH, PSTM detects conflict if two threads read the same node as PSTM does not differentiate between the reads and writes. Since actual updates to nodes are fewer, most of the conflicts detected by PSTM are false (Figure 4.5), which hampers its performance.

4.3.2 STM Scalability

In this section, we show that the proposed STM designs scale to different data set sizes and contention without significant impact on the performance. We present here results for two benchmarks, HASH-S and BANK.



(a) Performance of ESTM, PSTM, and ISTM normalized to FGL. X-axis represents the hash table size.

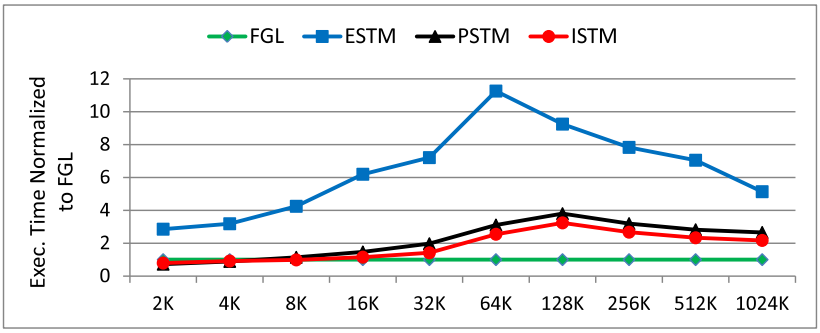


(b) Abort/commit ratio of ESTM, PSTM, and ISTM.

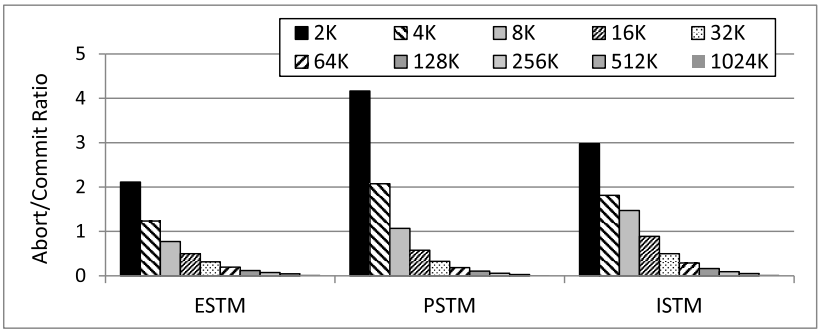
Figure 4.6: Impact on performance and abort/commit ratio in HASH-S benchmark with varying hash table size.

In HASH-S, the number of hash table entries are varied from from 256 to 128K. Figure 4.6(a) shows the performance of all STMs with varying hash table size, while Figure 4.6(b) shows the impact on abort/commit ratio. As seen in Figure 4.6(b), the conflict rate increases exponentially with decrease in the number of hash entries. ESTM's performance degrades severely below 4K entries as contention increases, while other STMs, however, approach the FGL performance. Furthermore, below 1K entry hash table, PSTM outperforms FGL. On the other hand, as contention decreases with increase in hash table size, the gap between FGL and STMs broadens and settles at around 64K-entry hash table.

Figure 4.7(a) and Figure 4.7(b) show the impact on performance and abort/commit



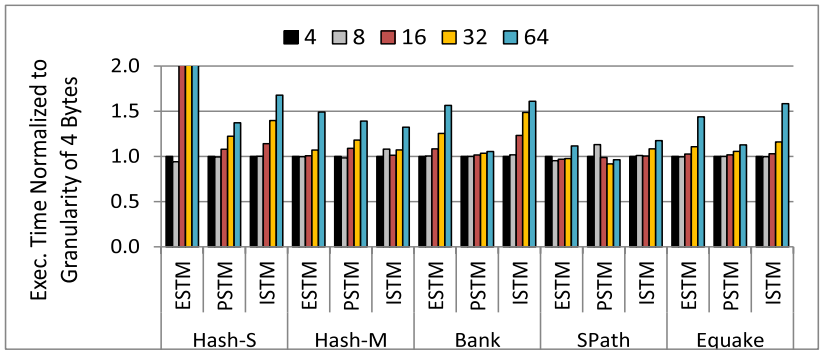
(a) Performance of ESTM, PSTM, and ISTM normalized to FGL. X-axis represents the number of bank accounts.



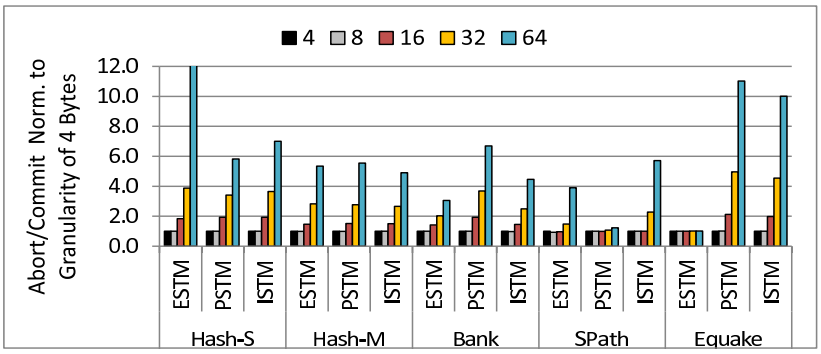
(b) Abort/commit ratio of ESTM, PSTM, and ISTM.

Figure 4.7: Impact on performance and abort/commit ratio in BANK benchmark with varying number of accounts.

ratio, respectively, for BANK benchmark when the number of accounts is varied from 2K to 1024K. Similar to HASH-S, contention increases with decrease in the data set size. Below 8K bank accounts, both PSTM and ISTM outperform FGL. At 1024K accounts or beyond, the performance stabilizes. Note that under very low contention, STMs on BANK perform better than STMs on HASH-S, compared to FGL. This is because transactions in BANK have more non-memory instructions than that of HASH-S, which can hide the penalty incurred by the transactional reads and writes. These two examples show that the PSTM and ISTM designs can scale well with varying data set sizes and contention.



(a) Performance of ESTM, PSTM, and ISTM normalized to granularity of 4 bytes.



(b) Abort/commit ratio of ESTM, PSTM, and ISTM normalized to granularity of 4 bytes.

Figure 4.8: Impact on performance and abort/commit ratio of three STM designs with varying conflict detection granularity. 4, 8, 16, 32, and 64 indicate granularity in bytes.

4.3.3 Memory Space Overhead for STM

In our STMs, the shadow memory tracks accesses to each memory location. The space required for shadow memory is same as the space required for storing locks in FGL. In this section, we evaluate the impact of varying shadow memory size on the performance and conflict detection rate of transactional execution. The size is varied by changing the shadow memory tracking granularity, which refers to the number of consecutive bytes each shadow entry corresponds to. A shadow entry can map to a single or multiple memory locations.

False violation can occur when two transactions access different memory locations, but map to the same shadow entry. Coarser tracking granularity leads to more false conflicts, but takes less memory space. Such conflicts can cause transactions to abort prematurely, thus wasting their work, and incurring the re-execution overhead. Figure 4.8(a) shows the performance impact of varying the access tracking granularity of transactions from 4 bytes to 64 bytes for the three STM designs. All bars are normalized to the execution with granularity of 4 bytes. Figure 4.8(b) shows the corresponding impact on abort/commit ratio. A general trend shows increase in conflicts and execution time with decrease in tracking granularity as a result of false dependence violations. Among the three STMs, PSTM is able to keep impact on performance minimal with increasing conflicts. On average, at 64-byte granularity, ISTM incurs 46% overhead compared to 4-byte granularity, while PSTM incurs only 17% overhead. Thus, when the space for shadow memory is limited, PSTM is a better choice than ISTM.

4.3.4 Selecting an Optimal STM Design

Our evaluation shows that ESTM design never outperforms both PSTM and ISTM designs due to its slower conflict detection mechanism; therefore, ESTM is never a desirable choice of STM. Among PSTM and ISTM, PSTM design should be selected when transactions in an application have write-after-read access patterns or contention is likely to be very high; otherwise, ISTM is a better choice. A compiler can assist in obtaining these application characteristics, which then can be utilized to select the most appropriate STM design for the given application.

4.3.5 Comparison with CPU

We now compare our best performing STM against the sequential and parallel execution on CPU. The pthread-based CPU versions of the benchmarks use up to 32 available threads. Figure 4.9 shows the speedup on CPU and GPU for various configurations relative to the

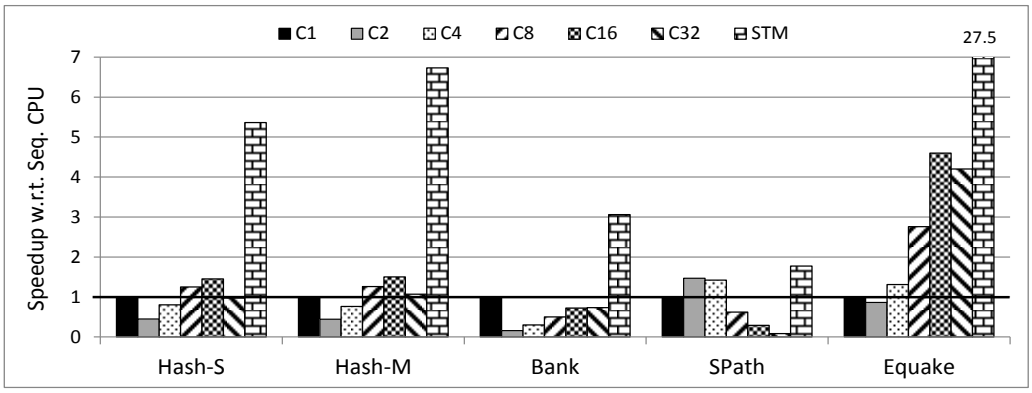


Figure 4.9: Performance comparison with CPU. Cn corresponds to CPU execution with n threads, while STM corresponds to our best performing design. Each bar is normalized to the sequential CPU execution time (C1).

sequential CPU execution. It can be seen that STM on GPU convincingly outperforms a 32-core CPU-based system.

4.4 Related Works

Software transactional execution on GPU has been proposed earlier. Cederman et. al [9] propose an STM on GPU that performs lazy conflict detection and lazy version management, specifically targeting high contention workloads. In their design, a transaction executes at a thread-block level rather than at a thread level. It means that threads within a thread-block do not cause dependence violation, however, such assumption may not hold true for all types of workloads. Therefore, we modify this design for evaluation and treat each thread as an independent transaction. We refer to this STM design as BSTM (Block STM). Xu et. al [75, 74] propose an STM on GPU that performs hierarchical validation by combining timestamp-based [17] and value-based validations [15]. We refer to this design as HSTM (Hierarchical STM).

Next, we compare the BSTM and HSTM designs with our best performing STM. On

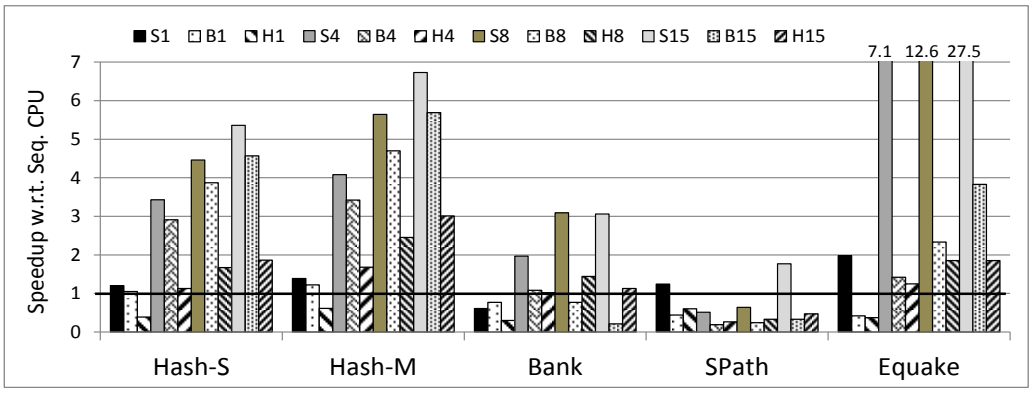


Figure 4.10: Scalability of STMs with thread count. Sk, Bk, and Hk correspond to our best performing STM, BSTM, and HSTM execution with k SMs, respectively. Each bar is normalized to the sequential CPU execution time.

GPU, we measure the performance of STMs with different thread counts, utilizing 1, 4, 8, and all 15 SMs, by varying the number of thread-blocks in the GPU kernels. Figure 4.10 shows the speedup for various configurations relative to the sequential CPU execution. It can be seen that our STM design scales well with the number of GPU threads compared to BSTM and HSTM. Both BSTM and HSTM perform opacity [23] checks on reads, which hampers their performance. For this check, BSTM waits during the read operation until a concurrent write from other transaction to the same location is complete, while HSTM performs incremental validation after every read that involves timestamp checking and value-based validation. In particular, HSTM causes significant slowdown for EQUAKE despite the transactions contribute less to the total execution time. Note that BSTM and HSTM are lazy version management designs, but even under high contention our eager mechanisms outperform these two because of lightweight conflict detection mechanisms.

Hardware transactional memory support on GPU has been proposed by Fung et. al [22, 21], which employs value-based conflict detection and performs lazy version management. The performance of the proposed HTM suffers with more number of concurrent transactions. Furthermore, significant hardware changes are required in GPU architecture

to support HTM. There have been other works that improve performance of irregular applications on GPU [54, 53]. Nasre et. al [54] have proposed generic techniques to accelerate morph algorithms having irregular access patterns on GPUs. In [53], global barrier-based synchronization and exploiting algebraic properties have been proposed to remove atomic operations from irregular applications on GPUs. The global barrier requires *all* threads to reach the barrier before the next phase is executed, which limits the number of threads launched in a GPU kernel. Further, the algebraic properties that can avoid atomics in certain applications are not observed in many irregular applications. Therefore, scalable STM support is essential in improving programmability of GPUs.

Extensive work has been done on supporting software transactions on CPUs [26, 64, 67, 18, 17, 15, 49, 27, 28, 68]. STMs on CPU employ complex mechanisms with the help of high level programming languages to improve their efficiency. However, since GPU threads are not powerful like CPU threads, the overhead in STM operations should be kept minimal. Therefore, in this work we propose lightweight STM support on GPU with low overhead conflict detection mechanisms. This support can result in more transaction aborts, but the large number of threads available in GPU can tolerate increase in aborts and still provide high throughput. By comparing with multicore CPU and state-of-the-art STMs on GPU, we have shown that our approach results in significant performance gains.

4.5 Summary

As GPUs emerge as a novel computing engine to drive the performance of future parallel workloads, it is important to facilitate the development of diverse applications on this platform. This work proposes software transactional memory to facilitate the development of applications that require lock-based synchronizations. We propose lightweight software transactions on GPUs that trade-off conflict detection accuracy with detection overheads. We evaluate these techniques on a set of benchmarks with various data sharing patterns.

Our analysis shows that slowdown caused by the best performing STM is only 56% compared to FGL. We also demonstrate that the proposed STMs can sustain different data set sizes as well as high transaction abort rates caused by false conflicts. Furthermore, outperforming parallel CPU execution by STM asserts GPU's potential to extract parallelism in irregular applications. Therefore, we believe that STM is a viable alternative programming model for developing GPU applications with irregular memory access patterns that require synchronizations.

Chapter 5

Hardware Transactional Memory on GPUs

In this chapter, we present a hardware support for transactional execution on GPUs. The previous chapter has demonstrated software transactional memory (STM) as a promising technique for extracting parallelism from irregular applications on GPUs, with minimal programming efforts. However, STM could introduce non-trivial performance overheads due to the additional instructions that perform transaction management. In particular, the overheads associated with conflict detection and recovery from failed speculation can be significant; however, these overheads potentially can be mitigated with adequate hardware support.

Hardware support for transactional execution for CPU-based CMPs has been extensively studied previously; however, these works cannot be directly extended to GPUs for two reasons. First of all, they often leverage existing infrastructures in CMP such as caches and cache coherence protocols [10, 25, 30, 76, 72]. Unfortunately, these infrastructures to enhance memory access efficiency are often absent in GPUs. Secondly, hardware support for transactional execution in CMPs is unable to scale to a large number of threads, and

is thus inadequate for GPUs that are capable of executing thousands of threads simultaneously.

Recent works have proposed transactional memory (TM) support on GPUs for applications that are already parallelized on GPUs using locks [9, 74, 22]. These approaches treat small sections of codes within critical sections as transactions. However, there is a significant effort involved, including algorithmic and data structure changes, to get these lock-based versions. On the contrary, we propose direct conversion of sequential applications to GPUs and executing them as transactions without making changes to the applications. Such transactions are much larger than those based on critical sections and have bigger speculative states. Prior approaches cannot effectively handle large transactions because of the various types of overheads associated with maintaining speculative states and performing transaction management based on these states.

Our hardware support for transactional execution scales to thousands of GPU threads. The transaction management overhead is minimized due to eager detection of ambiguous data dependences on every memory access. We provide a scalable mechanism to maintain large speculative states, and further optimize additional bandwidth requirements.

The rest of this chapter is organized as follows: Section 5.1 presents the hardware transactional execution model on GPUs. Section 5.2 discusses the hardware support required for implementing transactional memory on GPUs. Section 5.3 describes the simulation infrastructure and benchmarks, while Section 5.4 evaluates the performance of HTM. Finally, we discuss the related works in Section 5.5.

5.1 Transactional Execution on GPUs

In this work, we extend the existing GPU execution model to support transactional execution. In GPU, a *kernel* containing large number of threads performs the work on an array of computing cores, often referred to as *streaming multiprocessors* (SMs). Threads within the kernel are organized as *blocks*, where each block is further divided into groups of threads,

referred to as *warps* [56] or *wavefronts* [36]. Threads in the same warp execute in lockstep, i.e., all threads execute the same instruction and then proceed to the next instruction. Under the new model, GPU threads with ambiguous data dependences are referred to as *transactional threads*. These threads must satisfy the *atomicity*, *consistency*, and *isolation* properties defined for transactions. *Atomicity* states that a transactional thread either succeeds or fails as an entity; *consistency* states that the execution of a transactional thread brings the application from one valid program state to another (*consistency* implies the necessity for recovering program states in the case of transaction failures); *isolation* states that modifications made by an uncommitted transactional thread are invisible outside of the thread.

Figure 5.1 shows an overview of the proposed execution model on GPUs, where all threads belonging to the same thread-block are executed simultaneously as transactional threads. Execution is broken down into two phases. During the transactional phase, threads are treated as transactions: backups are created for each data stored; inter-thread data dependences are tracked; and threads that violate these dependences are marked as failed and are suspended. When all active transactional threads reach a barrier synchronization, a recovery phase is initiated. During this phase register contents are discarded and stores made by the failed threads are recovered with the data backed up during the transactional phase. At the completion of the recovery phase the application enters a valid program state, and is ready to re-enter the transactional phase to re-execute the *failed* transactional threads. In the absence of explicit synchronization, end of a program is implicit synchronization. A thread-block is complete when all transactional threads are committed. When a transactional thread fails, it is suspended, and no longer makes any progress. Thus, the warp scheduler must be able to ignore the status of failed threads when scheduling warps. Otherwise, the warp could be held indefinitely.

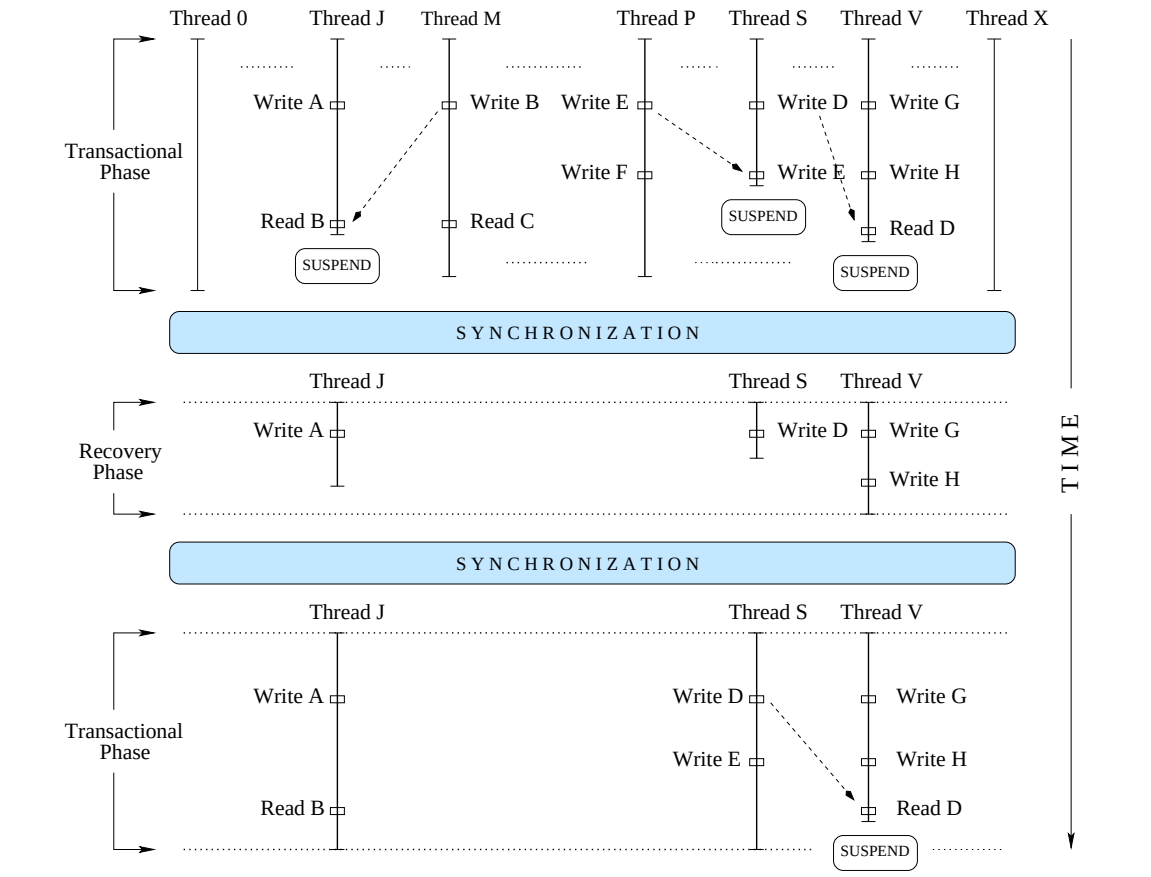


Figure 5.1: Transactional execution model on GPUs.

5.2 Hardware Support

In this section, we describe the hardware support necessary for ensuring correct transactional execution in GPU. The GPU is capable of supporting a large number of long executing threads, and thus we propose transactional execution support that is able to scale with the number and size of the threads.

The additional hardware structures necessary for supporting transactional execution are marked in shaded boxes in Figure 5.2. To each SM, two bit vectors, with width equal to the maximum thread count, are added (shown in Figure 5.2(a)): the *status bit vector* indicates whether the corresponding thread is a failed transactional thread; the *WrCount register* vector indicates the number of backed up memory locations for each thread. The *transaction management logic* in each memory partition, shown in Figure 5.2(b), is responsible for detecting conflicts and creating backups. GPUs memory subsystem is divided into separate memory partitions, each serving a range of memory addresses. Figure 5.3 shows the global memory map of GPU extended to store thread backups and keep track of speculative modifications. In GPUs, the shared memory is small and often explicitly managed by the programmer. Thus, we believe that it is suitable to rely on the programmer to disambiguate dependences in the shared memory. In this work, accesses to the global memory that are visible to all threads are considered for transaction management. However, the proposed mechanism can be extended to the shared memory as well.

In the rest of this section, we first describe conflict detection mechanism for identifying data dependence violations, followed by version management mechanism for providing the proper version of the data when responding to memory request and for recovering failed transactions. We then discuss bandwidth optimization techniques and hardware complexity of the proposed architecture.

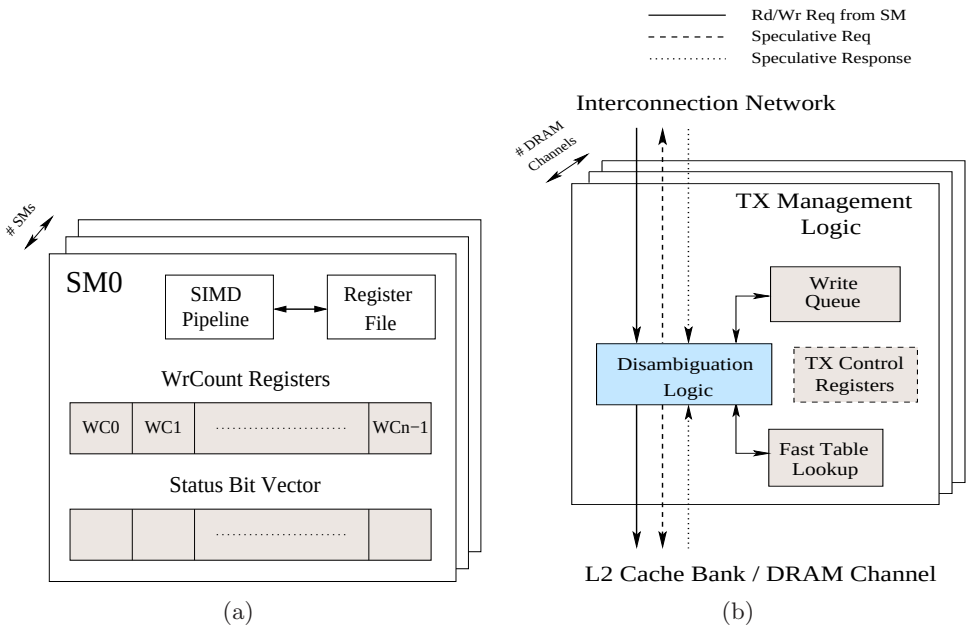


Figure 5.2: GPU architecture changes to support transactional execution (N = number of transactional threads). (a) SM augmented with per thread write count register and a status bit (n = SM thread capacity). (b) Hardware support to detect conflicts and create backups.

5.2.1 Conflict Detection

When two threads of execution access the same memory location, and one of the accesses is a write, a data dependence violation occurs and one of the threads must be squashed. To detect such violations, the runtime system can potentially maintain read/write logs for each thread, and compare log entries to identify dependence violations. Given the number and size of threads in GPUs, maintaining and comparing per thread logs is unrealistic, thus a centralized log that tracks the memory accesses from all threads is maintained in our implementation. Before the completion of each memory access, this log is queried; and the issuing thread is marked as failed if a dependence violation is detected. Failed threads are suspended. This scheme is referred to as the eager conflict detection scheme since data dependence violation detection occurs when memory accesses are issued. Eager

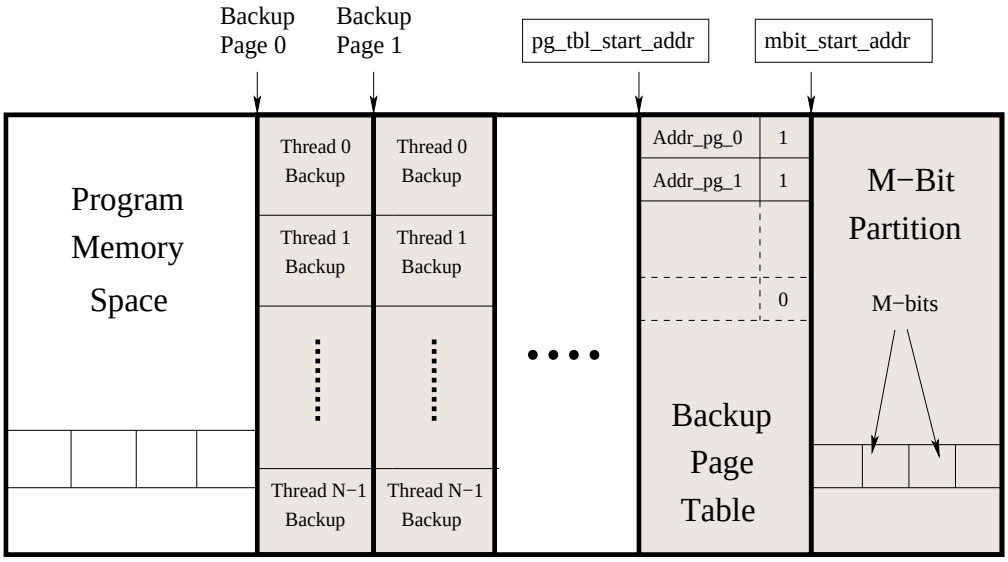


Figure 5.3: GPU global memory map: Transactional support requires backup pages, backup page table, and M-bit partition.

dependence violation detection allows for a smaller log, and thus, is particularly attractive for supporting scalable transactional execution on GPUs. Furthermore, by suspending conflicting threads early, more resources can be freed for other threads. The necessary hardware and software support for conflict detection is shown in Figures 5.2 and 5.3.

Scalable support for transactional execution requires adequate storage for recording access history to track memory accesses from all threads. In our implementation, the log is maintained in the global memory, where a section is allocated to keep track of all modifications from all threads, and is referred as the *M-bit partition*, as shown in Figure 5.3; M stands for *modified*. The starting address of the *M-bit partition* is stored in a control register and is initialized when the speculative kernel is launched. The *M-bit partition* is cleared at the beginning of every speculative execution phase through a low overhead mechanism similar to *cudaMemset* [55]. The *M-bit partition* acts as a shadow memory of the application's memory, and thus every address to be accessed in the transaction has a corresponding entry in the M-bit partition. An *M-bit* is a 16-bit value that stores the

ID of the thread that has speculatively modified the corresponding memory address. A value of zero means that no thread has speculatively modified the address. Granularity of M-bit mapping can affect the size of partition and conflict detection accuracy. If an M-bit corresponds to a larger chunk of data in the global memory, it may detect false dependences resulting in more thread failures. We evaluate different granularities for M-bits in our analysis.

Every memory access to global memory is validated by checking the corresponding M-bit entry. Address of an M-bit entry is easily calculated from the *M-bit start address* control register because of one-to-one mapping between the M-bit partition and the program's memory space. A read or write is considered safe if 1) the M-bit value is zero or 2) the M-bit value is the same as the thread ID in the control packet, indicating the same thread has already modified the location. A write request sets its thread ID in the M-bit partition when it finds that the address is not speculatively modified by any other transactional thread. After setting an M-bit, a backup copy of the original data is created as explained in the following section. If a conflict is detected for a thread, its corresponding bit is set to '1' in the status bit vector in the SM, shown in Figure 5.2(a). The status bit vector is used to detect failed threads and recover them; it is always reset to 0 when transactional execution starts. The warp scheduler in SM uses the status bit vector to suspend the failed threads.

In the proposed scheme, speculative reads and corresponding M-bit checks happen in parallel. However, writes must wait until M-bit check identifies if it is a safe access before updating the global memory. Figure 5.2(b) shows the write queue in the transaction management logic where write requests wait. It is possible that the queue contains write requests from two different threads to the same address. In this case, one of the requests is squashed using an associative lookup over the queue. It avoids more than one thread updating the same address in the program memory space. Dependent reads are served from the same queue.

Tracking Reads As discussed earlier, M-bit partition records only writes, and can detect read-after-write and write-after-write violations. However, transactional memory should also be able to detect write-after-read (WAR) violations. To detect WAR violations, we maintain a per-thread read log that stores the addresses of speculative reads. Once all threads finish their speculative execution phase, we validate the read logs. Specifically, for each entry in the read log, the corresponding M-bit is checked. If the M-bit value is zero or is the same as the ID of the thread, the read access is deemed safe; otherwise, a conflict is detected. After the read validation is complete, the conflicting threads are recovered and executed again.

5.2.2 Version Management

Version management requires buffering sufficient information such that a failed thread can be brought back to its original memory state to restart its execution. Under the proposed execution model, no transactional thread reads data modified by other threads. There are two ways to accomplish speculative writes; either speculative modifications should be done in place (eager) or buffered separately (lazy). In place modifications automatically commit when a transaction finishes without failure. Therefore, eager version management is a better approach than the lazy scheme, as most of the times transactions are expected to be successfully committed due to inherently parallel workload.

We have created a section in the global memory, referred to as *backup partition*, to keep the backup copies generated on writes during transactional execution. Whenever a speculative write is issued, its backup is stored in this section of memory. The backup partition is required during recovery of the failed threads. To restore the data during recovery, its address in the global memory must be known. Therefore, backups are maintained as <*address, data*> tuples, where the *address* is the target of a speculative write that is backed up. The *address* part of the tuple comes from the write request control packet from SM and the *data* part is obtained by reading the location pointed by *address*

from the global memory. The total amount of space required for backup depends upon the number of speculative writes made by all the threads, which is unknown at the start of the transaction. Allocating a big chunk of memory initially for backups would be inefficient.

To deal with the demand for backup space during transactional execution, we propose a page based backup mechanism. A backup page contains fixed space for each transactional thread to store its tuples. The space allocated to each thread is configurable through a GPU register. If a thread utilizes all its space in the backup page, a new page is created with the same capacity. Future backups from the thread are now stored in the newly allocated backup page. To keep track of backup pages, a page table is also maintained in the global memory. The size of the page table is very small compared to the size of backup pages. Initially, a single page is created and an entry is made in the page table. When a page is not found in the page table, we make a system call to the GPU library to allocate a backup page and update the page table accordingly.

Figure 5.3 shows backup page table and backup pages in the global memory. Each entry in the page table contains two fields — start address of the backup page and a valid bit. Valid bit '1' indicates that the corresponding page table entry has a valid address to a backup page. Figure 5.3 shows two backup pages corresponding to the 0th and 1st entries in the backup page table, where each page contains backup space for N threads. All the tuples of a transactional thread are stored in contiguous memory locations in a backup page to make the backup and recovery processes simpler.

To keep tuples of a same thread in contiguous memory locations, we introduce one *WrCount* register per thread in an SM, shown in Figure 5.2(a), which keeps track of the number of speculative writes made by the thread during its transactional execution; it is reset to zero at the start of a transaction. The value of the *WrCount* register is incremented when an SM executes a store instruction for the corresponding thread. Apart from the *WrCount* registers, we have added four programmable control registers to support the paging mechanism in GPU. The *page table address* register stores the start address of the

backup page table. The *thread backup space* register stores the number of bits required to specify the maximum number of tuples a thread can have in a single backup page. The *page table entry size* register stores the number of bits needed to specify the size of a single entry in the page table. It is required to get an entry from the backup page table. The *tuple size*, an architectural parameter stored in another control register, indicates size of a tuple in bytes.

To store the backup tuple, an address in the backup page corresponding to the requesting thread ID is required. The tuple address is calculated using our paging scheme. First, the backup page address is obtained from the page table using *WrCount* obtained from the control packet. Second, offset of the tuple inside the backup page is calculated. The sum of the backup page address and tuple's offset gives the exact address to store the tuple. These calculations are implemented using bit shift and logical operations, the details of which are omitted here due to space constraints. Such operations are easy to implement in hardware using basic gates, when the control registers used in these calculations contain values that are powers of 2.

Fast Backup Page Table Lookup There are two memory accesses required to store each backup tuple — reading the page table entry and writing the tuple. Writing the tuple into backup partition is a compulsory access. But we save accesses to the page table by having a small cache like structure in the transaction management logic, as shown in Figure 5.2(b), which stores the *page id* and the corresponding *page addr*. The number of writes would vary for each transaction; it might happen that a few threads have issued more writes than other threads and have higher *WrCount* values. Such threads will access different backup pages than the threads behind them. In our simulations, two-entry lookup is able to remove most of the misses except the compulsory ones.

Failed Speculation Recovery Failed threads must be recovered before they are re-executed. We recover the failed threads when all transactional threads reach the end

of a transaction. All thread blocks synchronize using standard barrier synchronization techniques in GPUs. Recovery is performed in the backward direction, which ensures the correctness of the memory state. In our model, a failed thread is responsible for its own recovery, making recovery a parallel process. Recovery can be done using software as well as hardware. For software recovery, we use a small recovery routine. The routine performs three functions — 1) generate an address to read a tuple, 2) read the tuple, and 3) restore the tuple data. The routine reads control registers to generate tuple addresses. Similar support could be easily added in the hardware to perform thread recovery, resulting in much lower overhead.

Avoiding Livelocks Due to eager conflict detection, it is possible that a transaction aborts another and is itself aborted later. If the pattern repeats, a livelock can occur. CPU based TM systems use random backoff to avoid livelocks [1, 6]. Livelocks in a GPU can be identified in the recovery phase by reading M-bit entries when data are recovered. If a transaction failure is caused by another transaction that has also failed, then these two threads are candidates for livelock, and can be scheduled differently during re-execution. In addition, the failed transactions can adopt a backoff mechanism by skipping the next transactional phase(s).

5.2.3 Bandwidth Optimization

In our proposed architecture, the M-bit partition in the slower global memory is accessed for every read or write request from an SM. To reduce the number of accesses to global memory for M-bit reads, we use Bloom filters [5] and on-chip buffers; buffers store the recently accessed M-bits. If a miss is detected in a buffer, then the M-bit check request goes to L2 cache or DRAM.

Bloom filters are useful to detect addresses that are speculatively modified and thus reading the M-bits only for accesses to such addresses. It reduces the number of speculative

messages in the network and decreases the DRAM pressure. A Bloom filter signature is created using the addresses of all successful writes in a memory partition. The address of each memory access in a memory partition is cross checked with its signature to detect if the same address is already modified by any thread. If the signature contains the address to be accessed, then the corresponding M-bit is read from the global memory. Otherwise, M-bit read is not required since no thread has modified the address. Signatures have probability of false hits, which results in unnecessary M-bit reads. However, they will never miss a real address match that could lead to a race condition. There is one write signature per memory partition. The signature configuration is similar to the default signature in Bulk [10].

5.2.4 Hardware Complexity

In each SM, per thread write count registers (32 bits each) track the number of speculative writes made by each thread, and the status bit vector indicates active or suspended transactions in an SM. If each SM supports 1024 threads, it needs 4KB register file for the write count registers and 1Kbits for the status bit vector.

Fast page table lookups in the transaction management logic have negligible hardware overhead. Disambiguation logic takes various decisions based on the types of requests, implemented using logic gates that does not need significant hardware resources. We use the existing write queues available in GPU to hold the write requests waiting for M-bit reads. Our Bloom filter signatures are 2Kbits each, which need total of 2KB space for 8 memory partitions. To improve the M-bit read performance, we add on-chip buffers on top of L2 cache, which take 64KB space for all the partitions. Resources needed in the transaction management logic are constant and do not depend upon the additional compute power or the number of transactions.

Overall, for Nvidia's Fermi GPUs, the additional hardware will cost less than 1% of the total chip area. Therefore, we do not expect significant increase in power consumption with the transactional support. Also, the proposed changes should not affect the GPU clock

period; however, the access latencies will increase due to the additional memory accesses.

5.3 Infrastructure

We evaluate the performance of the proposed transactional execution model on a detailed GPGPU simulator (GPGPU-Sim [4]) that is extended with the proposed hardware support for transactional execution. The architectural configuration, shown in Table 5.1, is modeled after the NVIDIA Fermi architecture [57]. The simulator models the SM, the interconnection network, the memory controllers, and DRAM in detail. L1 data caches in GPUs are non-coherent, which cache thread-private local memory and global memory. However, stale global memory data should not be present in L1 caches for coherent accesses. Local memory does not cause speculation to fail. Hence, our baseline model does not include L1 data caches. When coherent data is required by the kernel across all SMs, caching of global memory data in L1 caches is avoided by setting a flag during kernel compilation.

To compare the effectiveness of GPU on irregular applications with data-level parallelism, we compare the performance of a single GPU SM and a similar die area superscalar processor operating at the same clock frequency. Although modern GPU and CPU often differ in die area and operating frequency, we have decided to level these parameters to focus on evaluating the impact of different architectural features. GPU die area analysis is based on Fermi, which has a transistor count of three billion [57, 58], and the transistor count for each SM is approximately 130 million, including a 64KB on-chip memory that can be used as L1 cache or shared memory. A superscalar processor with similar transistor count is the SPARC64 V that contains 190 million transistors [19]. We adopted the Simics-based [47] Gems [50] simulator to evaluate the superscalar CPU, and the CPU configuration is shown in Table 5.2.

Processing Unit Configuration	
Number of SMs	16
SIMD Pipeline Width	32
Warp Size	32
Number of Threads per SM	1024
Number of Registers per SM	32768
L2 Data Cache	256KB/8 way/128B line
Number of Memory Channels	8
Number of GPU Clusters	4
DRAM Request Queue Size	32
Memory Controller	Out-of-Order
Warp Scheduling	Round Robin
Branch Divergence	Immediate Post-dominator
Interconnect Configuration	
Network Topology	Butterfly
Routing Policy	Destination Tag
Virtual Channels	1
Virtual Channel Buffer Size	32
Flit Size	32
Transactional Execution	
Backup Page Size	256 Entries/Thread
Fast Table Lookup	2 Entries/Memory Partition
Bloom Filter	2Kbits/Memory Partition
M-bit Buffers	8KB/Memory Partition

Table 5.1: Architectural configuration parameters for the GPU.

5.3.1 Benchmarks

Supporting a transactional execution model facilitates the parallelization of irregular applications with inherent data-level parallelism, but are difficult to parallelize on GPUs due to ambiguous data dependences. The Lonestar [38, 39] benchmark suite exemplifies such applications. Overall, we use eight benchmarks from Lonestar and Equake from SPEC CPU2000 [70] for our evaluation. Table 5.3 lists the benchmarks and their characteristics. The benchmarks are compiled using CUDA [55].

We converted the original CPU-based *C* benchmarks to GPUs, and used a worklist-based approach for task allocation using a bit vector to allocate items from the worklist to GPU threads. The worklist is present in the global memory with a bit per item indicating if the item needs to be processed. Each thread works on an item from the worklist if the bit is set. Successful threads reset the bit, while failed threads do not. After the recovery

Fetch/Decode/Dispatch/Execute/ Commit width	4/4/4/4/4
L1 Cache (size/assoc/line)	64KB/4 way/64B
L2 Cache (size/assoc/line)	512KB/4 way/128B
Reorder Buffer Size	128
Instruction Window Size	64
DL1 MSHR Entries	128
Branch Prediction Table	16K Entries
TLB Entries	64
L2 Access Latency	6 Cycles
Memory Access Latency	80 Cycles

Table 5.2: Architectural configuration parameters for the CPU.

Benchmark	**Short Name**	**#Comt.**	**IPC**	**Rd/Tx**	**Wr/Tx**	**#Inst/Tx**	**#Inst/Rd**	**#Bkup Pages**
Shortest Path	SPT	200000	9.16	39.05	7.02	184	2.67	1
Spanning Tree	STR	200000	5.93	58.24	15.98	58	3.73	3
Survey Prop	SPR	40000	6.92	145.7	41.58	605	3.45	1
Delaunay Ref	DRF	5200	35.02	19992.62	55.73	119607	5.97	1
Delaunay Tri	DTR	3000	24.79	56424.29	528.13	338688	5.95	8
Preflow Push	PUSH	300000	9.67	53.84	10.11	238	3.78	1
Barnes Hut	BH	32768	33.48	1043.35	5	11243	9.58	1
Equake	EQK	200000	6.38	222.53	23.51	518	2.11	1
Boruvka	BVK	17500	13.33	225.91	43.03	2175	9.62	2

Table 5.3: Benchmarks' characteristics. The numbers correspond to simulations with 16 SMs, each running 128 threads with bandwidth optimizations enabled.

phase, items having bit set are processed again.

SPT calculates single-source shortest paths from a given source node to all the other nodes in the graph. Node-to-node distances are iteratively updated as shorter distances are discovered. Each node is processed in parallel to update distances of its neighbors from the source node. *STR* produces a spanning tree that connects all the nodes in a graph without creating cycles. This algorithm randomly selects an edge from the graph to add it to the tree if no cycle is formed. The algorithm terminates when there are no more edges left which can be added without creating cycles. *SPR* is a heuristic SAT solver that determines whether there is a set of values that can satisfy a boolean expression. A boolean expression and its variables are represented using a graph. An edge exists between an expression and a variable if the variable is used in the expression. The algorithm evaluates all the nodes

in the graph and computes the likeliness for each variable to be true. Once the value of a variable is determined, it is removed from the graph. All the nodes, including both variables and expressions, are evaluated in parallel as long as there is no edge between them. *DRF* is a Delaunay mesh refinement algorithm that refines the Delaunay triangulated graph so that it satisfies certain constraints, such as no angle is less than 30 degrees in the graph. The algorithm identifies triangles that do not satisfy the quality constraints and refines them. Refinement is performed in parallel for all triangles that are not adjacent. *DTR* triangulates a graph containing a set of points specified by their coordinates. The graph is initialized with one triangle with all the points located inside. The triangulation process selects points from the graph, and partitions the surrounding triangle into smaller triangles. Points can be processed in parallel that do not share the same surrounding triangle and their neighbors do not overlap and cause speculation to fail. *PUSH* computes maximum flow from a source to a sink node through the edges of the directed graph. Nodes push the excess flow to their neighbors. Nodes having excess flow (input exceeds output) are active nodes. Active nodes are processed in parallel as long as the nodes they process do not overlap. *BH* is a gravitational N-body simulation, where particles are placed in a 3-D octree. In each timestep of the simulation, all threads first descend the octree and insert particles to the tree; compute the total force on each particle from the neighbors; and then finally update the position and velocity of each particle. Nodes can be processed in parallel when they are not ancestors of each other in the octree. *EQK* is the parallel version of 183.equake from SPEC CPU2000, which simulates the propagation of seismic waves by tracking the displacement at each point in an unstructured mesh over several time steps. Computation is dominated by sparse matrix-vector multiplication, which is speculatively parallelized on a GPU. *BVK* implements the Borůvka's minimum spanning tree algorithm for an undirected weighted graph. For each node, the edge with the smallest weight is selected and added to the spanning tree. This edge is then removed from the original tree, and the nodes it was connecting are merged. Edges can be contracted in parallel if the

nodes involved do not share a common neighbor.

5.4 Performance Evaluation

In this section, we evaluate the effectiveness and efficiency of supporting transactional threads in a GPU with the infrastructure and benchmarks described in the previous section. All the data presented in this section are based on the M-bit granularity of 4 bytes, unless specified explicitly. Figure 5.4 shows the IPC of each benchmark with 128, 256, 512, and 1024 threads executing on a single SM. Three benchmarks are unable to execute with the 1024-thread configuration since they oversubscribe the register file in this configuration. The height of the bars corresponds to *total* IPC, which represents the number of instructions graduated per cycle from all transactional threads, including those that failed. The *effective* IPC corresponds only to instructions graduated from successful transactional threads. The difference between effective and total IPC is the overhead associated with transactional execution.

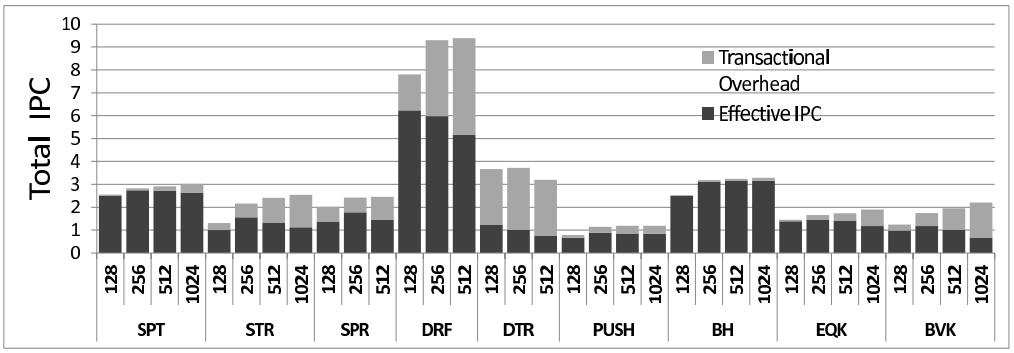


Figure 5.4: IPC achieved for transactional execution: *Transactional overhead* corresponds to instructions wasted in executing failed threads. *Effective IPC* corresponds to instructions executed by succeeded threads.

The *total* IPC increases sub-linearly with the number of threads. In the case of DTR, *total* IPC actually decreases when thread count increases from 256 to 512 as a result of

very high thread failure rate causing lower utilization of the SM. The *effective* IPC is always lower than the *total* IPC. For almost all applications, *effective* IPC decreases as the number of threads per thread block increases beyond a certain value due to significant overhead associated with the failed threads. Three factors determine the IPC of these applications: i) control flow divergence; ii) streaming multiprocessor resource contention; and iii) transaction management overhead.

Control Flow Divergence Divergence in control flow causes threads in the same warp to take different execution paths, degrading program performance as a result of serialized execution. A program with control flow divergence does not always have full warp occupancy. Divergence is the primary cause of low *total* IPC in all applications.

Streaming Multiprocessor Resource Contention In all applications, the performance is unable to increase linearly with the number of threads. Higher number of threads effectively hide the memory latency by scheduling more warps; thus, resulting in better execution time. However, the percentage of stall cycles increases with the number of threads. As we increase the thread count, the stalls become dominant over the useful cycles, which increases the execution time. Such behavior also explains the effective IPC decline in Figure 5.4. Decrease in a GPU's performance is expected when contention increases, but with transactional execution the degradation of performance is observed much earlier.

Transaction Management Overhead Transaction management overhead includes the cost of additional reads and writes, and the cost of transaction failure recovery. Dependence tracking and state buffering for transactional reads and writes can increase memory access time, as described in Section 5.2. Furthermore, the additional memory traffic generated by the transaction management logic increases network and memory contention, and lead to performance degradation. This impact is discussed in details in Section 5.4.3.

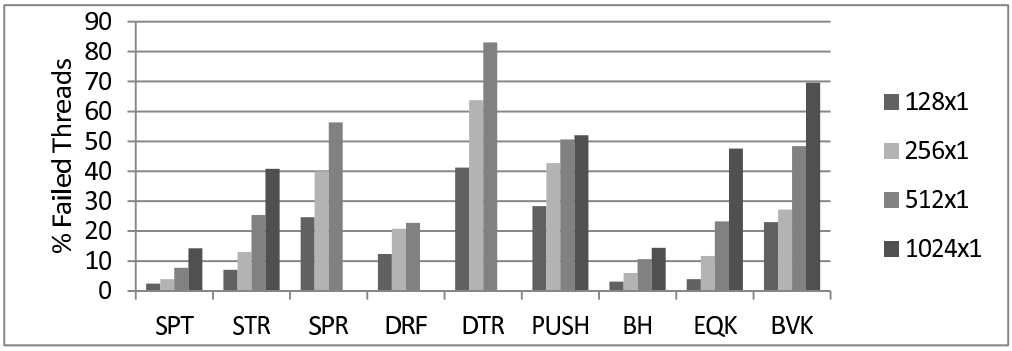


Figure 5.5: Failure rate of transactional threads obtained after running 128, 256, 512, and 1024 threads on one SM.

When transactional threads fail, they must be recovered and re-executed. On an average, the cost of recovery is relatively small — 2.4% of total cycles are spent in recovery and the maximum is for SPR, which is below 10%. However, the performance overhead associated with transaction failure, shown as transactional overhead in Figure 5.4, is significant for some benchmarks. For a given thread count, failure rate decreases as the problem size increases; on the other hand, for a given problem size, failure rate increases as the thread count increases, as shown in Figure 5.5. High transactional failure rate observed in SPR, DTR, and PUSH is an indication of insufficient parallelism in these application for the given data set. The higher failure rates for more threads in STR and BVK are because the problem size decreases as the application starts to converge. DRF does not have a high failure rate, but it suffers from speculation overhead due to a significant number of instructions executed by failed threads. All these benchmarks have substantial speculation overhead as seen in Figure 5.4, except SPT and BH due to their very low failure rate.

Since our execution model allows parallelization of sequential irregular applications on GPU, we compare the performance of transactional execution on a single SM against an out-of-order superscalar processor with the similar transistor count, operating at the same frequency. The results are presented in Figure 5.6 as the speedup achieved by GPU over

the single-threaded execution on a superscalar core, with M-bit granularity of four bytes. All the benchmarks perform better than CPU with maximum average speedup of 4.6x for 256-thread configuration. With bandwidth optimization techniques enabled the speedup becomes 5.1x, which is an improvement of 10%. It is worth pointing out that the CPU takes advantage of on-chip L1 and L2 caches, while the GPU does not. The integration of on-chip cache further improves the GPU performance as discussed later in Section 5.4.3. Further performance study of lock-based irregular applications on multicore CPU against GPU with transactional execution support is possible, but we leave it for future work.

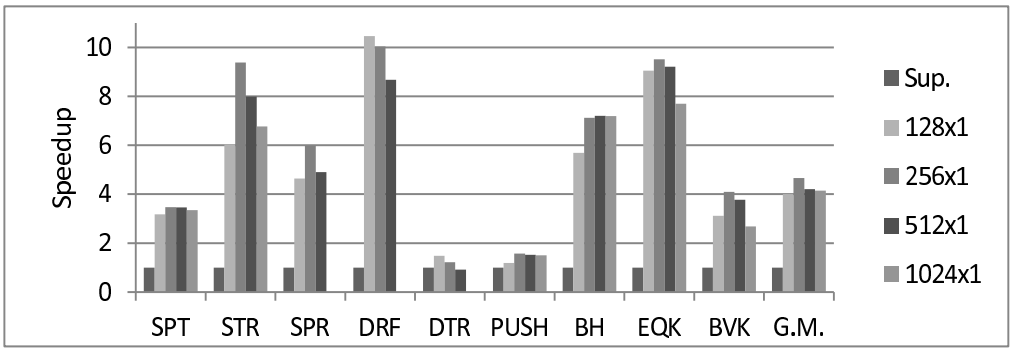


Figure 5.6: Performance of transactional execution on GPU (one SM) compared to superscalar CPU. (Sup. = Superscalar)

The transactional execution model is able to effectively extract parallelism from the benchmarks studied, with the exception of DTR and PUSH. DTR works on a small graph, where enough parallelism is not available for the thread numbers we have selected, limiting its speedup over sequential execution. With 256 threads and beyond, it experiences excessive squashing, limiting the best speedup at 128 threads. PUSH has a very high divergence which limits its performance. It runs four threads or fewer most of the time. In BH, there are two opportunities for speculative parallelization, to create a tree for computing gravitational forces and to compute the forces. The force computing kernel did not encounter any speculation failures with M-bit granularity of four bytes. The speedup

for BH is combined speedup of both the kernels. Note that the GPU benchmarks are not fine-tuned for performance. These benchmarks can benefit from detailed performance tuning; however, it is beyond the scope of this work.

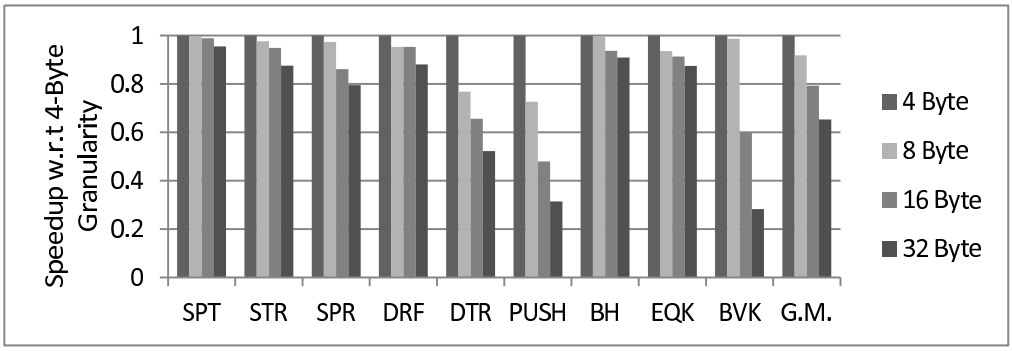


Figure 5.7: Effect of varying M-bit granularity on the execution time for 256-thread configuration. Normalized to 4-byte granularity.

5.4.1 M-bit Granularity Impact

M-bits track the speculative modifications made by transactional threads. Each M-bit entry corresponds to multiple bytes of data in the program memory space. False violation occurs when two memory operations access different memory locations that are mapped to the same M-bit entry. *M-bit granularity* refers to the number of bytes each M-bit entry corresponds to. Coarser M-bit granularity leads to more false sharings, but takes less memory space for the M-bit partition. The performance impact of varying M-bit granularity is shown in Figure 5.7, as slowdown caused by different M-bit granularities compared to that of 4 bytes. All benchmarks suffer performance degradation, as M-bit granularity becomes coarser. PUSH and BVK use an array to represent a graph. As M-bit granularity becomes coarser, neighboring nodes map to the same M-bit entry. PUSH has only a few active nodes that can be processed in parallel, and each node is represented by a 4-byte value. Thus, when M-bit granularity is 8 or more, false violations increase

dramatically. DTR works on a smaller graph, and thus has a high false conflict rate. On the other hand, BVK shows little performance degradation with 8-byte granularity, but false violations increase with 16- and 32-byte granularities. Other benchmarks use pointer-based graph data structures and the active nodes are spread throughout the graph, resulting in fewer false conflicts. To summarize, the impact of M-bit granularity is application- and input-dependent, and thus is best determined at runtime.

5.4.2 Scalability Study

Figure 5.8 shows the performance achieved on 16 SMs with bandwidth optimizations present when thread count is increased from 64 to 512 threads per SM. DTR, PUSH, BH, and BVK show improvement in execution time for 128 threads compared to 64, while others degrade. Monotonous decline in the performance for some of the benchmarks is attributed to their memory contention. Table 5.3 lists the average number of instructions issued per load operation for each benchmark. Benchmarks with a very high percentage of loads cannot hide the memory latency despite having more warps. Higher number of threads increases the load latency further which cannot be accommodated with additional threads. Such benchmarks do not scale with more threads. Our experiments on unmodified GPGPU-Sim comply with our observations. Therefore, under transactional execution these benchmarks show degradation due to additional overheads incurred by speculation messages and transaction management.

On the other hand, BH and BVK have the lowest number of loads compared to others, which scales their performance with more threads. BVK shows a decline in performance for 256 threads and beyond due to excessive squashing. Likewise, DRF and DTR scale with higher number of threads, but increase in the failure rate impacts their execution time. Compute-bound benchmarks easily scale when the overhead due to failed threads is not dominant.

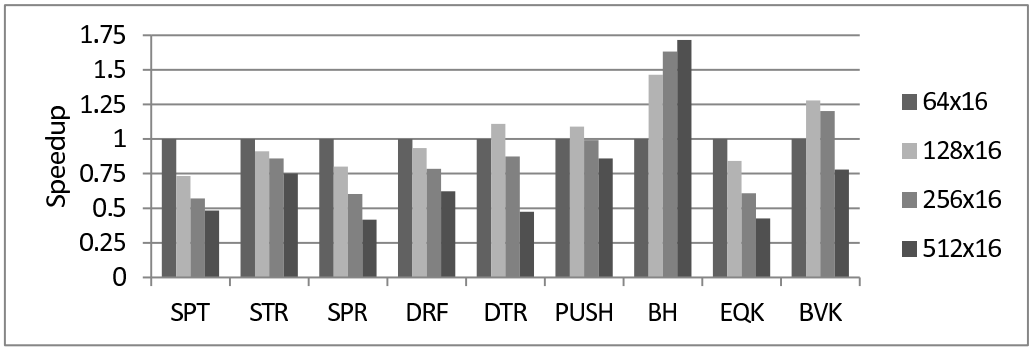


Figure 5.8: Performance of transactional execution when number of threads per SM are varied from 64 to 512, on total of 16 SMs. Normalized to 64x16 results.

5.4.3 Bandwidth Utilization

Transactional execution introduces additional memory traffic for M-bit accesses and version management. This traffic can exacerbate existing memory bottlenecks and degrade performance. Additional hardware that can potentially reduce this memory traffic, described in Section 5.2.3, is evaluated here. Figure 5.9 shows the DRAM request distribution for each benchmark with different optimizations. The base configuration does not include L2 cache and all the memory requests are served by DRAM. The speedup achieved by each optimization relative to the base configuration is labeled on the top of the bars. In this study we simulate 16 SMs, each running 128 threads.

We classify the transactional execution traffic into M-bit read, M-bit write, and backup requests. Every memory access has a corresponding M-bit read while every new write generates M-bit write as well as backup requests. A write operation by the owner thread of an address creates only backup requests. All the benchmarks benefit largely by adding L2 cache to the GPU reducing the total DRAM traffic. Most of the M-bit write requests are absorbed by the L2 cache. However, backup requests reaching the DRAM are not effectively reduced due to their uncoalesced nature and fewer numbers. With L2 cache, DRAM traffic is reduced by 56%, while the performance is improved by 28%.

Table 5.3 shows that for most applications, the write set is much smaller than the read set. Bloom filter takes advantage of this property to reduce the M-bit read requests. DRF and BH have the lowest write to read ratio, which results in negligible M-bit read requests with a Bloom filter. On average, addition of a Bloom filter further reduces DRAM traffic by 20% and improves performance by 21% relative to the L2 cache. An additional M-bit buffer that only buffers M-bit entries not only reduces the DRAM traffic, but also prevents pollution of the L2 cache. Thus, this buffer can improve transactional execution performance significantly. Compared to a Bloom filter, an M-bit buffer reduces DRAM traffic by 5% and increases the performance by 12%.

Signatures and buffers potentially reduce the M-bit access traffic over the network due to effective caching in the cases when the M-bit of an address is mapped to a different memory partition. SPT, SPR, DTR, and EQK gain most from these buffers. To summarize, there is an average reduction of 70% in DRAM traffic and 88% performance gain compared to the base configuration, which shows that additional bandwidth requirements by the transactional execution are effectively mitigated.

5.5 Related Works

Transactional memory support on GPUs has been previously proposed. Cederman et al. [9] propose STM using locks on GPUs, while Fung et al. [22] propose support for HTM. The prior approaches treat critical sections as transactions, whereas our execution model treats segments of execution between two barrier synchronizations as transactions. This is because our goal is to facilitate the extraction of parallel threads on GPU from sequential irregular applications. As a result, our mechanism must be scalable to the size and the number of threads.

Fung et al. propose a technique (KILO TM) with lazy version management and lazy conflict detection mechanisms which requires separate read and write logs for uncommitted

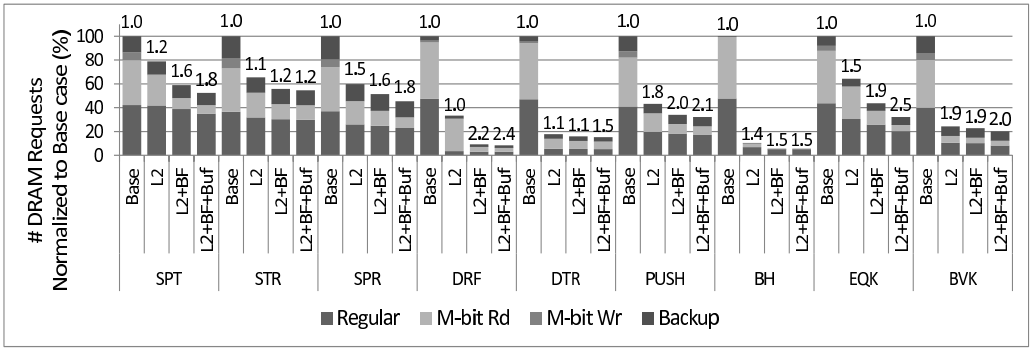


Figure 5.9: Distribution of total DRAM traffic with 16 SMs, 128 threads/SM: *Base* corresponds to traffic when no cache is present between transaction management logic and the DRAM; *L2* corresponds to traffic when a L2 data cache is introduced; *L2+BF* corresponds to traffic when an additional bloom filter is introduced; *L2+BF+Buf* corresponds to traffic when a separate M-bit buffer is introduced. Hardware description of these components can be found in Section 5.2.3. All bars are normalized to the *Base* configuration. Numbers on each bar indicate speedup w.r.t. *Base*.

transactions. We implement an eager conflict detection using a centralized M-bits structure. For each memory access, our scheme only requires one additional read for conflict detection, while KILO TM needs two additional reads and one additional write. With our benchmark set, average read/write set size is much larger than those analyzed by Fung et al. This increases the conflict detection overhead in KILO TM because it transfers the entire read/write logs for each thread to commit units and re-validates every entry in the logs against the global memory. Bigger logs also cause more misses in L2 cache which can increase the DRAM pressure and validation time in KILO TM. To identify whether a thread is accessing its own write set, KILO TM employs per-thread Bloom filter. A hit in the Bloom filter leads to a sequential scan of the write log, which could lead to performance degradation when the size of write log is large. This overhead is eliminated in our mechanism. Furthermore, while Fung et al. allocate a large fixed space for storing read/write logs, we dynamically manage our backup logs. By providing a more restrictive execution, our eager/eager scheme is simpler than that of KILO TM.

Thread-level speculation and transactional memory in CPU have been an important area of research in the computer architecture community for many years. Numerous hardware implementations of transactional memory (HTM) have been proposed [3, 10, 11, 13, 14, 25, 46, 76, 72]. Extensive work has been done using software to support transactional execution [17, 26, 29, 48, 64, 69]. Conflict detection and version management are then performed in software using low-level synchronization primitives supported by the underlying processors. STMs are often slower than HTMs. Hybrid TM support have also been proposed that benefit from both faster HTM and more scalable STM [16, 40]. In the case of HTMs, caches, cache coherence protocols, and virtual memory are used to implement version management and dependence violation detection. Similar to LogTM-SE [76], we use eager conflict detection and version management in our GPU-based transactional execution support. LogTM-SE is able to use separate read and write signatures per thread for conflict detection in hardware. However, since GPUs support thousands of threads, maintaining two signatures per thread and performing conflict detection would incur unacceptable hardware cost as well as performance overhead. By using a part of the global memory to track speculative modifications, our approach can easily scale to thousands of transactional threads on GPUs.

5.6 Summary

In this work, we propose a transactional execution model on GPUs to extract parallelism from sequential applications with irregular memory access patterns. The execution model allows straightforward conversion of sequential applications to GPUs and eases the burden on programmers to parallelize them. To scale the proposed execution model on GPUs to thousands of threads, we adopt eager conflict detection which effectively utilizes GPU resources, and eager version management which favors transactions with lower failure rate. We have integrated hardware support onto the streaming multiprocessors and in the memory subsystem to enhance the efficiency of transactional support. We compensate for the

additional memory bandwidth requirements using techniques like Bloom filters and on-chip buffers. By evaluating a selected set of benchmarks from the Lonestar [38] and the SPEC CPU2000 [70] benchmark suites, we found that a large class of applications with inherent data-level parallelism and irregular memory access patterns can be parallelized on GPU. With adequate hardware support, the overhead associated with managing transactional states and transactional failure recovery can be minimal. Transactional execution on a GPU achieved 5.1x speedup compared to sequential execution on a superscalar CPU of similar die area, operating at the same clock frequency.

Chapter 6

Data Race Detection in GPUs

Designing and implementing correct GPU programs is challenging since programmers must consider interaction between thousands of parallel threads. Many concurrency bugs are results of different threads perceiving memory states differently at runtime. Such differences are often a manifestation of data races present in the programs. Concurrency bugs pertaining to data races are often difficult to identify, since variant thread scheduling can change the behavior of the program. A data race occurs when more than one thread simultaneously accesses the same memory location, and at least one of the accesses is a write. Improper placement and usage of synchronizations and critical sections can lead to data races. Furthermore, the lack of memory coherence support in contemporary GPUs can introduce data races that do not exist in computational systems with a coherent memory system such as CPUs. In addition, a large number of concurrent threads in a modern GPU makes detecting and resolving data races a challenging task.

Efficient and effective runtime data race detection mechanisms are the basis for implementing powerful tools for facilitating software development as well as enhancing software correctness and reliability. Runtime data race detection techniques on CPUs have been thoroughly investigated by a large body of previous work [51, 52, 59, 60]. These proposals often require some mechanisms for tracking memory accesses per-thread. However, for a

modern GPU that is capable of executing thousands of threads simultaneously, tracking per-thread accesses to detect data races poses challenges both in terms of performance as well as hardware overhead.

There have been some efforts initiated recently for ensuring software correctness in GPUs [44, 7, 34, 42, 43, 77, 78]. These approaches mainly include software-based static analysis or dynamic monitoring of GPU programs. The static analysis approach inspects the program source code, and based on the knowledge of the GPU execution model, can either detect the static races or predict runtime races. This approach is limited since it cannot foresee the runtime program behaviors; thus, can result in significant false positive rate. The dynamic monitoring approach adds software instrumentation to GPU programs, thus causes significant slowdown. However, it is more effective as the runtime program control flows and indirect memory accesses are exposed to the monitoring system.

Adequate hardware support can potentially improve the effectiveness and efficiency of data race detection. Such hardware support is responsible for tracking and comparing the memory accesses from all threads to detect data races. In this work, we propose a Hardware-Accelerated data Race detection tool for GPU (HAccRG), a low overhead, high accuracy data race detection mechanism for GPU. To the best of our knowledge, HAccRG is the first hardware implementation for detecting data races in the GPU. It detects data races in both shared and global memory spaces of a GPU.

The rest of this chapter is organized as follows: In Section 6.1, we elaborate data races occurring in GPUs with the help of two examples. Section 6.2 presents the data race detection framework on GPU. Then, we discuss the necessary hardware and software support for data race detection in Section 6.3. The evaluation methodology is described in Section 6.4, followed by evaluation of HAccRG in Section 6.5. Finally, we discuss the related works in Section 6.6.

6.1 Data Races in GPUs: Case Studies

Data races are one of the common issues faced in concurrent programming that are hard to debug and can often result in incorrect program behaviors [45]. A data race occurs when more than one thread simultaneously accesses the same memory location, and at least one of the accesses is a write. In this section, we present two classes of data races in GPUs. Some of the data races discussed in this section occur in both CPU and GPU, while the others occur only in GPU.

```
1.  __device__ uint count = 0;
2.  __device__ void race_example (int *in, int *out)
3.  {
4.    int tid = threadIdx.x;
5.    for(int i=0; i¡32; i++) {
6.      out[tid] = foo(in, tid, i);
7.      // missing memory fence
8.      if(blockDim.x-1 == atomicInc(&count, blockDim.x)) {
9.        out[0] = out[0]+out[1]+...+out[blockDim.x-1];
10.       count = 0;
11.     }
12.     // missing barrier synchronization
13.   }
14. }
```

Figure 6.1: Two sources of global memory data races in a GPU kernel: missing memory fence (line 7) and missing barrier synchronization (line 12).

6.1.1 Lack of Correct Synchronization

In the example shown in Figure 6.1, threads within the same thread-block update the array *out* in global memory at the index *tid* in `line 6`. Each thread atomically increments a global variable *count* (initialized to 0) in `line 8`. Every atomic increment operation returns the old value of *count*. The last thread from the block that increments the variable *count* reads the global array modified by all threads in the block and writes the final sum

to *out[0]* in `line 9`. The same thread resets the *count* to 0. The entire process is repeated in a *for* loop.

Because of the *for* loop in `line 5`, all threads but the last one, after atomically incrementing *count*, immediately loop back and update the *out* array. It can happen that before the last thread reads the *out* array in `line 9`, the other threads modify *out* again. A barrier synchronization (e.g. *syncthreads* [56]) in `line 12` is required to ensure correct execution. Note that, memory accesses from threads within the same warp are always ordered. The barrier synchronization ensures correct ordering of memory accesses across all warps in a thread-block.

GPU does not guarantee ordering of memory accesses across all threads. For this reason, the last thread that reads the *out* array in `line 9` can see the *count* value incremented before the write to the *out* array is complete, and can compute the sum incorrectly. A barrier synchronization in `line 7` will ensure memory accesses are properly ordered across all threads in a block; however, this is an overkill since it blocks all threads. A memory fence function [36, 56], on the other hand, if added in `line 7`, ensures that all the writes made by a thread are visible to all other threads after the memory fence operation. In systems with coherence support, a fence call is not required because stricter memory consistency is maintained by hardware. Thus, when more than one thread-block updates and accesses the global memory, memory fence functions must be used to ensure correct ordering of the memory operations.

Thus, when more than one thread-block updates and accesses the global memory, memory fence functions must be used to ensure correct ordering of the memory operations.

6.1.2 Incorrect Use of Locks

Locks allow multiple threads to read or write shared variables atomically by serializing the accesses from different threads. We describe two data races that can occur while implementing locks. Figure 6.2(a) shows execution of a critical section in a GPU kernel.

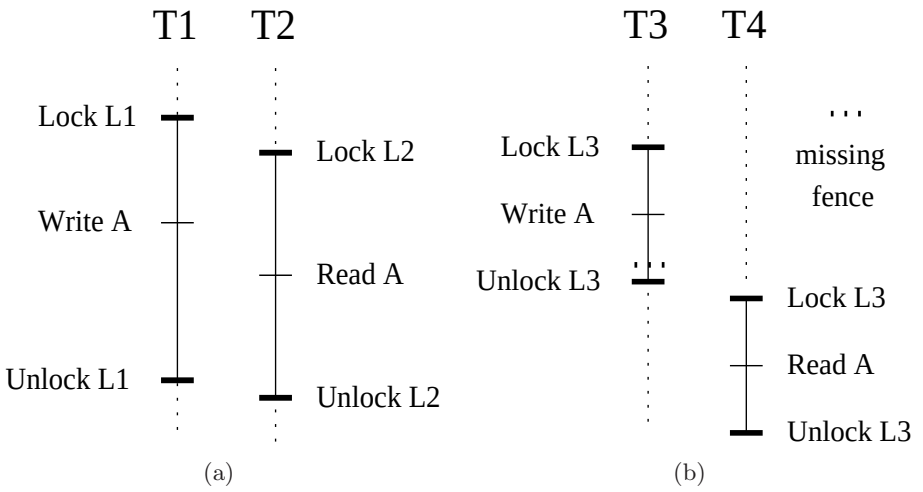


Figure 6.2: Data races in critical sections on a GPU. (a) A data race that occurs in both GPU and systems with coherence support. (b) A data race due to missing fence that only occurs in GPU.

Threads *T1* and *T2* acquire locks L1 and L2, respectively. Thread *T1* writes to an address A within its critical section, while thread *T2* reads the same address before thread *T1* releases its lock L1, resulting in a data race. This type of data race can occur in both GPU and systems with coherence support.

Figure 6.2(b) shows two threads *T3* and *T4* contending for the same lock L3, which causes both threads to execute their critical sections sequentially. Thread *T3* writes to an address A within its critical section, while thread *T4* reads the same address upon acquiring the lock L3. However, since GPU does not guarantee memory access ordering across threads, it is possible that *T4* finds the lock variable updated before the write to the address A is complete. Thus, thread *T4* can read the old value at address A. This can be avoided by calling a memory fence function after write to address A and before *T3* releases lock L3. This type of data race can occur only in GPU.

Data race shown in Figure 6.2(a) can occur in both CPU and GPU, however, situation in Figure 6.2(b) does not occur in CPU because of stricter memory consistency model than

GPU. Both these cases result in data race conditions that often could require significant debugging efforts since such an execution pattern may not be obvious from the kernel code.

Detection of all the data race conditions discussed above demand expert debugging skills and familiarity with the code. In addition, the data dependent races increase the debugging efforts. An automatic data race detection technique will save much of the debugging efforts and time, improve the quality of the software, as well as reduce its development time.

6.2 Data Race Detection Framework

There are two common techniques for detecting data races at runtime: comparing set of locks held by threads when accessing shared data items or establishing happens-before relationship between accesses. Lockset-based data race detection technique tracks the set of locks held by each thread and reports a data race when different threads access the same memory location without holding a common lock, with at least one write access. In happens-before-based data race detection technique, thread execution is partitioned into epochs using synchronization events. Such epochs are considered concurrent if their execution times overlap, while a data race is reported when concurrent epochs from different threads access the same memory location with at least one write. Although both techniques are powerful, none of them covers all the data races. The lockset-based detection only covers data races caused by improper use of locks. In addition to the runtime data races, it also reports the potential data races that do not occur at runtime. The happens-before-based detection considers all synchronization events, but only reports data races that occur at runtime, while potentially missing some data races that are not exposed due to thread scheduling. HAccRG uses happens-before mechanism to detect data races caused by incorrect barrier synchronizations and fences, and lockset mechanism to detect data races in critical sections.

To detect data races, memory accesses from different threads must be tracked. There are two approaches for recording memory access history: aggregating memory accesses for

each thread or tracking memory accesses for each memory location. In the first approach, memory accesses from each thread in a GPU are compared against each other. Since the number of comparisons grow quadratically with the number of threads, this approach is impractical for GPU systems with thousands of concurrent threads. Thus, HAccRG takes the second approach that keeps track of thread(s) that accessed each memory location. This approach is particularly attractive for GPUs, because the number of data race comparisons has a linear relationship with the number of memory accesses and the space overhead grows linearly with the application memory size.

The metadata for data race detection for a shared data is stored, in what we refer to as, *shadow memory*. Consider a memory location M: accesses to M are tracked in the shadow memory at location M_{shadow}. Each access to memory location M invokes a concurrent access to the corresponding M_{shadow} entry. In the rest of this section, we describe what information is tracked by the shadow memory; and how different data races are detected using this information. The hardware/software support necessary for implementing the shadow memory depends on the performance characteristics of the shared data, and is described in details in Section 6.3.

6.2.1 Races Between Barrier Synchronizations

In the GPU, barrier instructions ensure that memory accesses issued before the barrier are completed before instructions following the barrier are executed. Hence, memory accesses across barriers are ordered, however, all accesses between two barrier synchronizations are concurrent. Between two barriers, a data race can occur when different threads read and write shared data. The start and the end of a kernel are implicit barriers.

In HAccRG, when a thread accesses a memory location, the corresponding M_{shadow} entry is checked and updated. For each memory location, the corresponding shadow entry has three fields: the *tid* field containing the thread ID of the first thread that accessed the given memory location; the *modified (M)* field indicating whether the memory location

has been written to by a thread; and the *shared (S)* field indicating whether the memory location has been read by more than one thread.

Figure 6.3 shows the state transitions of the M_{shadow} entries. At the beginning of the execution, the shared and modified fields in the shadow entries are set to true, indicating that there has been no access to the memory locations. When a thread-block reaches barrier synchronization, the modified and shared fields in the shadow memory entries are set to true. When a memory access is issued, a shadow entry can be in one of the four states:

- State 1 (M=true, S=true): There is no prior access to the corresponding memory location. HAccRG performs the following operations for memory access from thread T: i) reset S to false because this is the first access; 2) reset M to false if the access is a read; and 3) set the *tid* to the thread identifier of T.

- State 2 (M=false, S=false): There have only been read accesses from a thread with its ID stored in *tid*. For a read access, no action is necessary if the same thread accesses the location; otherwise set S to true. The shared field (S) is required because we only store the ID of the thread that has first accessed the given memory location. For a write access, set M to true for access from the thread *tid*; otherwise report a data race of type write-after-read (WAR).

- State 3 (M=true, S=false): There have been at least one write access from a thread with its ID stored in *tid*. For a read access, no action is necessary for access from the thread *tid*; otherwise report a data race of type read-after-write (RAW). For a write access, no action is necessary for access from the same thread *tid*; otherwise report a data race of type write-after-write (WAW).

- State 4 (M=false, S=true): There have been read accesses from more than one thread. For a read access from any thread, no action is necessary. For a write access from any thread report a data race of type write-after-read (WAR). The shared field is set

to true when more than one thread reads the location; hence, any write to the same location is a potential data race.

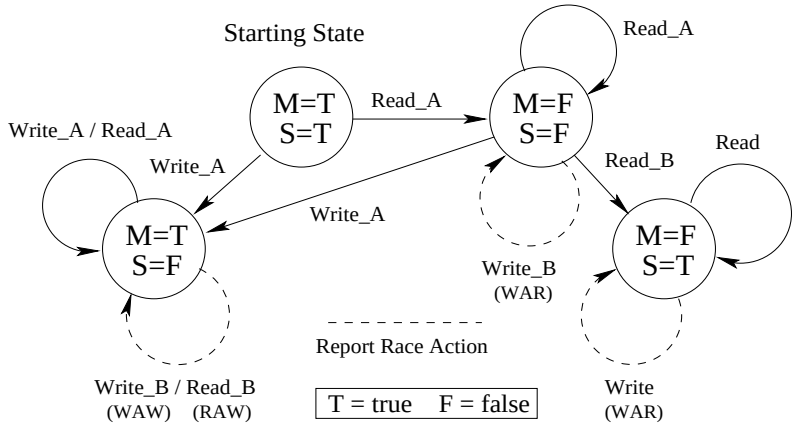


Figure 6.3: Data race detection process. Each state represents 'MS' fields in a shadow entry. *Read_x* or *Write_x* indicates an access by a thread *x*, while *Read* or *Write* indicates an access by any thread. The dotted lines indicate the report race action.

Impact of Warps on Reporting Races: After analyzing the shadow entries, HAccRG reports data races for instructions between two barrier synchronization points only if the two threads belong to different warps. The threads within the same warp do not create a data race condition since all threads in a warp are synchronized. Therefore, for every memory access by a warp, it is guaranteed that the previous memory accesses by the same warp are complete. The shared field in the shadow entry is set to true when a read occurs from a different warp than that of thread *tid*. However, HAccRG does detect write-after-write violations within the same warp before the memory request is issued.

When warp re-grouping occurs dynamically at runtime [20], threads that originally belonged to different warps are merged into a single warp. Consequently, for these threads some memory accesses would come from different warps. Thus, when warp re-grouping is enabled, HAccRG reports data races regardless of the warp considerations.

6.2.2 Races in Critical Sections

GPUs support atomic operations for implementing critical sections to ensure sequential access to shared data. When a memory location is accessed by different threads holding a common lock, the accesses to that memory location are serialized. To detect data races related to incorrect acquisition and release of locks, HAccRG supports lockset-based data race detection [65].

In HAccRG, a per-thread register, referred to as the *atomic ID*, records the set of locks held by each thread. When a thread acquires a lock, the lock variable is entered into the atomic ID; when the lock is released, the corresponding entry is removed from the atomic ID. The register is a small Bloom filter [5] that keeps track of lock variables, similar to the prior work for detecting CPU-based data races [79]. A Bloom filter signature is a bit vector divided into multiple bins. When an address is added into the signature, one bit selected using a hash function in each of the bins is set to 1. To remove addresses, we simply clear the signature when a thread releases all the lock variables held. Most of the CPU applications use single level locks, while the ones that use nested locks have very small nesting levels [61, 79]. The GPU applications also follow similar trends. Therefore, clearing the Bloom filter signatures provides a low overhead mechanism to remove lock variables from atomic IDs. Atomic ID is attached to each memory request that is issued within a critical section, and is stored in the shadow entry along with the other fields. A more accurate look-up table based approach for tracking lock variables can also be adopted, however we choose Bloom filter due to its low hardware overhead. The effect of Bloom filter signature size and number of bins on data race detection accuracy is discussed later in Section 6.5.

To determine whether two or more threads have accessed a shared variable without holding a common lock, HAccRG tracks the set of common locks protecting each shared variable at runtime. This is obtained by intersection of the set of locks protecting a shared variable so far, with the set of locks held by the thread accessing the shared variable. The

intersection of Bloom filter signatures is a simple bitwise *AND* operation. The atomic ID field in the shadow entry indicates the set of locks protecting the shared variable so far, while the atomic ID register of the accessing thread represents the current set of locks held by that thread.

In critical sections, lockset-based detection has priority over barrier synchronizations. For the first access to a memory location (M=true, S=true), the lock variable is stored in the atomic ID if the access is protected, otherwise the atomic ID is set to 0 in the shadow entry. For later protected accesses, the intersection of the atomic ID in the shadow entry and the atomic ID of the accessing thread is stored in the shadow entry. A data race can occur in two different scenarios when using locks.

- Accesses using different locks: For a protected access by a thread other than *tid*, if the shadow entry shows a non-zero atomic ID, a data race is reported if the modified field is true (write by *tid*) or the current access is a write, and if the intersection of the atomic ID in the shadow entry and the atomic ID of the accessing thread is null. A null intersection indicates that no common locks are used to access a shared variable.

- Unprotected accesses: When accesses to a memory location involve protected and unprotected accesses from different threads, a data race can occur if one of the accesses is a write. When the atomic ID in the shadow entry is non-zero and the current access is unprotected, or the atomic ID in the shadow entry is 0 and the current access is protected, a data race is reported when either the modified field is true or the current access is a write.

In HAccRG, we distinguish between memory accesses issued within and outside critical sections. To detect the start and end of a critical section, we insert marker instructions after lock acquire and before lock release operations in the kernel code. These operations

in GPU are often implemented using atomic compare-and-swap or atomic exchange instructions [36, 56]. Compiler support can also be used to insert the marker instructions automatically. Only for accesses related to critical sections, lockset-based detection is performed as described above. For other accesses, happens-before detection is used to detect data races.

6.2.3 Races Due to Improper Memory Fencing

In GPU, memory access ordering across threads is non-deterministic, as discussed earlier in Section 6.1. Programmers use memory fencing functions to impose explicit ordering on memory requests. A fence operation in CUDA ensures that modifications made by a thread are visible to other threads in a GPU before the instructions following the fence are executed by that thread [56].

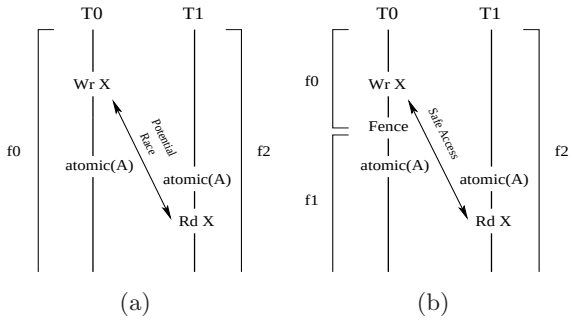


Figure 6.4: Detecting memory fence races in HAccRG. (a) Potential data race as thread T1 consumes T0's updates without fence execution by thread T0. (b) Safe access as thread T0 executes fence before performing atomic operation.

To understand importance of fences, consider two threads *T0* and *T1* that have producer-consumer relationship as shown in Figure 6.4. In Figure 6.4(a), thread *T0* writes to location X, while thread *T1* reads from location X. The order between *T0* and *T1* is ensured through an atomic operation over variable A. Such interaction between threads can be implemented without using explicit critical sections, similar to the example shown in Figure 6.1 where

an atomic increment operation over the *count* variable creates a synchronization point for all threads. At the time of read access by thread *T1*, HAccRG determines that thread *T0* has not executed memory fence function by monitoring *T0*'s fence epoch f0, and flags the access as data race. If a fence call is inserted as shown in Figure 6.4(b) after write to X, HAccRG flags the read access by *T1* as safe since *T0*'s fence epoch changed from f0 to f1. Similarly, HAccRG can also detect data races occurring in critical sections due to missing fences (Figure 6.2(b)).

To monitor fence calls in GPU, HAccRG maintains a per-warp *fence ID* that is incremented by one when a warp completes a memory fence call. Fence ID is a per-warp logical clock that keeps track of execution of fences by a warp, which is sent along with the memory requests. The fence ID of a thread's warp is stored in the shadow entry with the other fields. For the first access to a memory location (M=true, S=true), the shadow entry fields are set along with the fence ID. For all other accesses, if the modified field is set and the new access is a read, the fence ID stored in the shadow entry is compared with the current fence ID of the warp that belongs to the thread indicated by *tid*. A match indicates that the thread *tid* has not yet executed a fence instruction since its last write to the same memory location. Therefore, there is a data race since a different thread is reading the same memory location. A mismatch means the thread *tid* has completed its fence call since its last write to the same memory location, which indicates that other GPU threads can safely consume *tid*'s updates before the fence call.

6.3 HAccRG Implementation

In this section, we describe *Race Detection Unit* (RDU), the hardware support necessary for accelerating the detection of data races in GPU. Since GPUs have two separate memory modules, the shared and the global memory, two independent RDUs are designed to match the performance and bandwidth requirements of these memory modules. While they do not alter memory accesses originated from the cores, they are responsible for generating

shadow memory accesses and matching these accesses with the corresponding memory accesses. This ensures that the application running on the GPU does not observe any functional changes when HAccRG is enabled.

6.3.1 Data Race Detection in Shared Memory

Since the shared memory is private to each SM, the *shared memory RDU* is present in each SM for detecting data races among threads in the same SM. The shared memory is small and fast, and hence the shared memory shadow entries are stored in hardware for faster accesses. In reality, we extend each shared memory entry with its shadow memory information, as shown in Figure 6.5. For every shared memory access, the RDU accesses the corresponding shadow entries and performs data race detection.

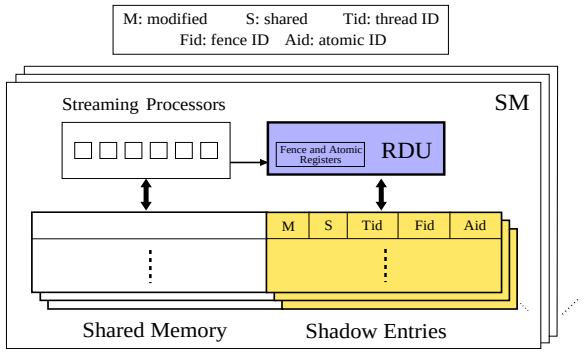


Figure 6.5: Modifications to the SM for the shared memory data race detection.

For every shared memory access, the RDU accesses the corresponding shadow entries and performs data race detection. When a thread-block reaches barrier synchronization, RDU resets all the shadow entries that belong to the block by setting their shared and modified fields to true. Note that the shared memory RDU accesses fence IDs in its own SM since shared memory is accessed only within an SM.

6.3.2 Data Race Detection in Global Memory

The global memory is off-chip and is much larger than the shared memory in size. Thus, it is impractical to provide explicit hardware support to track memory accesses. We reserve a section in the global memory, referred to as *global shadow memory*, to record the meta-data necessary for data race detection. The shadow entries in the global shadow memory have a one-to-one correspondence with the entries in the kernel data structures. The global shadow memory is allocated when the kernel is launched using the cudaMalloc [56] API. At execution time, the global memory RDUs automatically generate requests and updates to the shadow memory. When the kernel terminates, the cudaMemset [56] API is invoked to invalidate all global memory shadow entries. Since the global memory is accessible to all threads across all thread-blocks, we augment the shadow memory entries with the *bid* field for the block ID, and the *sid* field for the originating SM of the access. Since multiple thread-blocks can access the global memory, the shared field in the global shadow entries is set to true if two different warps or thread-blocks read the same global memory location. Consequently, a data race is reported if different thread-blocks perform read-write accesses to a shared data without using locks or memory fences.

Since global memory shadow entries are part of device memory, setting the modified and shared fields to true after barrier synchronization has additional performance overhead. To avoid this overhead, we add one more field in the global memory shadow entries, referred to as *sync ID*, to notify execution of barrier instructions. Each SM maintains per-thread-block sync ID, and increments a thread-block's sync ID when the thread-block reaches barrier synchronization and *only* if the block has accessed the global memory since its last barrier call. This avoids unnecessary increments to sync IDs after every barrier call. The sync ID is a logical clock which keeps track of barriers executed by a thread-block. It is sent with each global memory request and stored in the shadow entry. The sync ID in the shadow entry and the one sent with the memory request are compared when the request comes from the same thread-block as that of *tid*. If the sync IDs match, HAccRG checks the shadow

entry for data race as shown in Figure 6.3. If the sync IDs are different, the shadow entry is updated with the current access information. When a memory request comes from a thread-block other than that of *tid*, sync ID check is not required because scope of barrier is limited to a thread-block. For such accesses, the shadow entry is analyzed for detecting a data race and then updated.

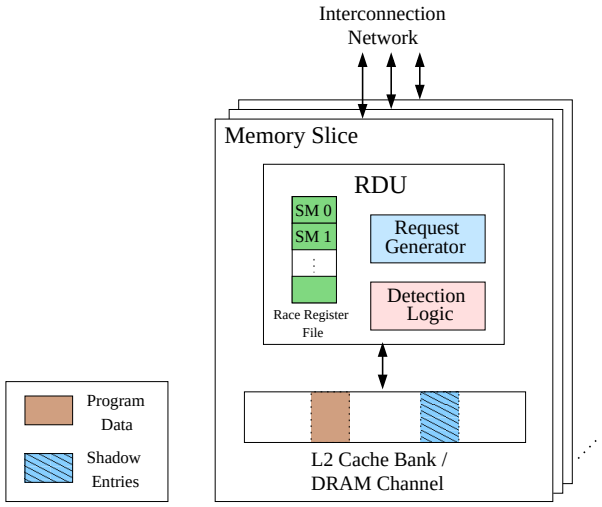


Figure 6.6: Modifications in the memory slices for the global memory data race detection. Each memory slice in GPU contains an L2 cache bank and a DRAM channel.

Hardware support for global memory data race detection are shown in Figure 6.6. The *global memory RDU*, a hardware module present in each memory slice, is responsible for data race detection. The GPU memory subsystem is partitioned into memory slices, each containing an L2 cache bank and a DRAM channel, where the global memory shadow entries reside. For every incoming global memory access from the interconnection network, global memory RDU generates a new data race detection request using the *request generator*. A data race detection request is essentially a read request to obtain the shadow entry. For global memory reads that hit in the L1 data cache, a data race detection request is sent to the corresponding global memory RDU. Upon reading a shadow entry, the *detection logic* analyzes the entry, reports a data race if detected, and updates the entry accordingly.

When multiple requests accessing the same global memory location enter a global memory RDU simultaneously, the requests are checked associatively to detect a possible data race.

Accessing Fence IDs: Memory fences in the GPU provide memory consistency across threads by imposing memory access ordering. RDUs access the fence IDs maintained by the SMs, to detect data races as described in Section 6.2.3. Global memory RDUs require access to fence IDs from all the SMs in GPU. Therefore, the fence IDs from all the SMs are stored in the *race register file*, which is replicated in every global memory RDU for faster access, as shown in Figure 6.6. At the time of data race detection, a fence ID is read *only* if the access checked for race is a read and the corresponding shadow entry has the modified field set. A data race is reported if the read fence ID matches with the one stored in the shadow entry. We discuss the sizes of sync IDs, fence IDs, and atomic IDs in Section 6.5.1.

Effect of L1 Caches on Detecting Global Memory Races: Non-coherent L1 data caches have been introduced in recent GPUs [57], enabling caching of global memory data in SMs. For global memory reads that hit in L1 data cache, HAccRG sends data race detection requests to the corresponding global memory RDUs. When thread-blocks executing on different SMs communicate through the global memory, it is possible that an SM does not get the global memory updates from other SMs because of the stale data in its own L1 data cache. HAccRG handles this case by sending the read hit information with the data race detection request to the global memory RDU. When global memory RDU detects a read-after-write access across SMs and if the read is an L1 hit, HAccRG reports a data race. Although such data races can be avoided by disabling caching of global memory data in the L1 caches or by declaring the shared variables as *volatile* [56], HAccRG is capable of reporting them when L1 caching is enabled. Systems with coherent caches do not face such issues.

Supporting Virtual Memory: Virtual memory simplifies programming by assigning separate address space to each process, thus offloading memory management overhead from the applications. Although virtual memory has traditionally been implemented only

for CPUs, recent GPUs have also deployed virtual memory [2, 35]. To track global memory accesses using shadow memory in such architectures, HAccRG should be able to support virtual memory.

Virtual memory systems use page tables for translating virtual addresses to physical addresses. In systems like Intel Sandy Bridge and AMD Fusion, page tables for CPU and GPU are separately maintained [2, 35]. We propose an on-demand paging for shadow memory in HAccRG. Shadow memory pages are allocated when GPU's application memory pages are generated, i.e., on-demand. However, shadow memory pages are allocated only for global memory space. A one-bit field in the GPU page table entry can indicate if the page belongs to the global memory space.

Modern processors rely on translation lookaside buffer (TLB) to speed-up the virtual address translation mechanism. Intel Sandy Bridge and AMD Fusion provide independent TLBs for GPU address translation. In HAccRG, when GPU's global memory accesses go through TLB, they should be able to get shadow memory locations along with the application memory. We propose two mechanisms to incorporate dual address translations in the TLB. A TLB is a cache like structure. The first mechanism appends 1-bit to the tag fields in the GPU TLB. This bit is set to 1 when a TLB entry points to a shadow memory page. During address translation, two tags are searched in the TLB: with the appended bit 0 and 1. This approach can potentially reduce the effective TLB capacity for regular (non-shadow) memory entries. In the second mechanism, we propose a separate TLB structure for shadow memory pages. GPU memory accesses go through regular as well as shadow memory TLB. Shadow memory TLB can be smaller than the regular TLB since all GPU pages do not belong to the global memory space. This approach provides faster TLB accesses than the first one. With the aforementioned modifications, HAccRG can be easily adapted for GPUs having virtual memory support.

6.3.3 Accuracy Trade-offs in HAccRG

Accurate runtime data race detection is often costly in terms of additional resources required, such as hardware and memory. Two approaches to tackle limited resources are either to reduce data race detection accuracy or to detect data races only on a subset of memory accesses.

By mapping one shadow entry to one or more consecutive elements in the program's memory space, the storage overhead of HAccRG can be adjusted. We refer to such mapping as *tracking granularity* of HAccRG. One-to-one correspondence between shadow entries and elements in the kernel data structures does not report any false positives, while one-to-many correspondence can report false positives. However, varying the tracking granularity changes the hardware as well as the memory requirements of HAccRG. Making the granularity coarser reduces these overheads significantly.

By tracking access information only for the recently accessed data, HAccRG can detect data races with high accuracy, however, it can miss some of the data races. In other words, when the available shadow memory is lesser than that is required at the finest tracking granularity, HAccRG overwrites the shadow memory contents with the latest access information. This is important as the data race conditions between memory accesses that execute close to each other in time are the most critical, while those which occur far apart are potentially benign races. Such approach is common for data race detection techniques in CPUs that track accesses at cache line granularity. When the cache lines are evicted, access information stored with them is also lost. Since the shadow memory size for global memory data race detection could be significant, we propose this technique for the global memory space.

For keeping access information only for the recently accessed data, we track the global memory accesses at page level by partitioning the global memory into pages of fixed sizes. A small page table in hardware keeps track of most recently accessed pages. The number of active entries in the page table is configurable, which also specifies how many shadow

memory pages are available for global memory data race detection. For each global memory access, the page table is queried. When a page entry is not found in the page table, a new entry is allocated by replacing the least recently used page entry. Essentially, the shadow entries belonging to the replaced page entry are overwritten. This technique can be easily incorporated in GPUs supporting virtual memory by tracking accesses only to those global memory pages which have entries present in the TLB.

We evaluate the data race detection trade-offs in HAccRG in Section 6.5.

6.4 Evaluation Methodology

# SMs / GPU Clusters	30 / 10
SIMD Pipeline Width / Warp Size	8 / 32
# Threads / Register per SM	1024 / 16384
Warp Scheduling	Round Robin
Branch Divergence	Immediate Post-dominator
Shared Memory per SM	16KB
L1 Data Cache per SM	48KB/6 way/128B line
Texture Cache per SM	5KB/5 way/128B line
Constant Cache per SM	8KB/2 way/64B line
Unified L2 Cache	64KB/Memory Partition, 8 way/128B line
# Memory Partitions	8
DRAM Request Queue Size	32
Memory Controller	Out-of-Order
GDDR3 Memory Timing	t_{CL}=10,t_{RP}=10,t_{RC}=35,t_{RAS}=25 t_{RCD}=12,t_{RRD}=8,t_{CDLR}=6,t_{WR}=11
Network Topology / Routing Policy	Butterfly / Destination Tag
Virtual Channels / Flit Size	1 / 32
Virtual Channel Buffer Size	8

Table 6.1: Hardware parameters.

We implement the proposed HAccRG changes by extending a detailed GPGPU simulator (GPGPU-Sim 3.0.2 [4]). GPGPU-Sim is configured to model NVIDIA Quadro FX5800 GPU. L1 data caches and a unified L2 cache have been modelled after NVIDIA Fermi GPUs [57]. The L1 data caches are non-coherent, while the unified L2 cache is coherent. GPGPU-Sim simulates timing for the SMs, the interconnection network, the memory controllers, and the GDDR3 memory. The GPU hardware parameters are provided in

Benchmark	Inputs	Shared Memory		Global Memory	
		% Inst.	% Shared Reads	% Inst.	% Shared Reads
MCARLO	256 elements, 64K paths	0.3	0	6.5	64.4
SCAN	512 elements	10.7	7.1	1.1	28.8
FWALSH	Data length 512K, Kernel length 32	9.6	0	7.5	0
HIST	Byte count 16M	22.9	0	1.4	0
SORTNW	32K elements, 2K values	17.1	0	2.6	0
REDUCE	1M elements	19.6	0.5	18.6	0
PSUM	16K elements	0.1	87.2	16.4	87.5
OFFT	meshW=256, meshH=256	8.5	0	2.6	9.4
KMEANS	color100.txt	0.04	0	17.4	86.3
HASH	256K-entry table, 16K elements	0	-	17.8	0

Table 6.2: Benchmarks. *Inst.* indicates % of shared or global memory instructions in a benchmark. *Shared Reads* indicates the % of reads to same memory location by different warps.

Table 6.1.

At the end of every barrier synchronization, we simulate the extra clock cycles required to invalidate the shared memory shadow entries. For the global memory data race detection, we model interconnection network traffic from the SMs to the global memory RDUs in the memory partitions. The network packets carry sync IDs, fence IDs, and atomic IDs along with the other control information.

We evaluate the effectiveness and performance of HAccRG using a set of CUDA applications listed in Table 6.2. Seven of the ten benchmarks are taken from NVIDIA's CUDA SDK [56]. MCARLO [56] is the Monte Carlo option pricing algorithm. SCAN [56] is the parallel prefix sum algorithm. FWALSH [56] is CUDA implementation of generalized class of Fourier transform, also known as Walsh transform. HIST [56] is a histogram implementation on GPU. SORTNW [56] implements bitonic sort, a special type of sorting algorithm.

REDUCE [56] performs parallel reduction operations on an input array to produce a final value. PSUM is a microbenchmark based on the *threadfence* example explained in the CUDA programming guide [56]. OFFT [56] is an ocean simulation based on fast Fourier transform. KMEANS [66] is a CUDA implementation of the parallel k-means clustering algorithm [73]. HASH is another microbenchmark where every thread updates a hash table atomically. REDUCE, PSUM, and HASH benchmarks use memory fencing functions in CUDA for inter-thread-block communication. The thread-blocks in other benchmarks work on independent sections in the shared and global memory spaces. All benchmarks are run until completion for detailed evaluation.

6.5 Experimental Results

In this section, we evaluate the overall performance of HAccRG. We start with a discussion on the effectiveness of HAccRG in detecting data races and the sensitivity of our results to different hardware and software configurations. We then evaluate various aspects of the performance overhead in HAccRG.

6.5.1 Effectiveness of Data Race Detection

We evaluate the effectiveness of HAccRG in detecting data races for both the shared and global memory accesses in the benchmarks listed in Table 6.2. For this purpose, we track the shared and global memory accesses at the word granularities. Overall, we report four categories of data races: i) races in the shared memory due to incorrect barrier synchronizations; ii) races in the global memory due to incorrect barrier synchronizations; iii) races in the global memory due to the lack of critical sections; and iv) races in the global memory due to missing memory fence instructions. No data race is detected in the shared memory; however, data races are found in the global memory for three benchmarks. In SCAN and KMEANS, the kernels are designed to execute as a single thread-block, but

multiple thread-blocks are launched to scale up the workload. Consequently, all thread-blocks operate on the same data, causing data dependences that otherwise would not exist. These bugs are documented by the developers of the benchmark suite. No data race is reported when both SCAN and KMEANS are executed with a single thread-block. In the spectrum generation kernel of OFFT, the input is processed to create a wave spectrum in frequency domain. Multiple thread-blocks process separate data and update the final result. However, the memory address is incorrectly calculated, and two threads accessed the same memory location, causing a write-after-read data race in the global memory space.

Injected Races: To identify the effectiveness of HAccRG in benchmarks that did not show data races, we inject artificial data races in the benchmarks for shared and global memory spaces to verify how well HAccRG performs in detecting them. Data races are injected by removing barrier calls from the benchmarks (23 races), by inserting dummy memory accesses across the thread-block access boundaries (13 races), by removing memory fence calls (3 races), and by inserting dummy memory accesses inside and outside the critical sections (2 races). HAccRG is able to detect all the forty-one injected data races.

Impact of Varying Memory Access Tracking Granularity

Table 6.3 shows the number of false shared memory data races detected when shared memory tracking granularity is varied from 4-byte to 64-byte. HAccRG does not detect false data races for five out of nine benchmarks because of their peculiar shared memory access patterns. They all have very regular accesses to the shared memory where all threads in a warp access successive locations. Accesses within a warp always have implicit synchronization because of SIMD execution. Therefore, HAccRG does not report a data race even when the entire warp's accesses map to a single shadow entry. On the other hand, HAccRG reports high number of false data races for HIST since the benchmark operates on a data structure having element size of one byte, which in turn translates to accesses from multiple warps mapping to the same shadow entries.

Benchmark	# False Races									
	Shared Memory					Global Memory				
	4	8	16	32	64	4	8	16	32	64
MCARLO	0	0	0	0	0	0	0	2	2	2
SCAN	0	0	2	7	7	0	0	0	0	0
FWALSH	0	0	0	0	0	0	0	0	0	0
HIST	32	32	32	32	32	0	2	2	2	2
SORTNW	0	0	0	0	0	0	0	0	0	0
REDUCE	0	1	1	1	1	0	2	2	2	2
PSUM	0	0	0	0	0	0	0	1	1	1
OFFT	0	0	0	0	0	0	0	5	6	7
KMEANS	0	0	0	0	3	0	0	0	0	0
HASH	0	0	0	0	0	0	2	2	2	2

Table 6.3: Number of false data races detected when the shared and global memory access tracking granularities are varied. 4, 8, 16, 32, and 64 indicate the granularities in bytes.

Table 6.3 also reports the number of false global memory data races detected when the global memory tracking granularity is varied. None of the benchmarks have false data race detection for 4-byte granularity since, for all the benchmarks, global memory data structure element sizes are at least 4 bytes. HAccRG does not detect any false data races for four out of nine benchmarks until 64-byte granularity due to very regular memory accesses.

To summarize, the impact of access tracking granularity is application-dependent. However, varying the granularity changes the hardware requirements as well as the memory footprint of HAccRG. To reduce hardware overhead of shared memory shadow entries, we decrease the shared memory tracking granularity. We set it to 16 bytes since 7 out of 10 benchmarks do not see any false positives at this granularity. However, since device memory available to the contemporary GPUs is quite large, we keep the global memory tracking granularity to 4 bytes. The hardware and memory overheads of HAccRG are discussed in Section 6.5.3.

Impact of Tracking Only Most Recent Accesses

In this section we quantify the effect of limited number of global memory shadow entries on data race detection. We achieve this by counting the number of race detection checks performed at runtime with limited amount of shadow memory, and comparing them to the

number of checks performed when entire shadow memory is available at the finest tracking granularity. The shadow memory page size is set to 4KB for this experiment, and all global memory accesses are tracked at the finest granularity. Figure 6.7 shows that most of the benchmarks can tolerate much lower amount of shadow memory without missing significant number of race detection checks. There are two reasons for this: first, some data structures are accessed only once, hence, overwriting their access information is harmless; and second, some programs work on a chunk of data and then move on to the next chunk. If such chunks fit in the available shadow memory, overwriting their access information with new chunks often does not reflect as missed data race checks. On average, only 5% data races detection opportunities are missed when 50% of the shadow memory is available. This shows that when limited amount of shadow memory is available, HAccRG is still able to detect most of the data races without reporting false positives.

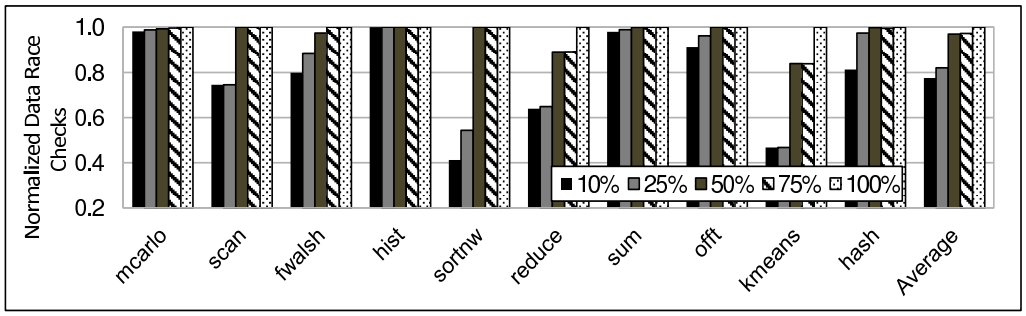


Figure 6.7: Number of data race checks performed at runtime when $X\%$ of shadow memory is available. Each bar is normalized to the number of data race checks performed when entire shadow memory is available at the finest granularity (X=100%).

Impact of Sizes of Sync, Fence, and Atomic IDs

Sync and fence IDs act as logical clocks for barrier and fence calls, respectively, while atomic IDs track the lock variables used in kernels. Sync and fence IDs are counters which

are incremented during execution. They can detect false positives when counters reset after overflow, and the overflowed values match the IDs in the shadow entries. However, we believe that such occurrences are very rare in actual execution when sufficiently large counters are provided. Moreover, optimization to increment sync ID only when global memory accesses are made, effectively limits the number of increments. We observe that sync ID increments are very small (maximum 5 for REDUCE) because barrier synchronizations are primarily used only for shared memory accesses in the benchmarks. Although fence ID is incremented when a warp completes a fence call, the number of fences executed is small, maximum being 5 for HASH. Therefore, we set sync and fence ID sizes to 8 bits each which are large enough to avoid overflows.

Atomic IDs, which are Bloom filter signatures, hold the addresses of lock variables protecting critical sections. Since signatures are compact representation of addresses, two different addresses can form the same signature, which is common in signature-based systems. Therefore, HAccRG can miss the actual data races if it is not able to distinguish between different lock variables. We perform a stress test on a microbenchmark by injecting data races for over 1 million addresses to study impact on accuracy. We test 8-bit, 16-bit, and 32-bit signatures with 2 and 4 bins each. Bits in each bin are set through direct indexing by lower order bits of the addresses [79]. We observe that signatures with 2 bins have better accuracy than those with 4 bins for the same signature size. 8-bit, 16-bit, and 32-bit signatures with 2 bins miss 25%, 12.5%, and 6.25% data races, respectively. To trade-off between hardware cost and accuracy, we set the atomic ID size to 16 bits in HAccRG.

6.5.2 Performance Impact of Race Detection

Figure 6.8 shows the performance impact of enabling HAccRG. The bars represent execution time normalized to that of benchmarks executing on unmodified GPU where HAccRG is disabled. The geometric mean across all benchmarks is shown at the end of Figure 6.8.

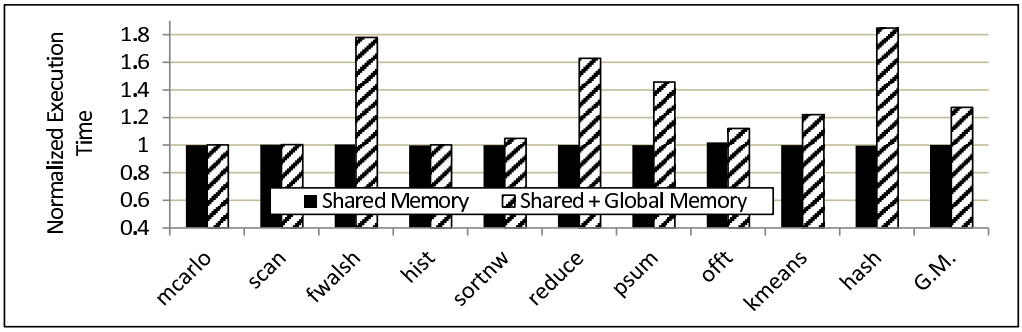


Figure 6.8: Performance impact of HAccRG. The bars are normalized to the execution time when HAccRG is disabled.

In Figure 6.8, the first set of bars indicated by *Shared Memory* represents the performance impact of shared memory data race detection, while the other set indicated by *Shared + Global Memory* shows the impact of detecting both shared and global memory data races. For shared memory data race detection, since the shadow entries in hardware are analyzed and updated concurrently with the corresponding shared memory accesses, the performance overhead is only about 1% on an average. On the other hand, global memory data race detection causes higher runtime overhead as shown in figure. This is due to slower accesses to the global memory shadow entries in the L2 cache or DRAM, which in turn delay the regular global memory accesses from SMs. Overall, for shared memory data race detection, HAccRG incurs near zero runtime overhead, while combining the global memory data race detection with the shared memory data race detection increases the execution time by 27% on average.

Performance Comparison with Software Implementations:

In this section, we evaluate the importance of hardware support by comparing HAccRG with a software-based implementation of data race detection that we refer to as Software Race detection for GPUs (SRG). SRG is implemented by explicitly maintaining a shadow memory with software and instrumenting all memory accesses and synchronizations to manage the shadow memory entries. The shadow memory entries for the shared memory

space are located in the faster on-chip shared memory, while the shadow memory entries for the global memory space are located in the slower device memory. Due to the development costs associated with manual implementation, we only evaluated SRG on three benchmarks. It is worth pointing out that depending on how the benchmark utilizes the shared memory, the SRG may allocate the shadow memory entries for the shared memory in the larger global memory space.

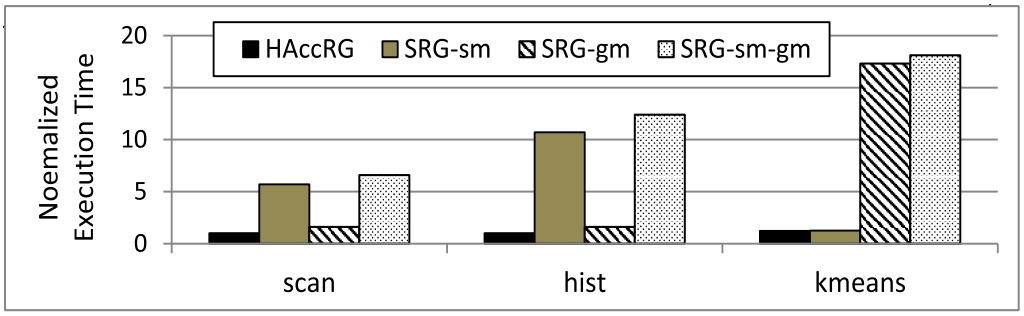


Figure 6.9: Execution time normalized to execution without data race detection. *SRG* indicates software implementation of HAccRG, while *sm* and *gm* mean shared and global memory data race detection, respectively.

Figure 6.9 shows the comparison between HAccRG and SRG. SRG incurs a significant overhead due to the execution of these instrumentation instructions. HAccRG, with hardware support, is able to avoid this overhead. Furthermore, since the data race detection units in HAccRG reside in the memory hierarchy, we are able to further reduce the performance overhead by avoiding shadow memory movement between the SMs and the device memory when detecting data races in the global memory. HAccRG incurs a 0.2%, 0.3%, and 22.1% slowdown for Scan, Hist, and Kmeans, respectively, while SRG incurs a slowdown of 6.6x, 12.4x, and 18.1x for these three benchmarks. The performance penalty of race detection is application dependent. In particular, the performance penalties for Scan and Hist are small compared to Kmeans because these two benchmarks make intensive use of the shared memory rather than the global memory. Thus, these two benchmarks are able to take full advantage of the additional hardware support for shared memory data

race detection.

It is worth pointing out that SRG represents an efficient implementation of the instrumentation-based race detection mechanism. A prior instrumentation-based mechanism, GRace[1] [77], maintains two tables per warp and two tables per thread-block for keeping track of the number of read and write accesses to each shared memory location, by each warp and by all warps in a thread-block combined. For each access to a shared memory location, GRace updates two tables: the read/write table of the warp and the read/write table of the thread-block. Even though GRace only supports data race detection in the shared memory, it is two orders of magnitude slower than SRG. Furthermore, in GRace, the memory overhead for the three benchmarks were 72MB, 102MB, and 20MB for SCAN, HIST and KMEANS, respectively; while the memory overhead for SRG is only 480KB (16KB per SM * 30 SMs). Thus, it is possible for us to allocate the shadow memory entries in the shared memory to further improve performance in SRG. GRace attempted to address this performance degradation by utilizing software analysis to reduce the number of instrumented instructions. The limitation of their approach is addressed in Section 6.6.

Performance Impact of Allocating Shared Memory Shadow Entries in Global Memory:

In HAccRG, the shared memory shadow entries are most significant in size, and consume the most static and dynamic power. When HAccRG is adopted for GPUs in mobile devices, it is likely to add pressure on to the already stringent power budget. We can split shared memory shadow entries between hardware and software to minimize their hardware overhead, at the expense of additional performance overhead. Figure 6.10 shows performance impact in HAccRG when the shared memory shadow entries are split between hardware and software. For this experiment, we enable both shared and global memory data race detection. Software shared memory shadow entries are stored in the global memory and fetched into the L1 data caches for data race detection using shared memory RDUs. In our

[1]GRace has two versions, *GRace-stmt* and *GRace-addr*; the former one is more accurate but slow. We compare HAccRG with faster but less accurate GRace-addr.

simulations, L1 data cache is 48KB per SM which is large enough to accommodate shadow memory for 16KB of shared memory. Therefore, most of the benchmarks do not show significant performance degradation, except OFFT. Since shared memory is banked, accesses to different shared memory rows across different banks are served in parallel. However, for data race detection, this can lead to reading multiple lines from the software shadow memory. OFFT suffers from this behavior as single shared memory access translates to multiple shadow memory lines read from the global memory. Thus, if hardware support in the shared memory is not possible, for most kernels, placing the shared memory shadow entries in the global memory incurs only a small performance penalty.

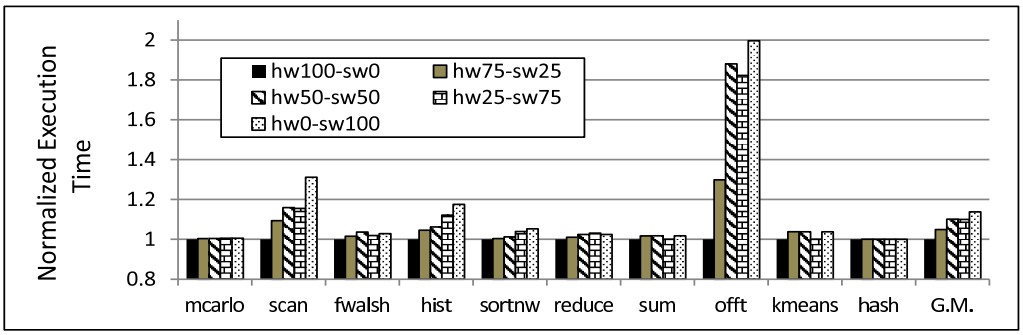


Figure 6.10: Performance impact of splitting shared memory shadow entries between hardware and software. hwX-swY means X% hardware and Y% software shared memory shadow entries. Each bar is normalized to the execution time of hw100-sw0.

6.5.3 Overheads in HAccRG

HAccRG tracks and records the shared and global memory accesses, and inspects those records to identify potential data races. In this section, we discuss impact of these data race detection tasks on the DRAM bandwidth utilization, the hardware requirements, and the memory requirements of a GPU.

DRAM Bandwidth Utilization

Figure 6.11 shows per-benchmark average bandwidth utilization reported by the simulator. Y-axis represents the average bandwidth utilization of all DRAM banks over the entire execution. The first bar of each benchmark shows the bandwidth utilization for execution without HAccRG. It is evident from Figure 6.11 that the DRAM bandwidth utilization is the characteristic of an application, which is dependent on the cache miss rate of L1 data caches and unified L2 cache. The benchmarks which have low L1 or L2 cache miss rate result in much lesser DRAM bandwidth utilization.

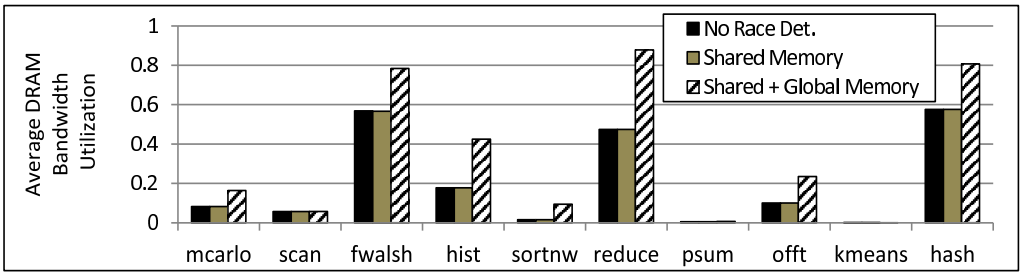


Figure 6.11: Average DRAM bandwidth utilization with and without HAccRG enabled.

Shared memory data race detection does not create memory requests, which is seen as the unchanged bandwidth utilization for all benchmarks. On the other hand, global memory data race detection accesses shadow entries that reside either in L2 cache or DRAM. Therefore, Mcarlo, Fwalsh, Hist, Sortnw, Reduce, Offt, and Hash have more bandwidth utilization as they rely on L2 cache performance due to their characteristic higher L1 miss rates. Enabling global memory data race detection degrades the L2 cache performance since the shadow entries pollute the L2 cache, thus increasing the DRAM utilization. However, Scan, Psum, and Kmeans do not depend upon the L2 cache performance because of their higher L1 hit rates. With their low L2 utilization, enabling global memory data race detection hardly affects their L2 cache performance, which in turn keeps the DRAM utilization almost constant for these benchmarks as seen in Figure 6.11.

To summarize, the applications that effectively exploit the L1 data caches increase the DRAM bandwidth utilization only marginally, while the applications that rely on the L2 cache show higher bandwidth utilization for the global memory data race detection. However, the overall bandwidth utilization is well within the DRAM limits.

Hardware Overhead

HAccRG requires additional control logic and storage to perform data race detection in GPU.

Control Logic: Fences and atomic operations are evaluated only for the global memory in this work. Shared memory shadow entries require 12-bit (1-bit modified, 1-bit shared, and 10-bit tid) comparator. For parallel comparison across shared memory banks at 16-byte granularity, HAccRG requires 8 12-bit comparators per SM. Global memory shadow entries are configurable and their size can be 28 bits (1-bit modified, 1-bit shared, 10-bit tid, 3-bit bid, 5-bit sid, 8-bit sync ID). With fence ID (8 bits) or atomic ID (16 bits) added, the size of shadow entries can be 36 bits or 52 bits, respectively. To summarize, for a cache line size of 128 bytes at 4-byte granularity, we need 32 28-bit comparators for basic shadow entries and 16 24-bit comparators for fence and atomic IDs per memory partition.

Storage: For the shared memory data race detection, HAccRG employs 12-bit shadow entries at the granularity of 16 bytes. The recent NVIDIA Fermi GPUs can have up to 48KB of shared memory per SM [57]. HAccRG will require 4.5KB storage per SM on Fermi for the shared memory shadow entries. For the global memory data race detection, each SM maintains a per-block 8-bit sync ID, a per-warp 8-bit fence ID, and a per-thread 16-bit atomic ID. For a single Fermi SM supporting 8 concurrent blocks, 48 warps, and 1536 threads, the storage size for global memory data race detection will be 3KB per SM. The race register file replicated in each memory partition takes 0.75KB per copy.

Overall, for NVIDIA Fermi GPUs, the additional hardware will cost less than 1% of the total chip area. Therefore, we do not expect significant increase in power consumption

with the data race detection support. Also, the proposed changes should not affect the GPU clock period; however, the global memory accesses can suffer from increased latency caused by the additional shadow memory accesses.

Benchmark	Shadow Memory Overhead	Benchmark	Shadow Memory Overhead
MCARLO	288KB	SCAN	9KB
FWALSH	4.7MB	HIST	27.9MB
SORTNW	576KB	REDUCE	6.7MB
PSUM	208KB	OFFT	1.2MB
KMEANS	9.5KB	HASH	4.6MB

Table 6.4: Global memory overhead of HAccRG.

Memory Overhead

HAccRG requires fixed space to store global memory shadow entries. This overhead at the granularity of 4 bytes is shown in Table 6.4. By changing the global memory tracking granularity or by tracking only most recent memory accesses, HAccRG can significantly reduce the overhead of shadow entries by trading off the data race detection accuracy.

6.6 Related Works

Automatic code generation for GPUs can potentially reduce the programming complexities, thus minimizing the chances of introducing data races. Hou et al. [33] propose bulk-synchronous GPU programming (BSGP) language for faster development of applications on GPUs. The BSGP programs are simple to write and maintain. Programmers can also specify barriers in the code for synchronization. However, more complex fine-grained synchronizations such as locks and fences are not supported in BSGP. Lee et al. [41] have proposed source-to-source translation of OpenMP applications to CUDA for automated creation of high performance CUDA kernels. However, since OpenMP applications are not written for GPU-like stream architectures, performance of the generated CUDA code

can suffer. Klöckner et al. [37] introduce scripting to generate GPU code at runtime for harvesting GPU's parallelism. None of the previous works on automatic code generation, however, have explicitly addressed issues related to correctness on GPUs. These proposals are prone to data races on GPUs if the inputs to them contain data races.

There have been some efforts initiated recently to ensure software correctness in GPUs [44], including data race detection [7, 34, 42, 43, 77]. These approaches mainly include software-based static analysis or dynamic monitoring of GPU programs. The static analysis approach inspects the program source code, and based on the knowledge of the GPU execution model, can either detect the static races or predict runtime races [43, 77]. This approach is limited since it cannot foresee the runtime program behaviors; thus, can result in significant false positive rate [43]. The dynamic monitoring approach adds software instrumentation to GPU programs, thus causes significant slowdown. However, it is more effective as the runtime program control flows and indirect memory accesses are exposed to the monitoring system.

Adequate hardware support can potentially improve the effectiveness and efficiency of data race detection. Such hardware support is responsible for tracking and comparing the memory accesses from all threads to detect data races. In this work, we design a hardware-accelerated data race detection framework for modern GPUs, referred to as HAccRG. To the best of our knowledge, HAccRG is the first hardware implementation for detecting data races in the GPU. It detects data races in both shared and global memory spaces of a GPU.

The underlying concept behind the proposed solution is to keep the associated overhead (performance as well as hardware) constant with respect to the number of memory accesses in a GPU program. The existing data race detection techniques often incur quadratic number of data race comparisons, relative to the number of threads, as well as have large overhead for tracking per-thread memory accesses. They cannot scale to GPUs that normally support thousands of threads simultaneously. Therefore, HAccRG employs

a per-memory access tracking mechanism, instead of tracking per-thread accesses. This method allows us to design a fast hardware data race detector with moderate hardware overhead.

The proposed hardware support covers both the shared and global memory spaces of a GPU. We implement HAccRG on a cycle accurate general purpose GPU (GPGPU) simulator and evaluate its effectiveness using a set of CUDA benchmarks. Our evaluation shows that HAccRG is able to accurately detect real and injected data races in GPU programs with an average performance overhead of only 1% for the shared memory and 27% for combined shared and global memory data race detection. We also compare HAccRG with its software implementation as well as with an existing instrumentation-based runtime data race detection mechanism called GRace [77]. Our experience shows that hardware can accelerate data race detection by order of magnitude than the software approaches. Furthermore, the software implementation of HAccRG is more efficient than GRace in terms of both performance as well as space overhead.

6.7 Summary

In this work, we propose a hardware-accelerated mechanism (HAccRG), for efficient and accurate data race detection in GPUs. We implement HAccRG support in the shared and global memory of the GPU with moderate hardware overhead. We evaluate the effectiveness and efficiency of the proposed technique using a set of GPGPU benchmarks. HAccRG accurately detects data race bugs in GPU programs with the average runtime overhead of 1% for the shared memory and 27% for combined shared and global memory. Furthermore, we show that the software implementation of HAccRG is two orders of magnitude faster than the previously proposed data race detection technique on GPU. We have demonstrated that it is feasible to devote reasonable hardware to facilitate the design and implementation of an efficient data race detection mechanism in GPU. Such a data race detection mechanism can form the basis of powerful tools for easing parallel software

development and enhancing software correctness.

Chapter 7

Conclusions and Future Directions

As energy became a first order constraint, microprocessors have evolved from simple single-issue in-order cores to modern superscalar out-of-order processors. Along with this evolution, several alternate architectures have emerged to address diverse workloads. These architectures fall under the category of accelerators and/or co-processors that aid the CPU by offloading certain tasks, and thus improving overall performance. The GPU is one such accelerator that renders graphics as well as is able to execute data-parallel compute workloads. Because of their versatility, GPUs have become an integral part of today's computing systems. Therefore, it is important to improve programmability of GPUs for enhancing programmers' productivity as well as software reliability. In this dissertation, we have addressed these challenges by proposing transactional execution and data race detection supports.

We begin our work by identifying issues that arise while programming irregular applications on GPUs. Irregular applications exhibit dynamic data sharing among application threads, which often requires lock-based synchronizations for functionally correct execution. A large number of GPU threads makes such complex lock implementations prone to concurrency bugs. Furthermore, CPU multithreaded programming practices can create deadlocks and/or livelocks on GPUs. To make things worse, weak memory consistency in

GPUs can cause data races that are not observed in CPU-based systems. These challenges make it difficult to write irregular applications for GPUs.

To address the aforementioned challenges, we propose transactional execution for irregular applications on GPUs. By transactionalizing critical sections, the programming efforts are reduced to that of coarse-grained locking. The programmer just has to mark the critical sections, while the rest of work of maintaining consistent memory state is done by the transactional memory. To enable this support on off-the-shelf GPUs, we propose a software implementation of transactional memory (STM). Our STM support scales with the large number of GPU threads and is able to achieve performance comparable to fine-grained locking, even beating it under high contention. To reduce transactional overheads, we further propose hardware transactional memory on GPUs. Our experiments show that, for irregular data-parallel applications, the proposed transactional memory designs are able to outperform execution on multicore CPUs. As GPUs emerge as a novel computing engine to drive the performance of future parallel workloads, our experience demonstrates that supporting transactional execution can help us achieve this goal.

We then move to the correctness aspect of GPU kernels. It is challenging to write functionally correct GPU applications consisting of thousands of threads. Programmers' inexperience with the GPU execution model or sheer negligence can introduce concurrency bugs, such as data races, in GPU applications. Nonetheless, debugging data races is extremely difficult and time consuming on a GPU platform. A large body of work has explored numerous avenues for addressing data race issues on CPUs. However, these works are not scalable and often inadequate for modern GPUs that support thousands of concurrent threads.

In this dissertation, we propose hardware-accelerated data race detection support (HAccRG) for GPUs. HAccRG provides a framework for detecting a multitude of data races that can occur in GPU-based systems. We implement HAccRG on both shared and global memory spaces of GPUs. With the help of a set of GPGPU workloads, we demonstrate that

the proposed hardware support can accurately detect data races in GPU. Average runtime overhead introduced by HAccRG is 1% for shared memory and 27% for combined shared and global memory. Furthermore, we compare the software implementation of HAccRG with previously proposed software data race detection schemes for GPUs. Our evaluation shows that our mechanism is an order of magnitude faster than earlier works, and has a considerably smaller memory footprint. Such a data race detection mechanism can form the basis of powerful tools for easing parallel software development and enhancing software correctness.

7.1 Future Work

With the advent of new architectures such as GPUs, it is important to equip programmers with tools for enabling efficient programming of these emerging architectures. With this motivation, this dissertation makes an effort for improving programmers' productivity on GPUs. There are several avenues to which this research can be extended in future.

7.1.1 Hardware-Software Co-Design of Transactional Memory

The transactional memory support proposed in this work is entirely implemented in either software or hardware. The software support adds runtime overhead, while the hardware support requires non-trivial architectural changes in the GPU. The newly added hardware features not only take valuable on-chip real estate, but also increase the design and verification efforts. A possible solution to this conundrum is hardware-software co-design of transactional memory. This can be achieved by implementing only a small section of transactional memory in hardware, while keeping the rest of it in software. From the evaluation of STM, we have seen that conflict detection consumes the most number of clock cycles among all STM operations. Therefore, a hardware-based conflict detection can dramatically improve performance of TM on GPU. Moreover, conflict detection is relatively simple to implement in hardware compared to other transactional operations. This makes

hardware-software co-design an interesting research direction for future work.

7.1.2 Compiler Support for Transactional Memory

We have proposed multiple TM flavors in this dissertation, and we rely on compiler support to select the best scheme among all. This can be achieved by extracting application characteristics before making a choice. Such characteristics might include memory access patterns, thread contention, and transaction sizes. However, avenues to obtain these characteristics remain unexplored. Furthermore, when it is difficult to get application characteristics through compiler analysis, other approaches must be researched. For example, a compiler could launch multiple kernels with different transaction schemes and choose the best one at runtime based on the phase behaviors. Another example where a compiler could help is collating multiple transactional operations on the same address into a single operation, which could reduce transactional overheads significantly. To summarize, a compiler for transactional memory plays an important role in TM's performance, and also opens doors for further optimizations and opportunities. Thus, compiler support becomes an interesting research direction for future work.

7.1.3 Unified Transactional Memory and Data Race Detection Support

In this dissertation, we have presented transactional memory [32] and data race detection [31] as two mechanisms for improving programmers' productivity on GPUs. However, both mechanisms have an overlapping aspect, which is detecting dependence violations. Upon detecting a dependence violation transactional memory aborts the transaction, while a data race detector reports a data race. Therefore, consolidating such a mechanism into a unified hardware unit will save expensive on-chip real estate. This also complements the hardware-software co-design proposal for transactional memory we discussed earlier.

References

[1] http://www.cs.wisc.edu/trans-memory/biblio/index.html.

[2] Advanced Micro Devices. The AMD Fusion Family of APUs. http://fusion.amd.com/.

[3] C. Scott Ananian, Krste Asanovic, Bradley C. Kuszmaul, Charles E. Leiserson, and Sean Lie. Unbounded transactional memory. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, HPCA '05, pages 316–327, Washington, DC, USA, 2005. IEEE Computer Society.

[4] Ali Bakhoda, George L. Yuan, Wilson W.L. Fung, Henry Wong, and Tor M. Aamodt. Analyzing cuda workloads using a detailed gpu simulator. In *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*, pages 163–174, April 2009.

[5] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, July 1970.

[6] Jayaram Bobba, Kevin E. Moore, Haris Volos, Luke Yen, Mark D. Hill, Michael M. Swift, and David A. Wood. Performance pathologies in hardware transactional memory. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, pages 81–91, New York, NY, USA, 2007. ACM.

[7] Michael Boyer, Kevin Skadron, and Westley Weimer. Automated Dynamic Analysis of CUDA Programs. In *Third Workshop on Software Tools for MultiCore Systems*, 2008.

[8] Martin Burtscher, Rupesh Nasre, and Keshav Pingali. A quantitative study of irregular programs on gpus. In *Proceedings of the 2012 IEEE International Symposium on Workload Characterization (IISWC)*, IISWC '12, pages 141–151, Washington, DC, USA, 2012. IEEE Computer Society.

[9] Daniel Cederman, Philippas Tsigas, and Muhammad Tayyab Chaudhry. Towards a software transactional memory for graphics processors. In *Proceedings of the 10th Eurographics Conference on Parallel Graphics and Visualization*, EG PGV'10, pages 121–129, Aire-la-Ville, Switzerland, Switzerland, 2010. Eurographics Association.

[10] Luis Ceze, James Tuck, Josep Torrellas, and Calin Cascaval. Bulk disambiguation of speculative threads in multiprocessors. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture*, ISCA '06, pages 227–238, Washington, DC, USA, 2006. IEEE Computer Society.

[11] Hassan Chafi, Jared Casper, Brian D. Carlstrom, Austen McDonald, Chi Cao Minh, Woongki Baek, Christos Kozyrakis, and Kunle Olukotun. A scalable, non-blocking approach to transactional memory. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, HPCA '07, pages 97–108, Washington, DC, USA, 2007. IEEE Computer Society.

[12] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, IISWC '09, pages 44–54, Washington, DC, USA, 2009. IEEE Computer Society.

[13] Weihaw Chuang, Satish Narayanasamy, Ganesh Venkatesh, Jack Sampson, Michael Van Biesbrouck, Gilles Pokam, Brad Calder, and Osvaldo Colavin. Unbounded page-based transactional memory. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, pages 347–358, New York, NY, USA, 2006. ACM.

[14] JaeWoong Chung, Chi Cao Minh, Austen McDonald, Travis Skare, Hassan Chafi, Brian D. Carlstrom, Christos Kozyrakis, and Kunle Olukotun. Tradeoffs in transactional memory virtualization. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, pages 371–381, New York, NY, USA, 2006. ACM.

[15] Luke Dalessandro, Michael F. Spear, and Michael L. Scott. Norec: Streamlining stm by abolishing ownership records. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 67–78, New York, NY, USA, 2010. ACM.

[16] Peter Damron, Alexandra Fedorova, Yossi Lev, Victor Luchangco, Mark Moir, and Daniel Nussbaum. Hybrid transactional memory. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, pages 336–346, New York, NY, USA, 2006. ACM.

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

[18] Sérgio Miguel Fernandes and João Cachopo. Lock-free and scalable multi-version software transactional memory. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP '11, pages 179–188, New York, NY, USA, 2011. ACM.

[19] Fujitsu Corporation. SPARC64 V Processor Whitepaper For UNIX Server, 2004. http://www.fujitsu.com.

[20] Wilson W. L. Fung and Tor M. Aamodt. Thread block compaction for efficient simt control flow. In *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture*, HPCA '11, pages 25–36, Washington, DC, USA, 2011. IEEE Computer Society.

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

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

[23] Rachid Guerraoui and Michal Kapalka. On the correctness of transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, pages 175–184, New York, NY, USA, 2008. ACM.

[24] Andreas W. Gtz, Mark J. Williamson, Dong Xu, Duncan Poole, Scott Le Grand, and Ross C. Walker. Routine microsecond molecular dynamics simulations with amber on gpus. 1. generalized born. *Journal of Chemical Theory and Computation*, 8(5):1542–1555, 2012. PMID: 22582031.

[25] Lance Hammond, Vicky Wong, Mike Chen, Brian D. Carlstrom, John D. Davis, Ben Hertzberg, Manohar K. Prabhu, Honggo Wijaya, Christos Kozyrakis, and Kunle Olukotun. Transactional memory coherence and consistency. In *Proceedings of the*

31st Annual International Symposium on Computer Architecture, ISCA '04, pages 102–, Washington, DC, USA, 2004. IEEE Computer Society.

[26] Tim Harris and Keir Fraser. Language support for lightweight transactions. In *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programing, Systems, Languages, and Applications*, OOPSLA '03, pages 388–402, New York, NY, USA, 2003. ACM.

[27] Tim Harris, James Larus, and Ravi Rajwar. *Transactional Memory, 2nd Edition*. Morgan and Claypool Publishers, 2nd edition, 2010.

[28] Tim Harris, Mark Plesko, Avraham Shinnar, and David Tarditi. Optimizing memory transactions. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pages 14–25, New York, NY, USA, 2006. ACM.

[29] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer, III. Software transactional memory for dynamic-sized data structures. In *Proceedings of the Twenty-second Annual Symposium on Principles of Distributed Computing*, PODC '03, pages 92–101, New York, NY, USA, 2003. ACM.

[30] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, ISCA '93, pages 289–300, New York, NY, USA, 1993. ACM.

[31] Anup Holey, Vineeth Mekkat, and Antonia Zhai. Haccrg: Hardware-accelerated data race detection in gpus. In *42nd International Conference on Parallel Processing (ICPP)*, pages 60–69, Oct 2013.

[32] Anup Holey and Antonia Zhai. Lightweight software transactions on gpus. In *43rd International Conference on Parallel Processing (ICPP)*, pages 461–470, Sept 2014.

[33] Qiming Hou, Kun Zhou, and Baining Guo. Bsgp: Bulk-synchronous gpu programming. In *ACM SIGGRAPH 2008 Papers*, SIGGRAPH '08, pages 19:1–19:12, New York, NY, USA, 2008. ACM.

[34] Qiming Hou, Kun Zhou, and Baining Guo. Debugging gpu stream programs through automatic dataflow recording and visualization. In *ACM SIGGRAPH Asia 2009 Papers*, SIGGRAPH Asia '09, pages 153:1–153:11, New York, NY, USA, 2009. ACM.

[35] Intel Inc. Intel Sandy Bridge Processor Family. http://www.intel.com/.

[36] Khronos Group. OpenCL - The open standard for parallel programming of heterogeneous systems. http://www.khronos.org/opencl/.

[37] Andreas Klöckner, Nicolas Pinto, Yunsup Lee, Bryan Catanzaro, Paul Ivanov, and Ahmed Fasih. Pycuda and pyopencl: A scripting-based approach to gpu run-time code generation. *Parallel Comput.*, 38(3):157–174, March 2012.

[38] Milind Kulkarni, Martin Burtscher, Calin Casçaval, and Keshav Pingali. Lonestar: A suite of parallel irregular programs. In *ISPASS '09: IEEE International Symposium on Performance Analysis of Systems and Software*, 2009.

[39] Milind Kulkarni, Keshav Pingali, Bruce Walter, Ganesh Ramanarayanan, Kavita Bala, and L. Paul Chew. Optimistic parallelism requires abstractions. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 211–222, New York, NY, USA, 2007. ACM.

[40] Sanjeev Kumar, Michael Chu, Christopher J. Hughes, Partha Kundu, and Anthony Nguyen. Hybrid transactional memory. In *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '06, pages 209–220, New York, NY, USA, 2006. ACM.

[41] Seyong Lee, Seung-Jai Min, and Rudolf Eigenmann. Openmp to gpgpu: A compiler framework for automatic translation and optimization. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '09, pages 101–110, New York, NY, USA, 2009. ACM.

[42] Alan Leung, Manish Gupta, Yuvraj Agarwal, Rajesh Gupta, Ranjit Jhala, and Sorin Lerner. Verifying gpu kernels by test amplification. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 383–394, New York, NY, USA, 2012. ACM.

[43] Guodong Li and Ganesh Gopalakrishnan. Scalable smt-based verification of gpu kernel functions. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10, pages 187–196, New York, NY, USA, 2010. ACM.

[44] Guodong Li, Peng Li, Geof Sawaya, Ganesh Gopalakrishnan, Indradeep Ghosh, and Sreeranga P. Rajan. Gklee: Concolic verification and test generation for gpus. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 215–224, New York, NY, USA, 2012. ACM.

[45] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIII, pages 329–339, New York, NY, USA, 2008. ACM.

[46] Marc Lupon, Grigorios Magklis, and Antonio Gonzalez. Fastm: A log-based hardware transactional memory with fast abort recovery. In *Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques*, PACT '09, pages 293–302, Washington, DC, USA, 2009. IEEE Computer Society.

[47] Peter S. Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hållberg, Johan Högberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. Simics: A full system simulation platform. *Computer*, 35(2):50–58, February 2002.

[48] Virendra J. Marathe, William N. Scherer, and Michael L. Scott. Adaptive software transactional memory. In *Proceedings of the 19th International Conference on Distributed Computing*, DISC'05, pages 354–368, Berlin, Heidelberg, 2005. Springer-Verlag.

[49] Virendra Jayant Marathe and Mark Moir. Toward high performance nonblocking software transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, pages 227–236, New York, NY, USA, 2008. ACM.

[50] Milo M. K. Martin, Daniel J. Sorin, Bradford M. Beckmann, Michael R. Marty, Min Xu, Alaa R. Alameldeen, Kevin E. Moore, Mark D. Hill, and David A. Wood. Multifacet's general execution-driven multiprocessor simulator (gems) toolset. *SIGARCH Comput. Archit. News*, 33(4):92–99, November 2005.

[51] Sang L. Min and Jong-Deok Choi. An efficient cache-based access anomaly detection scheme. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS IV, pages 235–244, New York, NY, USA, 1991. ACM.

[52] Abdullah Muzahid, Dario Suárez, Shanxiang Qi, and Josep Torrellas. Sigrace: Signature-based data race detection. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 337–348, New York, NY, USA, 2009. ACM.

[53] Rupesh Nasre, Martin Burtscher, and Keshav Pingali. Atomic-free irregular computations on gpus. In *Proceedings of the 6th Workshop on General Purpose Processor*

Using Graphics Processing Units, GPGPU-6, pages 96–107, New York, NY, USA, 2013. ACM.

[54] Rupesh Nasre, Martin Burtscher, and Keshav Pingali. Morph algorithms on gpus. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, pages 147–156, New York, NY, USA, 2013. ACM.

[55] NVIDIA Corporation. http://www.nvidia.com/cuda.

[56] NVIDIA Corporation. NVIDIA CUDA C Programming Guide. http://www.nvidia.com.

[57] NVIDIA Corporation. NVIDIA's Next Generation CUDA Compute Architecture: Fermi. http://www.nvidia.com.

[58] NVIDIA Corporation. GeForce GTX 400 Architecture, 2010.

[59] Milos Prvulovic. Cord: cost-effective (and nearly overhead-free) order-recording and data race detection. In *High-Performance Computer Architecture, 2006. The Twelfth International Symposium on*, pages 232–243, Feb 2006.

[60] Milos Prvulovic and Josep Torrellas. Reenact: Using thread-level speculation mechanisms to debug data races in multithreaded codes. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, ISCA '03, pages 110–121, New York, NY, USA, 2003. ACM.

[61] Shanxiang Qi, Norimasa Otsuki, Lois O. Nogueira, Abdullah Muzahid, and Josep Torrellas. Pacman: Tolerating asymmetric data races with unintrusive hardware. In *High Performance Computer Architecture (HPCA), 2012 IEEE 18th International Symposium on*, pages 1–12, Feb 2012.

[62] Ravi Rajwar and James R. Goodman. Transactional lock-free execution of lock-based programs. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS X, pages 5–17, New York, NY, USA, 2002. ACM.

[63] A. Ramamurthy. Towards scalar synchronization in simt architectures, 2011. Master's thesis, University of British Columbia.

[64] Bratin Saha, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Chi Cao Minh, and Benjamin Hertzberg. Mcrt-stm: A high performance software transactional memory system for a multi-core runtime. In *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '06, pages 187–197, New York, NY, USA, 2006. ACM.

[65] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15(4):391–411, November 1997.

[66] Serban Giuroiu. CUDA K-Means Clustering. http://serban.org/software/kmeans/.

[67] Nir Shavit and Dan Touitou. Software transactional memory. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '95, pages 204–213, New York, NY, USA, 1995. ACM.

[68] Michael F. Spear, Maged M. Michael, Michael L. Scott, and Peng Wu. Reducing memory ordering overheads in software transactional memory. In *Proceedings of the 7th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '09, pages 13–24, Washington, DC, USA, 2009. IEEE Computer Society.

[69] Michael F. Spear, Maged M. Michael, and Christoph von Praun. Ringstm: Scalable transactions with a single atomic instruction. In *Proceedings of the Twentieth Annual*

Symposium on Parallelism in Algorithms and Architectures, SPAA '08, pages 275–284, New York, NY, USA, 2008. ACM.

[70] Standard Performance Evaluation Corporation. The SPEC CPU2000 Benchmark Suite. http://www.specbench.org.

[71] John A. Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, vLi Wen Chang, Nasser Anssari, Geng Daniel Liu, and Wen mei W. Hwu. Impact technical report, 2012. University of Illinois, at Urbana-Champaign.

[72] Saša Tomić, Cristian Perfumo, Chinmay Kulkarni, Adrià Armejach, Adrián Cristal, Osman Unsal, Tim Harris, and Mateo Valero. Eazyhtm: Eager-lazy hardware transactional memory. In *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 145–155, New York, NY, USA, 2009. ACM.

[73] Wei-keng Liao. Parallel K-Means Data Clustering. http://users.eecs.northwestern.edu/∼wkliao/Kmeans/index.html.

[74] Yunlong Xu, Rui Wang, Nilanjan Goswami, Tao Li, Lan Gao, and Depei Qian. Software transactional memory for gpu architectures. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, pages 1:1–1:10, New York, NY, USA, 2014. ACM.

[75] Yunlong Xu, Rui Wang, Nilanjan Goswami, Tao Li, and Depei Qian. Software transactional memory for gpu architectures. *Computer Architecture Letters*, PP(99):1–1, 2013.

[76] Luke Yen, Jayaram Bobba, Michael R. Marty, Kevin E. Moore, Haris Volos, Mark D. Hill, Michael M. Swift, and David A. Wood. Logtm-se: Decoupling hardware transactional memory from caches. In *Proceedings of the 2007 IEEE 13th International*

Symposium on High Performance Computer Architecture, HPCA '07, pages 261–272, Washington, DC, USA, 2007. IEEE Computer Society.

[77] Mai Zheng, Vignesh T. Ravi, Feng Qin, and Gagan Agrawal. Grace: A low-overhead mechanism for detecting data races in gpu programs. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP '11, pages 135–146, New York, NY, USA, 2011. ACM.

[78] Mai Zheng, Vignesh T. Ravi, Feng Qin, and Gagan Agrawal. Gmrace: Detecting data races in gpu programs via a low-overhead scheme. *IEEE Trans. Parallel Distrib. Syst.*, 25(1):104–115, January 2014.

[79] Pin Zhou, Radu Teodorescu, and Yuanyuan Zhou. Hard: Hardware-assisted lockset-based race detection. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, HPCA '07, pages 121–132, Washington, DC, USA, 2007. IEEE Computer Society.