Improving GPU Programming Models Through Hardware Cache Coherence

by

Inderpreet Singh

B.A.Sc., The University of British Columbia, 2011

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF APPLIED SCIENCE

in

The Faculty of Graduate Studies

(Electrical and Computer Engineering)

THE UNIVERSITY OF BRITISH COLUMBIA

(Vancouver)

June 2013

© Inderpreet Singh 2013
Abstract

Graphics Processing Units (GPUs) have been shown to be effective at achieving large speedups over contemporary Chip Multiprocessors (CMPs) on massively parallel programs. The lack of well-defined GPU memory models, however, prevents support of high-level languages like C++ and Java, and negatively impacts programmability of GPUs. This thesis proposes to improve GPU programmability by adding support for a well-defined memory consistency model through hardware cache coherence. We show that GPU coherence introduces a new set of challenges different from that posed by scalable cache coherence for CMPs. First, introducing conventional directory coherence protocols adds unnecessary coherence traffic overhead to existing GPU applications. Second, the massively multithreaded GPU architecture presents significant storage overheads for buffering thousands of in-flight coherence requests. Third, these protocols increase the verification complexity of the GPU memory system.

Recent research, Library Cache Coherence (LCC), explored the use of time-based approaches in CMP coherence protocols. This thesis describes a time-based coherence framework for GPUs, called Temporal Coherence (TC), that exploits globally synchronized counters in single-chip systems to develop a streamlined GPU coherence protocol. Synchronized counters enable all coherence transitions, such as invalidation of cache lines, to happen synchronously, eliminating all coherence traffic and protocol races. We present two implementations of TC, called TC-Strong and TC-Weak. TC-Strong implements an optimized version of LCC, while TC-Weak uses a novel timestamp based memory fence mechanism to implement Release Consistency (RC) on GPUs. TC-Weak eliminates TC-Strong’s trade-off between stalling stores and increasing L1 miss rates to improve performance and reduce interconnect traffic. By providing coherent L1 caches, TC-Weak improves the performance of GPU applications requiring coherence by 85% over disabling the non-coherent L1 caches in the baseline GPU. We also show that write-through protocols outperform a writeback protocol on a GPU as the latter suffers from increased traffic due to unnecessary refills of write-once data.
Preface

The work described in this thesis is based upon the published work “Cache Coherence for GPU Architectures”, which appeared in the 19th IEEE International Symposium on High-Performance Computer Architecture (HPCA-19). The HPCA-19 work was co-authored by Inderpreet Singh, Dr. Arrvindh Shriraman, Wilson Fung, Mike O’Connor, and Dr. Tor Aamodt. The study and the novel protocols proposed in this work were designed collaboratively by Inderpreet Singh, Dr. Arrvindh Shriraman and Dr. Tor Aamodt. Inderpreet Singh, Dr. Arrvindh Shriraman, Wilson Fung and Dr. Tor Aamodt were responsible for the fusion of the GPU simulator with the cache-coherent memory system model that enabled this study. Inderpreet Singh performed the simulations, analyzed the results, and drafted the manuscript under the guidance of Dr. Arrvindh Shriraman and Dr. Tor Aamodt.
Table of Contents

Abstract ................................................................................................. ii
Preface .................................................................................................... iii
Table of Contents .................................................................................... iv
List of Tables ........................................................................................... vii
List of Figures ........................................................................................... viii
List of Abbreviations ............................................................................... ix
Acknowledgements .................................................................................. x

1 Introduction .......................................................................................... 1
  1.1 Motivation .......................................................................................... 2
  1.2 Contributions ...................................................................................... 4
  1.3 Thesis Organization ........................................................................... 5
2 Background ........................................................................................... 6
  2.1 Memory Consistency .......................................................................... 6
      2.1.1 Memory Consistency Models ...................................................... 6
      2.1.2 Types of Consistency Models .................................................... 8
      2.1.3 Which Consistency Model is the Best? ....................................... 9
  2.2 Cache Coherence ................................................................................. 9
      2.2.1 The Incoherence Problem .......................................................... 10
      2.2.2 Properties of Hardware Cache Coherence ................................. 11
      2.2.3 Coherence Implementation ....................................................... 14
      2.2.4 Directory Protocols ................................................................. 17
  2.3 Graphics Processing Units ............................................................... 22
      2.3.1 High-level GPU Architecture .................................................. 23
      2.3.2 GPU Memory System .............................................................. 24
      2.3.3 GPU Cache Hierarchy .............................................................. 25
# Table of Contents

2.3.4 Atomic Operation ........................................... 25

3 Challenges of GPU Coherence ................................. 26
  3.1 Coherence Traffic ............................................ 26
  3.2 Storage Requirements ......................................... 27
  3.3 Protocol Complexity ......................................... 28

4 Temporal Coherence ............................................ 30
  4.1 Time and Coherence ........................................... 30
  4.2 TC-Strong Coherence ......................................... 32
    4.2.1 TC-Strong Operation .................................... 33
    4.2.2 TC-Strong and LCC comparison ........................... 38
  4.3 TC-Weak Coherence ........................................... 39
    4.3.1 TC-Weak Operation ....................................... 40
  4.4 Lifetime Prediction .......................................... 43
  4.5 Timestamp Rollover ......................................... 44

5 Methodology .................................................... 45
  5.1 Simulation Infrastructure ..................................... 45
  5.2 Benchmarks .................................................. 46

6 Results ........................................................ 51
  6.1 Performance .................................................. 51
  6.2 Interconnect Traffic ......................................... 54
  6.3 Power .......................................................... 57
  6.4 TC-Weak vs. TC-Strong ....................................... 58
  6.5 TC-Strong Optimizations ..................................... 60
  6.6 TC-Weak Performance Profile ................................. 61
  6.7 Directory Size Scaling ...................................... 61
  6.8 TC-Weak Predictor Parameters ............................... 64
  6.9 Summary of Results .......................................... 64

7 Correctness .................................................... 65
  7.1 TC-Strong Correctness ....................................... 65
    7.1.1 TC-Strong Coherence Invariants ......................... 65
    7.1.2 Relaxed Consistency Implementation ..................... 67
  7.2 TC-Weak Correctness ........................................ 67
  7.3 Towards Full Verification .................................... 69
Table of Contents

8 Related Work ........................................... 70
  8.1 Timestamp-Based Software Coherence ............... 70
  8.2 Timestamp-Based Hardware Coherence ............... 71
  8.3 Self-Invalidation Based Coherence .................. 71
  8.4 Scalable Coherence .................................. 72
  8.5 Consistency Models for Throughput-Oriented Processors ... 72

9 Conclusion and Future Work .......................... 73
  9.1 Conclusions ......................................... 73
  9.2 Future Work ....................................... 74
    9.2.1 Lifetime Prediction ............................ 74
    9.2.2 Graphics Applications ......................... 74
    9.2.3 Cache Management ............................... 74
    9.2.4 CPU-GPU Coherence .............................. 75

Bibliography ............................................. 76
# List of Tables

2.1 Example program with inter-core communication. . . . . . . . 7
2.2 Possible executions of example program. . . . . . . . . . . . 7
2.3 Independent Read Independent Write example. . . . . . . . . . 13
2.4 L1 States for GPU-VI Protocol. . . . . . . . . . . . . . . . 19
2.5 L2 States for GPU-VI Protocol. . . . . . . . . . . . . . . . . 20
3.1 Comparison of protocol state counts. . . . . . . . . . . . . . . 28
4.1 L1 States for TC-Strong Protocol. . . . . . . . . . . . . . . . 36
4.2 L2 States for TC-Strong Protocol. . . . . . . . . . . . . . . . 37
4.3 L1 States for TC-Weak Protocol. . . . . . . . . . . . . . . . . 41
4.4 L2 States for TC-Weak Protocol. . . . . . . . . . . . . . . . . 42
5.1 Simulation Configuration. . . . . . . . . . . . . . . . . . . . . 47
5.2 Benchmarks. . . . . . . . . . . . . . . . . . . . . . . . . . . . 48
# List of Figures

1.1 Motivation for non-traditional coherence. .......................... 2

2.1 Incoherence example. .............................................. 10
2.2 Coherence example. ............................................... 12
2.3 Deadlock example. ................................................ 16
2.4 Baseline non-coherent GPU Architecture. ....................... 23

4.1 Coherence invalidation mechanisms. ............................... 31
4.2 Hardware extensions for TC-Strong and TC-Weak. ............... 33
4.3 Detailed Temporal Coherence operation. ........................... 34

6.1 Performance of coherent and non-coherent GPUs. ............... 52
6.2 Performance of DLB with varying interconnect latencies. .... 53
6.3 Interconnect traffic breakdown. ................................... 55
6.4 MESI interconnect traffic overheads. ............................ 56
6.5 Breakdown of interconnect power and energy. ................. 57
6.6 TC-Strong vs. TC-Weak with single timestamps. ............... 58
6.7 TC-Strong vs. TC-Weak with per-application timestamps. .... 59
6.8 TC-Strong optimization benefits. ................................ 60
6.9 Performance with different fixed lifetimes for TC-Weak. .... 61
6.10 Impact of scaling directory sizes and associativities. ....... 62
6.11 TC-Weak predictor performance with varying parameters. .... 63
List of Abbreviations

CMP  Chip Multiprocessor.
CTA  Cooperative Thread Array.
DSI  Dynamic Self-Invalidation.
GPU  Graphics Processing Unit.
GWCT Global Write Completion Time.
IRIW Independent Read Independent Write.
LCC  Library Cache Coherence.
MSHR Miss Status Holding Register.
RC  Release Consistency.
SC  Sequential Consistency.
SC for DRF Sequential Consistency for Data Race Free.
SIMD Single-Instruction Multiple-Data.
SIMT Single-Instruction Multiple-Thread.
SWMR Single-Writer Multiple-Reader.
TBE Transaction Buffer Entry.
TC  Temporal Coherence.
TSO  Total Store Order.
Acknowledgements

I am indebted to Professor Tor Aamodt for his guidance, support and encouragement throughout the last three years. I am grateful to Professor Arrvindh Shriraman for his help, guidance and insight, without which this thesis would not have been possible. I would also like to thank Wilson Fung for being a great teacher and for always having helpful insight to offer; the vastness of his knowledge is unsurpassed. I also thank all my colleagues in the computer architecture group for their help, support, and most of all, excellent company.

I wish to express my gratitude towards my loving family. I owe all my accomplishments to their support and sacrifices.

Finally, I am grateful to Natural Sciences and Engineering Research Council (NSERC) of Canada for providing financial support through the CGS-M award.
Chapter 1

Introduction

Graphics Processing Units (GPUs) were originally designed to render graphics for user interfaces, video games, and Computer Aided Design software. Today, GPUs have also become ubiquitous in high-throughput, general purpose computing. Recent trends show exponential growth in the number of GPU supercomputers in the list of top 500 most powerful supercomputers in the world [38]. This is because the high amount of compute power used to render millions of pixels in parallel translates well to data-parallel execution. Recent studies [40, 42, 63, 64] have shown that the large speedups attained by GPUs over Chip Multiprocessors (CMPs) are not just limited to regular parallelism; even highly irregular algorithms can attain significant speedups on a GPU. Furthermore, the inclusion of a multi-level cache hierarchy in recent GPUs [9, 68, 69] frees the programmer from the burden of software managed caches and further increases the GPU’s attractiveness as a platform for accelerating applications with irregular memory access patterns.

The proliferation of GPU computing has benefited considerably from the introduction of new programming models. GPU vendors introduced C-based programming interfaces like OpenCL [51] and NVIDIA CUDA [71] to ease GPU programming by abstracting away the Single-Instruction Multiple-Data (SIMD) hardware and providing the illusion of independent scalar threads executing in parallel. Despite the advances in GPU programming languages, a desire to support high-level languages deeply familiar to CPU programmers, such as C++ and Java, remains. Many recent projects [37, 41, 44, 65, 73] attempt to add limited form of C++ or Java support through libraries or source-to-source compilation, but remain far from supporting the full programming models.

Supporting high-level languages first requires support of well-defined memory consistency models. These consistency models form the basis of memory models for high-level languages [15, 58] and provide the synchronization primitives used by multithreaded applications. Today’s GPUs have ill-defined memory consistency models. For example, NVIDIA’s Fermi [68] and AMD’s Southern Islands [9] GPUs present problems for GPU applications where threads may communicate across different GPU cores. Hence,
1.1 Motivation

Cache coherence greatly simplifies supporting well-defined consistency and high-level language memory models on GPUs [60, 85]. Current GPUs [9, 68, 69] lack hardware cache coherence and require disabling of private caches if an application requires memory operations to be visible across all cores. GPU coherence is also the first step in enabling a unified address space in heterogeneous architectures with single-chip CPU-GPU integration [16, 48]. Note that this thesis focuses on coherence in the realm of GPU cores; we leave CPU-GPU cache coherence as future work.

A coherent memory system can be provided trivially by disabling any privates caches. AMD’s Southern Islands GPUs [9] currently have no sup-
1.1. Motivation

Port for disabling private caches. NVIDIA’s Fermi GPUs [68] allow the programmer to manually disable private L1 caches while their Kepler GPUs [70] automatically disable L1 caching of all read-write shared data. However, this disabling of caches adds significant cost by reducing application performance. Figure 1.1(a) shows the potential improvement in performance by enabling L1 caches for a set of GPU applications (described in Chapter 5) that contain communication across GPU cores. We classify these applications as containing \textit{inter-workgroup communication} because groups of threads, called workgroups, that execute on different GPU cores require communication across these cores. These applications require a coherent memory space to execute correctly. In Figure 1.1(a), compared to disabling L1 caches (NO-L1), an ideally coherent GPU (IDEAL-COH) improves performance of these applications by 88% on average. IDEAL-COH maintains a coherent memory in hardware without incurring any latency or traffic costs of real hardware coherence. Thus, simply disabling private caches is insufficient and a hardware cache coherence mechanism for GPUs is warranted.

GPUs present three main challenges for hardware coherence. Figure 1.1(b) depicts the first of these challenges by comparing the interconnect traffic of the baseline non-coherent GPU system (NO-COH) to three GPU systems with cache coherence protocols: writeback MESI, inclusive write-through GPU-VI and non-inclusive write-through GPU-VIni (all three are described in Section 2.2.4). These three are representative of the commonly implemented coherence protocols on CMPs. The applications here are classified as containing \textit{intra-workgroup communication} because their threads limit communication to within workgroups only. Hence, no communication occurs across cores and coherence is not needed. Figure 1.1(b) shows that introducing a coherence protocol to our baseline GPU adds additional traffic overheads. This overhead occurs because coherence traffic is generated even though these applications have no need for it.

The second challenge of GPU coherence is tracking in-flight coherence requests. To ease protocol implementation, CMP coherence designs use worst case sized buffers [33] to track in-flight requests. On a GPU, CMP-like worst case sizing requires an impractical amount of storage for tracking thousands of in-flight coherence requests. This is because CMPs have, at maximum, tens of requests in-flight at a time, while on GPUs this is on the order of tens of thousands.

Third, existing coherence protocols increase hardware complexity by introducing transient coherence states and additional message classes. They require additional virtual networks [85] on GPU interconnects to ensure forward progress. This also increases power consumption in the interconnection
network.

The three aforementioned challenges are unique to high-throughput architectures like GPUs, and to our knowledge are not addressed by any CMP coherence protocols. They arise because GPU architectures are designed to handle a large number of memory accesses in parallel. Recent research on scalable coherence [50, 94] for CMPs with thousands of cores focuses on issues such as tracking a large number of sharers. This is not a problem for current GPUs as they contain only tens of highly parallel cores. In this thesis, we propose a new hardware cache coherence protocol for GPUs that addresses the three challenges described above.

1.2 Contributions

This thesis makes the following contributions:

1. It motivates cache coherence for GPU architectures by quantifying the performance benefits of having coherent caches in GPUs.

2. It presents the challenges of introducing existing hardware cache coherence protocols to GPUs.

3. It presents the GPU-VI and GPU-VIni coherence protocols. These include two optimizations to a VI protocol [52] that make it more suitable for GPU architectures.

4. It provides detailed breakdown of interconnect traffic to show why writeback protocols common on CMPs are not the best design choices for GPUs.

5. It provides detailed complexity and performance evaluations of inclusive and non-inclusive directory protocols on a GPU. It shows that unlike for CMPs, performance does not scale with directory sizes on GPUs.

6. It describes Temporal Coherence, a GPU coherence framework for exploiting synchronous counters in single-chip systems to eliminate coherence traffic and protocol races. Temporal Coherence relies on prediction of lifetimes of private cache lines.
7. It proposes and evaluates the TC-Weak coherence protocol which employs novel timestamp based memory fences to implement Release Consistency (RC) [35] on a GPU.

8. It proposes and evaluates a simple cache line lifetime prediction mechanism that performs well across a range of GPU applications.

1.3 Thesis Organization

The rest of this thesis is organized as follows:

- Chapter 2 provides an overview of the fundamental concepts in memory consistency and cache coherence, and describes the baseline GPU microarchitecture used in this study.

- Chapter 3 describes the challenges of introducing traditional cache coherence to GPUs.

- Chapter 4 describes Temporal Coherence and details the implementations of our two proposed coherence protocols: TC-Strong and TC-Weak.

- Chapter 5 describes the simulation methodology and the applications used in this study.

- Chapter 6 presents and analyzes our experimental results.

- Chapter 7 shows that our two proposed protocols correctly implement the appropriate coherence and consistency invariants.

- Chapter 8 discusses related work.

- Chapter 9 concludes this thesis and presents ideas for future work.
Chapter 2

Background

This chapter describes the necessary background for this thesis. It describes memory consistency models and explains their importance. Then, it presents the basics of hardware cache coherence. Finally, it describes the memory system and cache hierarchy of the baseline non-coherent GPU architecture, similar to NVIDIA’s Fermi GPUs [68], that we study and evaluate in this thesis.

2.1 Memory Consistency

A memory consistency model formally specifies the behaviour of the memory system in a shared-memory multiprocessor. It describes the guarantees on orderings of memory requests and places restrictions on values that can be returned by the memory system. The following subsections describe the basics of memory consistency models.

2.1.1 Memory Consistency Models

The memory consistency problems deals with reordering of memory requests by the memory system. These reorderings can lead to incorrect program execution and result. Table 2.1 lists an example program from Sorin et al. [85] where a reordering of memory requests leads to incorrect results. The program represents a common programming idiom used to implement non-blocking queues in pipeline parallel applications [36]. It describes a producer-consumer relationship with Core 1 producing data and Core 2 consuming it. This communication is facilitated by the shared memory variable data. Another shared memory variable, flag, indicates when new data has been produced by Core 1 and is safe to be read by Core 2. Core 2 waits until flag has been set before reading data into a local register.

Table 2.2 lists two possible executions of the example program in Table 2.1. In Execution 1, Core 1’s two store requests, S1 and S2, complete in the order specified by the program. This execution results in the correct value of data read by Core 2. In Execution 2 of the same program, S1 and S2
2.1. Memory Consistency

Table 2.1: Example program with inter-core communication from Sorin et al. [85].

<table>
<thead>
<tr>
<th>Core 1</th>
<th>Core 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1: data = NEW</td>
<td>L1: r1 = flag</td>
</tr>
<tr>
<td>S2: flag = SET</td>
<td>B1: if(r1 ≠ SET) goto L1</td>
</tr>
<tr>
<td></td>
<td>L2: r2 = data</td>
</tr>
</tbody>
</table>

Table 2.2: Possible executions of example program.

<table>
<thead>
<tr>
<th>cycle</th>
<th>Core 1</th>
<th>Core 2</th>
<th>Core 1</th>
<th>Core 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>S1: data=NEW</td>
<td>S2: flag=SET</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>S2: flag=SET</td>
<td></td>
<td>L1: r1=flag</td>
<td>L1: r1=flag</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>L1: r1=flag</td>
<td>L2: r2=data</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>L2: r2=data</td>
<td></td>
<td>S1: data=NEW</td>
<td></td>
</tr>
</tbody>
</table>

result @ Core 2:  
  r1=SET  
  r2=NEW  
result @ Core 2:  
  r1=SET  
  r2=???

do not complete in program order. Further, S1, the request that updates data, is delayed by the memory system and finishes after request L2 which reads data. Hence, Core 2 reads an incorrect value of data and an incorrect program execution results.

The incorrect outcome in Execution 2 resulted because the program required the update to flag to happen after the update to data, but the memory system reordered these updates. This is an example of a Store→Store reordering. Similar reorderings can occur for two load instructions (Load→Load reordering), a load followed by a store (Load→Store reordering) and a store followed by a load (Store→Load ordering). All of these can lead to incorrect behaviour.

There are many reasons why a core may reorder memory accesses, such as to support features that improve performance or to ease hardware implementation. Modern architectures implement out-of-order cores that may execute instructions out of program order to extract parallelism within a single thread. Reordering of accesses to different addresses that is safe for single-threaded programs may not be safe for multi-threaded programs. Table 2.2 shows an example of a reordering that is unsafe for multi-threaded programs. Reorderings can also occur in stages of the memory system such as the interconnection network, cache controllers and DRAM controllers.
2.1. Memory Consistency

Caches may cause reorderings if the first access misses while the second hits. A Store→Store reordering can occur if the hardware combines the second store with another store earlier than the first store. Store→Load reorderings can arise when a store buffer is implemented to allow latency-critical loads to bypass stores.

A memory consistency model addresses these issues by clearly specifying which reorderings are permissible and which orderings will be enforced. Hardware that implements that memory consistency model must guarantee that the specified orderings are correctly enforced. For memory consistency models that allow reorderings, the hardware must also provide instructions to prevent reordering when a program’s correctness depends on it. These instructions are known as memory fences. For example, inserting a memory fence between instructions S1 and S2 in the example program in Table 2.1 would prevent reordering of their accesses. This modified program would execute correctly on a larger set of processors than the unmodified program.

2.1.2 Types of Consistency Models

Consistency models are typically classified into two groups: strong consistency models and relaxed (or weak) consistency models. Strong models allow fewer reorderings of memory operations in hardware than relaxed models. Relaxed models mandate fewer ordering constraints which may permit more hardware and compiler optimizations leading to higher performance. However, relaxed models may make parallel programming more challenging as the programmer needs to identify instructions in the code where the memory order must be maintained. For example, the program listed in Table 2.1 would execute correctly on a device that supported a strong consistency model, but requires a memory fence instruction for correct execution on relaxed models.

The most well-known and intuitive strong memory consistency model is Sequential Consistency (SC) \[54\]. SC requires that all memory operations from a thread complete in program order, i.e., no reorderings of memory accesses are permitted. Another strong consistency model is Total Store Order (TSO). TSO has been shown to match the memory model of the widely implemented x86 architecture \[81\]. TSO only permits Store→Load reorderings. This relaxation allows the inclusion of store buffers, which hide the latency of store operations by temporarily buffering them at the core and prioritizing load operations. In doing so, load operations occurring later in program order than stores complete before them. TSO fences may be used to enforce a Store→Load ordering, though most programs do not require
Relaxed memory models include commercial models such as IBM Power memory model [29], Alpha [84], SPARC Relaxed Memory Order (RMO) [86], and ARM [80], and academic models such as Release Consistency (RC) [35] and Weak Ordering (WO) [7]. All of these models permit reordering of load and store accesses, i.e., all of Load→Load, Load→Store, Store→Load and Store→Store reorderings are allowed. Each model provides memory fences or similar instructions to enforce an ordering when needed.

2.1.3 Which Consistency Model is the Best?

The choice for best consistency model is debatable. Proponents of strong consistency models argue for their intuitiveness, familiarity and wide-spread use in the form of x86 architectures. Arguments for relaxed consistency include better performance afforded by the relaxation of restrictions on compiler and hardware. With no clear answer to which consistency model is best, assuring portability of programs across models becomes a challenge.

To address this challenge, the concept of Sequential Consistency for Data Race Free (SC for DRF) programs was introduced [7]. A data-race occurs when two threads within a program access the same memory location, at least one of the accesses is a store, and there is no synchronization operation, such as a lock, between the two racing accesses. A program without data-races is called a data-race free program. SC for DRF guarantees that a data-race free program will achieve the same execution on a relaxed consistency model that it would on SC. All consistency models mentioned earlier support SC for DRF. The programmer now only needs to ensure that the program is data-race free to allow portability across platforms with different memory consistency models.

High-level languages like Java and C++ make this task easier for the programmer by providing primitives and libraries for data-race free synchronization. Both the C++ memory model [15] and the Java memory model [58] adopt SC for DRF. Hence, a well-defined consistency model that supports SC for DRF is of paramount importance in supporting C++ and Java on GPUs. To this end, in this work we propose to support Release Consistency, which supports SC for DRF, on GPUs.

2.2 Cache Coherence

Cache coherence is a component of the memory consistency model that deals with problems that arise due to private replication of data in caches.
2.2. Cache Coherence

With the addition of caches, a thread can continue to read the stale, cached copy after it has been updated in main memory. A coherence mechanism is tasked with invalidating those stale copies. It allows the consistency model to provide guarantees that updates will become visible to all threads on an architecture with caches. Coherence can also be thought of as a mechanism to make caches architecturally invisible in the memory system [85].

2.2.1 The Incoherence Problem

To illustrate the problem with naively introducing caches in a multi-core system, Figure 2.1 shows a possible execution of the program from Table 2.1. The figure shows two cores, each with a private data cache, and the main memory. The main memory initially contains the values OLD and NULL.
null for data and flag, respectively. At cycle 1, Core 2 issues the load instruction L1 to read flag. The value NULL is read and stored in Core 2’s private cache. At cycle 2, Core 1 issues the two store instructions which update data and flag in main memory. At cycle 3, Core 2 again issues the load instruction for flag because it did not equal SET the first time. The load access hits in Core 2’s private cache and returns the stale value NULL. A livelock is introduced by adding caches as Core 2 will continuously spin on the load instruction as it waits for the update to flag.

2.2.2 Properties of Hardware Cache Coherence

The coherence mechanism can be implemented in software or hardware. Software coherence conservatively invalidates stale cache entries by inserting invalidation instructions into the code. Hardware coherence does not require any code modifications, provides superior performance to software-based coherence [60], and is more prevalent. A hardware cache coherence protocol performs the following three duties [6]:

1. It propagates newly written values to all privately cached copies.

2. It informs the writing thread or processor when a store operation has been completed and is visible to all threads and processors.

3. It may ensure write atomicity [6], i.e., a value from a store is logically seen by all threads at once.

The first property of hardware coherence is that it propagates new values to processors. Most hardware coherence protocols do so by invalidating all stale copies before the new value is written in a cache or main memory. Coherence assigns coherence states to all cache lines. In directory-based coherence protocols, a structure called a directory tracks the states of all lines currently cached by all processors. For each cache line, the directory also stores a sharer list. The sharer list indicates which processors contain a copy of that line.

Figure 2.2 shows how a directory-based protocol would provide coherence for the example in Figure 2.1. For simplicity, Figure 2.2 shows the directory states embedded with main memory. Most implementations today embed the directory with a shared last-level cache [83] or include a separate on-chip directory structure [28]. We show a conventional MSI coherence protocol, named for the three coherence states Modified, Shared and Invalid. The Modified state indicates the latest value and is an exclusive state; only one processor may have the line in the Modified state, all other private copies
2.2. Cache Coherence

Figure 2.2: Example with idealized directory coherence. Compared to Figure 2.1, coherence adds states to each cache line and a directory in main memory. The directory tracks the coherence state and a sharer list for each line.
2.2. Cache Coherence

Table 2.3: Independent Read Independent Write example from Sorin et al. [85]. Initially, data1 and data2 = OLD.

<table>
<thead>
<tr>
<th>Core 1</th>
<th>Core 2</th>
<th>Core 3</th>
<th>Core 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>data1 = NEW</td>
<td>data2 = NEW</td>
<td>r31 = data1</td>
<td>r41 = data2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>FENCE</td>
<td>FENCE</td>
</tr>
<tr>
<td>r32 = data2</td>
<td></td>
<td></td>
<td>r42 = data1</td>
</tr>
</tbody>
</table>

must be invalid. The Shared state indicates that multiple processors may have a copy of the line. The Invalid state indicates an invalid copy.

In Figure 2.2, both data and flag start in the Invalid state as neither of the cores has a copy. At cycle 1, when Core 2 reads flag, its state changes to Shared. At cycle 2, Core 1 attempts to write new values to data and flag. To do this, Core 1 requests Modified permissions for these two lines from the directory. The directory finds flag in the Shared state in Core 2. Since Modified is an exclusive state, the directory sends an invalidation request to Core 2 which changes Core 2’s copy of flag to Invalid. After acquiring Modified permissions, Core 1 can update the values of data and flag in its local cache. At cycle 3, Core 2 again attempts to read flag. Its local copy is invalid, so a request is sent to the directory. The directory finds Core 1 to contain a Modified copy and forwards the request to Core 1. Core 1 changes its copy’s state to Shared and responds to Core 2 with the latest value of flag. By invalidating the stale copy in Core 2 and forwarding the new copy from Core 1, the coherence protocol is able to eliminate the livelock issue due to incoherence in Figure 2.1.

The second property of coherence is that it informs the writing processor when the value written by the store operation has become visible to all processors. As discussed earlier in Section 2.1, the code in Table 2.1 requires a memory fence instruction between the two stores on a relaxed memory consistency model. The fence instruction needs to guarantee that other processors cannot read the old value of flag beyond this point. In Figure 2.2, the fence instruction would wait until the store instruction S1 was complete, ensuring that all stale copies of flag were invalidated by the coherence protocol. This work presents a coherence protocol called TC-Weak where a store completes before all stale copies have been invalidated. Instead, we propose a novel memory fence mechanism to ensure that this second coherence property is met.
Finally, the third coherence property, called write atomicity, guarantees that the value from a store operation becomes logically visible to all threads or processors at the same time. The write atomicity property is optional in coherence; some memory consistency models \cite{7,35,84,86} require it while others \cite{2,35,80} do not. We propose a coherence protocol called TC-Weak that does not provide write atomicity. Table \ref{table:coherence} lists an example program, called Independent Read Independent Write (IRIW) \cite{85}, where write atomicity and lack thereof can return different results. In the IRIW example, Cores 1 and 2 each write to separate locations, and Cores 3 and 4 read both locations but in reversed order. The result $r_{31}=\text{NEW}, r_{41}=\text{NEW}$, $r_{32}=\text{OLD}, r_{42}=\text{OLD}$ is not possible with write atomicity. It implies that Core 3 saw the update to $\text{data1}$ before the update to $\text{data2}$, and Core 4 saw the update to $\text{data2}$ before $\text{data1}$. With no write atomicity, this execution is valid and that result may occur. The IRIW code example makes use of data races and does not conform to SC for DRF. Modifying the code to eliminate the data-races by adding the appropriate synchronization operations would allow it to conform to SC for DRF and would ensure it executes correctly across all memory models. Most programs do not depend on write atomicity for correctness \cite{6}, and write atomicity is not required by memory models for high level languages like Java and C++ \cite{6,85}.

Sorin et al. \cite{85} summarize the three coherence protocol properties into two invariants. The first is referred to as Single-Writer Multiple-Reader (SWMR) and requires that at any given time a memory location may be written by only a single thread, or it may be read by any number of threads. This allows the division of a memory location’s lifetime into epochs. In a read-write epoch, only a single thread is allowed to read and write to the given memory location. In a read-only epoch, any number of threads or processors are allowed to only read the given memory location. The second invariant is called the Data-Value Invariant and specifies that modifications to a value in one epoch must be visible in the next epoch. In Section 7.1 we show that one of our proposed protocols that supports write-atomicity, TC-Strong, ensures these two invariants. Section 7.2 also shows that our other proposed protocol that does not implement write-atomicity, TC-Weak, correctly implements the invariants needed for Release Consistency.

### 2.2.3 Coherence Implementation

Below we describe common design decisions for hardware coherence.

**States.** Coherence assigns additional states to lines in caches. The example in Figure \ref{fig:coherence} used the M(odified), S(hared) and I(nvalid) coherence
2.2. Cache Coherence

states. These are termed *stable* states as they describe lines not undergoing coherence activity. Additional *transient* states are needed to track transitions from one stable state to another. Transient states are tracked in associative lookup tables called Miss Status Holding Registers (MSHRs) (sometimes also called Transaction Buffer Entries (TBEs)), rather than in the cache or directory storage itself. Reasons for this are twofold [85]. First, transient states need to track additional information regarding the transition. For example, when a core attempts to store a new value to a line in the S state, it must first acquire M permission for the line from the directory. This line will move to the S→M transient state until a response is received. If the directory informs the core that other sharers exist, then the core must wait for invalidation acknowledgments from other sharers before transitioning to the M state. The number of pending acknowledgment messages is additional information that the transient state needs to track. This is tracked inside each MSHR entry. Second, a cache controller may buffer incoming messages into a pool and each cycle select a message that is ready to service. An associative lookup of transient states is needed to quickly determine which of the messages can be processed.

**Inclusion and Non-Inclusion.** An inclusive cache hierarchy is one where if a line is present in the lower level cache it must also exist in the higher level cache. For an inclusive two-level GPU cache hierarchy, a copy in the private L1 cache would mean that the shared last-level L2 cache also contains a copy. Inclusion allows the coherence directory to be embedded within the last-level cache by appending a directory state field and a sharer list to each cache line. It simplifies directory implementation [11] and saves tag storage that would be needed if the directory was a separate structure. The disadvantage is that to maintain inclusion, it invalidates all L1 copies by sending *recall* messages to all L1 sharers when an L2 line needs to be evicted. On CMPs, improperly sized caches can lead to excessive recalls and performance degradation [60]. In this work we study the effects of both inclusion and non-inclusion on GPU coherence.

**Write-Through and Writeback.** The MSI protocol in Figure 2.2 implemented writeback L1 caches. The store operation completed at the L1 cache after Modified permissions were acquired, and the data was not propagated to the main memory. Writeback caching reduces store latency and saves the energy of writing data through to the higher level cache at every store. Most modern CMP architectures [20, 28, 55, 77, 83] implement writeback caches and hence coherence protocols such as MSI, MESI and MOESI are common. The alternative to writeback caches are write-through caches. These propagate store data to the next level cache at every
2.2. Cache Coherence

Figure 2.3: Deadlock example. (a) Same network buffer for both requests and responses introduces a deadlock. (b) Separate network buffers eliminate deadlock.

store. As a result, write-through caches do not hold dirty data, eliminate the need for Modified and similar coherence states, and simplify coherence implementation. Write-through coherence requires only 2 stable states at the L1: Valid and Invalid. We refer to a write-through coherence protocol as the VI protocol.

Virtual Networks. Coherence protocols put additional constraints on interconnection networks. They require a minimum number of virtual networks depending on the protocol implementation. Figure 2.3 illustrates the need for this. Core 1 has a cache line in the S→M transient state that is waiting for an invalidation acknowledgement from Core 2 before going to the M state. Core 1 also has requests that currently cannot be processed. An example would be a request from Core 3 for the same line. In Figure 2.3(a), a single virtual network is used, Core 2’s response becomes stuck behind the blocked requests, and a system deadlock results. In Figure 2.3(b), the response is sent on a separate virtual network, it bypasses the blocked requests, and a deadlock is avoided. Coherence requires a separate virtual network for every message class [85]. Virtual networks may be implemented as separate physical networks, or as separate virtual channels [30] within the same physical network. We adopt the latter approach for our evaluations.
2.2. Cache Coherence

2.2.4 Directory Protocols

This section describes the MESI and GPU-VI directory protocols that we compare against in this thesis. All of MESI, GPU-VI and GPU-VIni require a coherence directory to track the L1 sharers. MESI and GPU-VI enforce inclusion through invalidation (recall) of all L1 copies of a cache line upon L2 evictions. Inclusion allows the sharer list to be stored with the L2 tags. GPU-VIni is non-inclusive and requires separate on-chip storage for a directory.

MESI

The MESI coherence protocol is taken from the gem5 simulator [14]. It is representative of common writeback CMP protocols. Both of MESI’s L1 and L2 caches are writeback write-allocate. It is similar to the MSI protocol described in Section 2.2 with the exception of the additional Exclusive or E state. The E state is like the M state in that only one processor may have a valid copy of the line in this state. Unlike M, the E state is read-only and is given out in place of the S state when a processor is the first and only agent to read a line. If that same processor later stores to that line, it can silently upgrade the line’s state from E to M without having to send the directory a message to acquire Modify permissions. This E state essentially optimizes for read-write data by optimistically assuming that any data read by a processor will soon be modified by the same processor.

Since MESI is a writeback protocol, it does not use the write-through mechanism for performing atomic operations at the shared L2 cache described in Section 2.3.4 Instead, MESI performs the atomic operation at the L1 cache after the write permission to the line has been acquired. All the other protocols we evaluate in this thesis use the write-through mechanism from the baseline GPU to perform atomic operations.

MESI also contains optimizations to eliminate the point-to-point ordering requirement of the non-coherent GPU interconnect and cache controllers. Instead, MESI relies on five physical or virtual networks to support five different message classes to prevent protocol deadlocks. MESI implements complex cache controllers capable of selecting serviceable requests from a pool of pending requests. The write-allocate policy at L1 requires that store data be buffered until proper coherence permission has been obtained. This requires the addition of area and complexity to buffer stores at each GPU core.
GPU-VI

GPU-VI is a two-state coherence protocol inspired by the write-through protocol in Niagara [52]. GPU-VI implements write-through, write-no allocate L1 caches. It requires that any store completing at the L2 invalidate all L1 copies. A store to a shared cache line cannot complete at the L2 until the directory has sent invalidation requests and received acknowledgments from all sharers. Since GPU-VI implements an inclusive cache hierarchy, the directory information is embedded in the L2 cache lines. It requires recall messages when an L2 line needs to be evicted. GPU-VI has 4 message classes and thus requires 4 physical or virtual networks to guarantee deadlock-free execution.

We now describe the detailed implementation of the GPU-VI protocol. A cache coherence protocol is formally specified as a set of state machines. There is one state machine for each type of cache controller. In the case of a two-level cache hierarchy present in GPUs, GPU coherence specifies one state machine for the L1 caches and one for the last-level L2 cache. Coherence state machines are commonly presented in a table format [59, 85]. Tables 2.4 and 2.5 present the L1 and L2 state machines, respectively, of the GPU-VI coherence protocol.

Tables 2.4 and 2.5 list the states in the leftmost column and events in the topmost row. Each table cell describes the transitions between states when an event, given by the column heading, occurs for a cache line in the state, given by the row heading. The cell lists the actions taken by the controllers and the next state of the cache line. Note that each cache line has its own independent state; coherence is not concerned with interactions between cache lines of different addresses. The underneath captions describe the messages, events and conditional statements used within the tables.

Table 2.4 lists the 5 L1 states. The stable states, which are also the protocol’s namesake, are \(V(alid)\) and \(I(nvalid)\). The remaining three transients states are \(V_M\), \(L_V\) and \(L_I\). \(V_M\) handles store operations to a valid line, \(L_V\) handles load misses, and \(L_I\) handles stores misses and invalidation races. A load transaction looks as follows. A load requested to an \(I\) line issues a (GETS) request to the L2 and moves to the \(L_V\) state. When data is returned from the L2, it moves to the \(V\) state. Similarly, a store request to an \(I\) line issues a request to the L2 and moves to the \(L_I\) state. However, a store acknowledgment from the L2 moves the line back to \(I\). This is because the L1 in GPU-VI is write-no allocate, i.e., a store miss does not allocate a line and bring in data. Atomic requests are handled similarly to store. Since atomic requests are performed at the L2 and also return data, for simplicity
### 2.2. Cache Coherence

#### Table 2.4: L1 States for GPU-VI Protocol.

<table>
<thead>
<tr>
<th>State</th>
<th>Processor Request</th>
<th>L1 Action</th>
<th>From L2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Load</td>
<td>Store</td>
<td>Atomic</td>
</tr>
<tr>
<td>I</td>
<td>GETS</td>
<td>GETX</td>
<td>ATOMIC</td>
</tr>
<tr>
<td>(Invalid)</td>
<td>→ I.V</td>
<td>→ I.J</td>
<td>→ I.J</td>
</tr>
<tr>
<td>V</td>
<td>hit</td>
<td>GETX</td>
<td>ATOMIC</td>
</tr>
<tr>
<td>(Valid)</td>
<td>→ V.M</td>
<td>→ I.J</td>
<td>→ I</td>
</tr>
<tr>
<td>V.M</td>
<td>GETS</td>
<td>GETX</td>
<td>ATOMIC</td>
</tr>
<tr>
<td>(Upgrade)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>L.V</td>
<td>×</td>
<td>GETX</td>
<td>ATOMIC</td>
</tr>
<tr>
<td>(Load Miss)</td>
<td>→ I.J</td>
<td>→ I.J</td>
<td>→ I.J</td>
</tr>
<tr>
<td>L</td>
<td>GETS</td>
<td>GETX</td>
<td>ATOMIC</td>
</tr>
<tr>
<td>(Store Miss)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**L1⇒L2 messages:** GETS (load). GETX (store). ATOMIC. INVACK (invalidation ack).

**L2 Triggered Events @ L1:** INV (invalidation or recall request). DATA (data from load or atomic request). ACK (store complete from L2).

**L1 Conditionals:** load/store/atomic? (response to GETS/GETX/ATOMIC?). pending 0? (all pending requests satisfied?).

19
### 2.2. Cache Coherence

#### Table 2.5: L2 States for GPU-VI Protocol.

<table>
<thead>
<tr>
<th>State</th>
<th>L1 Request or Response</th>
<th>L2 Action</th>
<th>From Mem Mem Data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>GETS</td>
<td>GETX</td>
<td>ATOMIC</td>
</tr>
<tr>
<td>I (Invalid)</td>
<td>FETCH</td>
<td>→ I$_S$</td>
<td>FETCH</td>
</tr>
<tr>
<td>N (No Sharers)</td>
<td>add sharer</td>
<td>ACK</td>
<td>DATA</td>
</tr>
<tr>
<td>S (≥1 Sharers)</td>
<td>add sharer</td>
<td>(private?)</td>
<td>ACK</td>
</tr>
<tr>
<td>L$_S$ (Load Miss)</td>
<td>merge</td>
<td>stall</td>
<td>stall</td>
</tr>
<tr>
<td>L$_M$ (Store Miss)</td>
<td>stall</td>
<td>stall</td>
<td>stall</td>
</tr>
<tr>
<td>S$_M$ (Invalidating)</td>
<td>stall</td>
<td>stall</td>
<td>stall</td>
</tr>
<tr>
<td>M$_J$ (Recalling)</td>
<td>stall</td>
<td>stall</td>
<td>stall</td>
</tr>
</tbody>
</table>

**L$_2$⇒L$_1$ messages:** ACK (store done). DATA (data response). INV (invalidation requests to all sharers except the requester). RCL (recall/invalidation requests to all sharers).

**L$_2$⇒MEM messages:** FETCH (fetch data from memory). WB (writeback data to memory).

**L$_2$ Conditionals:** dirty? (L2 data modified?). multiple? (multiple load requests merged?). private? (lone sharer is requester?).

**L$_2$ Actions:** merge (merge load requests). add sharer (add requester to sharers list). reset sharer (remove all sharers and add requester to sharers list).
reasons they immediately invalidate an L1 line in the V state. Also of note is that receiving invalidations in any of the transient states causes a line to move to the I state. I is a catch-all state to handle invalidation races and invalidate the line after its original transaction is complete.

Table 2.5 lists the 7 states for GPU-VI's L2 state machine. The I stable state is for an invalid L2 line, the N stable state for a valid L2 line with no L1 sharers, and the S stable state for a valid L2 line with one or more L1 sharers. The LS and LM transient states handle load and store misses, respectively.

The L2 is a writeback, write-allocate cache, so unlike the L1 both load and store misses allocate a line in the L2. The SM transient state handles sending invalidations to sharers and waiting for acknowledgments when a processor tries to store to a shared line. The MI transient state handles sending recall/invalidation messages to all sharers when the L2 wants to evict a shared line. Of note are the transactions involving invalidations. When an S line with sharers receives a store (GETX) request it may perform one of two transitions. It will either simply return an acknowledgment if the writer is the lone sharer, or it will send invalidation requests to all sharers and then wait in the SM state. In the SM state, when all invalidation acknowledgments have been received, the L2 sets the requester as the lone sharer, returns an acknowledgment to the L1 requester, and moves the L2 line back to the S state. A similar sequence of events occurs if an S line is evicted. After sending any dirty writeback data to memory, the L2 sends recall/invalidation requests to all sharers and moves the line to the MI state. The line waits in the MI state until all invalidation acknowledgments have been received, and then moves to the invalid state.

GPU-VI adds two optimizations over a conventional VI protocol [87] on CMPs. First, the CMP implementation buffers the store data in a store queue and only writes the data to the L1 cache when a store acknowledgment has been received from the directory. The storage needed to buffer store data presents a significant challenge for GPUs because the number of concurrent memory requests is orders of magnitude greater than CMPs. Instead, in case of a store hit, GPU-VI writes the store data to the L1 cache as soon as the request is serviced by the L1. In case of a store miss, the write no-allocate feature does not allocate an L1 line. In both cases, GPU-VI does not need to buffer store data at the core after it exits the in-order memory stage. This optimization alone breaks write atomicity. A thread can now read from the L1 a value written by another thread on the same GPU core before all invalidation acknowledgments for that line are received from the directory. A naive solution here would be to disallow a read operation to an L1 line that has a pending store. The native solution is problematic because the memory
2.3. Graphics Processing Units

stage in our baseline GPU cores is in-order. Stalling a read operation will stall the entire upstream memory pipeline, including memory requests from other wavefronts. GPU-VI implements a second optimization. It treats loads to any L1 line that has pending stores as misses. This prevents the load from stalling the upstream memory stages. The point-to-point ordering guarantee provided by the GPU memory system will ensure that the load cannot read the value from a pending store at the L2 prematurely, i.e., before the store operation invalidates all stale copies. Hence, write atomicity will also be maintained.

GPU-VIni

The non-inclusive GPU-VIni (the ni stands for non-inclusive) decouples the directory storage in GPU-VI from the L2 cache to allow independent scaling of the directory size. It adds additional complexity to manage the states introduced by a separate directory structure. The same cache controller in GPU-VIni manages the directory and the L2 cache. Eviction from the L2 cache does not generate recall requests. Eviction from the directory still requires recall because the directory must track all valid lines in the L1 caches. Eviction of a directory entry also triggers an L2 eviction because the directory must also track all valid L2 lines. The directory in GPU-VIni can be sized independently of the L2 cache; its size can be increased to reduce recall rates. By default, GPU-VIni implements an 8-way associative directory with twice the number of entries as the number of total private cache lines (R=2 as in the framework proposed by Martin et al. [60]). Section 6.7 presents data for GPU-VIni with larger directory sizes.

2.3 Graphics Processing Units

The GPU is a type of a compute accelerator architected to provide high throughput on parallel programs. The highly parallel GPU architecture leverages light-weight threads and SIMD hardware to support thousands of concurrent operations. In contrast, CMP architectures are designed to execute a single to a few threads on each core. CMPs expend hardware resources to improve the performance of individual threads through features such as speculation, out-of-order execution and large on-chip caches. The GPU forgoes such features in favour of a large number of ALUs and deeper memory pipelines. The GPU effectively hides long latencies, such as that of a cache-missed memory access, by having a large pool of threads ready to issue an instruction. This allows programs with sufficient independent
2.3. Graphics Processing Units

Figure 2.4: Baseline non-coherent GPU Architecture.

parallelism among threads to achieve high utilization of the parallel GPU hardware and achieve significant speedups over CMP architectures.

To support fast switching between thread contexts the GPU architecture contains key differences compared to CMPs. A large register file on the order of Megabytes holds register state for all concurrently scheduled threads and reduces the overhead of switching thread contexts. A scheduler selects between thousands of concurrently executing threads to issue the next instruction. Most importantly to the subject of this thesis, the GPU memory system is designed to handle thousands of concurrently executing memory operations that are generated by the thousands of concurrent threads. The GPU’s significantly higher memory parallelism is needed to effectively hide latencies for the large number of GPU threads.

The following subsections describe the baseline GPU architecture that we study in this work. We describe a non-cache coherent GPU architecture that resembles NVIDIA’s Fermi GPUs [68].

2.3.1 High-level GPU Architecture

Figure 2.4 shows the high-level organization of our baseline non-coherent GPU architecture. Only features of the GPU architecture that are relevant to this work are presented in the figure. A set of GPU cores (Compute cores in AMD terminology and Single-Instruction Multiple-Thread (SIMT) cores in NVIDIA’s terminology) connect via an on-chip interconnection network.
2.3. Graphics Processing Units

to a set of memory partitions. Each core’s memory unit contains an on-chip scratchpad memory (Local Memory in AMD systems and Shared Memory in NVIDIA systems), a memory coalescing unit, an L1 data cache and a set of L1 MSHRs. The memory partitions connect to off-chip GDDR DRAM chips. The memory partitions also house a slice of the shared L2 cache and an Atomic Unit that is used to execute read-modify-write operations.

GPUs are most commonly programmed using OpenCL \[51\] or CUDA \[71\]. An OpenCL or CUDA application begins execution on a CPU and launches compute kernels onto a GPU. Each kernel launches onto a GPU a hierarchy of threads: an NDRange of work groups of wavefronts of work items/scalar threads in OpenCL terminology. In NVIDIA’s terminology, the same hierarchy is named a grid of Cooperative Thread Arrays (CTAs) of warps of threads. Each workgroup or CTA is assigned to a single GPU core. Scalar threads within the workgroup are managed as SIMD execution groups consisting of 32 threads called wavefronts or warps. At each cycle the hardware scheduler in each GPU core selects a wavefront with an instruction ready to execute and schedules it onto one of the SIMD pipelines.

2.3.2 GPU Memory System

A GPU kernel commonly accesses the local, thread-private and global memory spaces. Software-managed scratchpad memory is stored in on-chip scratchpad buffers at each GPU core and used for communication between threads in the same workgroup. Thread-private memory is private to each thread. Global memory is shared across all threads on a GPU. Both thread-private and global memory are stored in off-chip GDDR DRAM and cached in the multi-level cache hierarchy. Of these two only global memory is accessible across threads and hence may require cache coherence. The off-chip DRAM memory is divided among a number of memory partitions that connect to the GPU cores through an interconnection network.

Memory accesses to the same cache line from different threads within a wavefront are merged into a single wide access by the Coalescing Unit. Therefore, a memory instruction from a single wavefront generates one memory access for every unique cache line accessed by the threads in the wavefront. This can range from one access when each of the 32 threads in a wavefront accesses a consecutive 4-byte word within a single 128-byte cache line, to 32 accesses when each thread accesses a different cache line. This behaviour is in stark contrast to a CPU where a single memory instruction only accesses small part of a single cache line. All memory requests on our baseline GPU are handled in First-In First-Out (FIFO) order by the in-order
memory stage of a GPU core. Stores to the same word by multiple scalar threads in a single wavefront do not have a defined behaviour \cite{71}; only one store will succeed. In this thesis, from the memory consistency model's perspective, a GPU wavefront is similar to a CPU thread.

2.3.3 GPU Cache Hierarchy

The GPU cache hierarchy consists of per-core private L1 data caches and a shared L2 cache. The L1s follow a write-evict \cite{71} (write-purge \cite{46}), write no-allocate caching policy. The L1 caches are not coherent. This means that updates to values will not become visible to other GPU cores that have the stale value cached in their L1 data caches. Those other cores will continue to read the stale value until it is evicted from their L1. Each memory partition houses a single bank of the L2 cache. The L2 cache banks are writeback with write-allocate. Memory accesses generated by the coalescing unit in each GPU core are passed, one per cycle, to the per-core L1 MSHR table. The L1 MSHR table combines read accesses to the same cache line from different wavefronts to ensure only a single read access per-cache line per-GPU core is outstanding. Stores are not combined and, since they write-through, any number of store requests to the same cache line from a GPU core may be outstanding. Point-to-point ordering in the interconnection network, L2 cache controllers and off-chip DRAM channels ensures that multiple outstanding stores from the same wavefront to the same address complete in program order. All cache controllers service one memory request per cycle in order. Misses at the L2 are handled by allocating an L2 MSHR entry and removing the request from the request queue to prevent stalling.

2.3.4 Atomic Operation

Read-modify-write atomic operations are performed at each memory partition by an Atomic Operation Unit. In our model, the Atomic Operation Unit can perform a read-modify-write operation on a line resident in the L2 cache in a single cycle.
Chapter 3

Challenges of GPU Coherence

This section describes the main challenges of introducing conventional coherence protocols to GPUs.

3.1 Coherence Traffic

Traditional coherence protocols introduce unnecessary traffic overheads to existing GPU applications that are designed for non-coherent GPU architectures. With MESI, the traffic overhead is primarily due to its write-allocate L1 caches. Write-allocate optimizes for store locality by allocating lines into the L1 on store misses. The detailed breakdown in Section 6.2 shows that GPU applications exhibit little store locality and the additional traffic is wasted. Implementing the writeback MESI with write-no allocate instead of write-allocate would greatly complicate the protocol as now a store operation can complete at both the L1 and the L2 cache. Additional transient states are needed at the L1 to generate and complete a write-through operation when a store misses. Additional transient states are also needed at the L2 to handle a store completing at the L2.

The write-through GPU-VI protocols are better suited than MESI for GPUs. However, they too reveal a traffic overhead of traditional coherence on GPUs: recall traffic. When a directory needs to evict an entry, it sends recall messages to all L1 sharers asking them to invalidate their copies. Recall traffic is a function of the directory size relative to the aggregate capacity of all L1 caches, the larger the directory’s relative size the fewer the number of recalls. For inclusive protocols, the directory size is fixed by the size of the last-level cache. Inclusive protocols on CMPs implement large last-level caches, which keeps the recall rate low [60]. A GPU cache hierarchy, however, is driven by graphics workloads. It features a last-level L2 cache roughly equal in size to the aggregate L1 caches [9, 68, 69]. CMP coherence protocols may also employ explicit notifications at L1 evictions to...
update the directory’s sharer state \cite{28} to reduce recall traffic. Each private cache on a GPU services tens of wavefronts, each of which can generate 32 memory requests per instruction, leading to high cache contention and line eviction rates. We tested a version of GPU-VI with explicit L1 eviction messages and found it to increase, rather than decrease, the traffic overhead. Hence, the inclusive GPU-VI is susceptible to recall traffic on GPUs.

The next option is a non-inclusive write-through protocol like GPU-VIni. We found that a directory sized according to CMP designs \cite{60} still suffers from a high recall rate in GPUs. Finally, we show in Section 6.7 that while very large directories reduce recalls to manageable amounts, they introduce invalidation traffic since they are large enough to capture working sets between two consecutive GPU kernels. For intra-workgroup applications, this invalidation traffic is an overhead that did not exist in the baseline non-coherent GPU.

An effective way to reduce coherence traffic is to selectively disable coherence for data regions that do not require it. Kelm et al. \cite{49} proposed a hybrid coherence protocol to disable hardware coherence for regions of data. It requires additional hardware support and code modifications to allow data to migrate between coherence domains. In this thesis we present a protocol that enables coherence without the use of coherence messages. Section 4.3 explains how TC-Weak uses timestamps to enforce coherence at cache line granularity without requiring any code modifications to identify coherent and non-coherent data.

### 3.2 Storage Requirements

In directory protocols, requests racing to the directory may be handled in one of three ways \cite{85}. They may be processed immediately, held at the directory in on-chip storage, or negatively acknowledged (NACKed) and re-issued at a later time. Negative acknowledgments greatly complicate a protocol, can lead to protocol livelocks, and should be avoided \cite{85}. Instead, CMP coherence implementations dedicate enough on-chip storage resources to buffer the worst case number of coherence requests \cite{33}.

GPUs, however, execute tens of thousands of scalar threads in parallel. These can lead to tens of thousands of memory requests in-flight at a time, leading to large MSHR storage overheads for buffering coherence transactions. Assume 11-byte MSHR directory entries (8 bytes for address, 0.5 bytes for coherence states, 2 bytes for core IDs/sharer vector, 0.5 bytes for pending acknowledgments count). Also assume a maximum of one out-
3.3 Protocol Complexity

Table 3.1: Comparison of protocol state counts.

<table>
<thead>
<tr>
<th>State Type</th>
<th>Non-Coh.</th>
<th>GPU-VI</th>
<th>GPU-VIni</th>
<th>MESI</th>
<th>TC-Weak</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 Cache</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stable</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>Transient Cache</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>Total L1 States</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>10</td>
<td>5</td>
</tr>
<tr>
<td>L2 Cache</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stable</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Transient Cache</td>
<td>0</td>
<td>2</td>
<td>8</td>
<td>9</td>
<td>1</td>
</tr>
<tr>
<td>Total L2 States</td>
<td>4</td>
<td>7</td>
<td>15</td>
<td>16</td>
<td>7</td>
</tr>
</tbody>
</table>

Understanding memory instruction per wavefront, which can diverge to generate 32 memory requests. With 48 wavefronts per GPU core and 16 GPU cores, a total of 24,576 memory requests may be in-flight. Worst case MSHR sizing would therefore require 264kB of storage. This means that 25% of our baseline GPU’s L2 cache is dedicated towards buffering coherence transactions. Note that our baseline memory system is not restricted to one memory instruction per wavefront, we only use this assumption for our numerical argument. Mechanisms that throttle the network via back-pressure flow-control mechanisms [61] reduce the worst-case storage requirements, but further complicate coherence controller implementations.

In this thesis we propose two TC coherence protocols that eliminate the worst-case storage requirement for coherence transactions. They do so by completely eliminating coherence messages, and hence any protocol races. As a result, they also do not require any additional virtual networks over the baseline GPU because they do not introduce any additional message classes.

3.3 Protocol Complexity

Table 3.1 lists the number of states in the protocols we evaluate. We term stable states as states conventionally associated with a coherence protocol, for example, Modified, Exclusive, Shared and Invalid for the MESI protocol. Transient states are intermediate states occurring between stable states. Specifically, transient cache states are states associated with regular cache operations, such as maintaining the state of a cache line while a read miss is serviced. Transient cache states are present in a coherence protocol as well as the non-coherent architecture. Transient coherent states are additional states needed by the coherence protocol. An example is a state indicating that the given line is waiting for invalidation acknowledgments. Coherence
3.3. Protocol Complexity

Protocol verification is a significant challenge that grows with the number of states \[24\], a problem referred to as state space explosion \[72\].

Table \[3.1\] shows that MESI is one of the more complex protocols and introduces the most number of transient coherence states over the baseline non-coherent GPU. An inclusive cache hierarchy simplifies coherence implementation \[11\] and write-through coherence is simpler than writeback coherence \[85\]. GPU-VI implements both features and therefore introduces the least complexity of the three traditional coherence protocols. GPU-VINi is more complex than GPU-VI because it implements a separate non-inclusive directory.

The TC-Weak protocol that we propose in this thesis offers the least complexity measured in the number of transient states. Unlike GPU-VI, TC-Weak does not require the addition of more virtual networks to the baseline GPU. It does so by eliminating coherence messages entirely.
Chapter 4

Temporal Coherence

This section presents Temporal Coherence (TC), a timestamp based cache coherence framework designed to address the needs of high-throughput GPU architectures. Time based hardware coherence was first proposed by Shim et al. [57, 82] for CMPs. They proposed the Library Cache Coherence (LCC) protocol that implements Sequential Consistency on CMPs. Like LCC, TC uses time-based self-invalidation for coherence. Unlike LCC, TC provides a relaxed memory model for GPU applications. In this thesis, we propose two implementations of TC: TC-Strong and TC-Weak. TC-Strong is similar to LCC in that it stalls store operations until they have been self-invalidated by all GPU cores. We find TC-Strong to perform poorly on a GPU. Our second implementation of TC, TC-Weak, uses a novel timestamp-based memory fence that eliminates this stalling, performs well on a GPU, and addresses the challenges of GPU coherence described in Chapter 3.

Section 4.1 describes time-based coherence. Section 4.2 describes TC-Strong and compares it to LCC. Section 4.3 describes TC-Weak, a novel TC protocol that uses time to drive both coherence and consistency operations.

4.1 Time and Coherence

In essence, the task of an invalidation-based coherence protocol is to communicate among a set of nodes the beginnings and ends of a memory location’s epochs [85]. Time-based coherence uses the insight that single chip systems can implement synchronized counters [43, Section 17.12.1] to enable low cost transfer of coherence information. Specifically, if the lifetime of a memory address’ current epoch can be predicted and shared among all readers when the location is read, then these counters allow the readers to self-invalidate synchronously, eliminating the need for end-of-epoch invalidation messages.

Figure 4.1 compares the handling of invalidations between the GPU-VI directory protocol and TC. The figure depicts a read by processors C1 and C2, followed by a store from C1, all to the same memory location. Figure 4.1(a) shows the sequence of events that occur for the write-through GPU-VI directory protocol. C1 issues a load request to the directory (1),
4.1. Time and Coherence

Figure 4.1: Coherence invalidation mechanisms. Messages: R=read, D=data, W=write, Inv=invalidation.
4.2. TC-Strong Coherence

and receives data. C2 issues a load request (2) and receives the data as well. C1 then issues a store request (3). The directory, which stores an exact list of sharers, sees that C2 needs to be invalidated before the store can complete and sends an invalidation request to C2 (4). C2 receives the invalidation request, invalidates the line in its private cache, and sends an acknowledgment back (5). The directory receives the invalidation acknowledgment from C2 (6), completes C1’s store request, and sends C1 an acknowledgment (7).

Figure 4.1(b) shows how TC handles the invalidation for this example. When C1 issues a load request to the L2, it predicts that the read-only epoch for this address will end at time T=15 (1'). The L2 receives C1’s load request and epoch lifetime prediction, records it, and replies with the data and timestamp of T=15 (2'). The timestamp indicates to C1 that it must self-invalidate this address in its private cache by T=15. When C2 issues a load request, it predicts the epoch to end at time T=20 (3'). The L2 receives C2’s request, checks the timestamp stored for this address and extends it to T=20 to accommodate C2’s request, and replies with the data and a timestamp of T=20 (4'). At time T=15 (5'), C1’s private cache self-invalidates the local copy of the address. At time T=20 (6'), C2 self-invalidates its local copy. When C1 issues a store request to the L2 (7'), the L2 finds the global timestamp (T=20) to be less than the current time (T=25) indicating that no L1 caches contain a valid copy of this line. The L2 completes the store instantly and sends an acknowledgment to C1 (8').

Compared to GPU-VI, TC does not use invalidation messages. Globally synchronized counters allow the L2 to make coherence decisions locally and without indirection. This example shows how a TC framework can achieve our desired goals for GPU coherence; all coherence traffic has been eliminated and, since there are no invalidation messages, the transient states recording the state of outstanding invalidation requests are no longer necessary. Lifetime prediction is important in time-based coherence as it affects cache utilization and application performance. Section 4.4 describes our simple predictor for TC-Weak that adjusts the requested lifetime based on application behaviour.

4.2 TC-Strong Coherence

TC-Strong implements RC with write atomicity [35]. It implements write-through L1 caches and a writeback L2. TC-Strong requires synchronized timestamp counters at the GPU cores and L2 controllers shown in Fig-
4.2. TC-Strong Coherence

Figure 4.2: Hardware extensions for TC-Strong and TC-Weak. (a) GPU cores and memory partitions with synchronized counters. A Global Write Completion Time (GWCT) table added to each GPU core (TC-Weak only). (b) L1 and L2 cache lines with timestamp field.

Figure 4.2(a) to provide the components with the current system time. A small timestamp field is added to each cache line in the L1 and L2 caches, as shown in Figure 4.2(b). The local timestamp value in the L1 cache line indicates the time until the particular cache line is valid. An L1 cache line with a local timestamp less than the current system time is invalid. The global timestamp value in the L2 indicates a time by when all L1 caches will have self-invalidated this cache line.

4.2.1 TC-Strong Operation

Every load request checks both the tag and the local timestamp of the L1 line. It treats a valid tag match but an expired local timestamp as a miss; self-invalidating an L1 line does not require explicit events. A load miss at the L1 generates a request to the L2 with a lifetime prediction. The L2 controller updates the global timestamp to the maximum of the current global timestamp and the requested local timestamp to accommodate the amount of time requested. The L2 responds to the L1 with the data and the global timestamp. The L1 updates its data and local timestamp with values in the response message before completing the load. If the load response
4.2. TC-Strong Coherence

<table>
<thead>
<tr>
<th>Core C1</th>
<th>Core C2</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1: data = NEW</td>
<td>L1: r1 = flag</td>
</tr>
<tr>
<td>F1: FENCE</td>
<td>B1: if (r1 \neq \text{SET}) goto L1</td>
</tr>
<tr>
<td>S2: flag = SET</td>
<td>L2: r2 = data</td>
</tr>
</tbody>
</table>

(a)

Write stalling at L2 (TC-Strong)
Fence waiting for pending requests (both)
Fence waiting for GWCT (TC-Weak)

(b) (c)

Figure 4.3: Detailed Temporal Coherence operation. (a) Code snippet from Sorin et al. [85]. (b) Sequence of events for C1 (left) that occur due to code in (a) and state of C2’s lines (right) for TC-Strong. (c) Sequence of events with TC-Weak.
4.2. TC-Strong Coherence

arrives after the lifetime received for the line has already expired, then the load request will be serviced with the received data but the L1 line will not be updated since it is now invalid. A store request writes through to the L2 where its completion is delayed until the global timestamp has expired.

Figure 4.3(b) illustrates how TC-Strong maintains coherence for the code snippet in Figure 4.3(a). The code snippet is the same code example we saw earlier in Table 2.1 but this time including the memory fence instruction. Figure 4.3(b) shows the memory requests generated by core C1 on the left, and the state of the two memory locations, flag and data, in C2’s L1 on the right. The example assumes that initially C2 has flag and data cached with local timestamps of 60 and 30, respectively. To keep the example simple, we assume that C2’s operations are delayed and do not occur in the time frame covered by this example.

C1 executes instruction S1 and generates a store request to L2 for data (1), and subsequently issues the memory fence instruction F1 (2). F1 defers scheduling the wavefront because the wavefront has an outstanding store request. When S1’s store request reaches the L2 (3), the L2 stalls it because data’s global timestamp will not expire until time T=30. At T=30, C2 self-invalidates data (4), and the L2 processes S1’s store (5). The fence instruction completes when C1 receives the acknowledgment for S1’s request (6). The same sequence of events occurs for the store to flag by S2. The L2 stalls S2’s store request (7) until flag self-invalidates in C2 (8).

Tables 4.1 and 4.2 present TC-Strong’s complete L1 and L2 state machines, respectively. Each table entry lists the actions carried out and the final cache line state for a given event (column heading) and an initial cache line state (row heading). The 4 stable L2 states, I, P, S and E, correspond to invalid lines, lines with one reader, lines with multiple readers, and lines with expired global timestamps, respectively. The LS and LM L2 transient cache states track misses at the L2 for load and store requests. The MI transient coherent state tracks evicted L2 lines with unexpired global timestamps. Note that store requests (GETX and UPGR) at the L2 stall until the L2 line is in the E state. At the L1, the stable I state indicates invalid lines or lines with expired local timestamps, and the stable V state indicates valid local timestamps. The LV and LI transient cache states are used to track load and store misses, while the VM transient coherent state tracks stores requests to valid lines.

**L2 Eviction Optimization.** To maintain inclusion, only expired global timestamps can be evicted from the L2. One approach is to stall the eviction, and hence the request at the L2 that is triggering the eviction, until the candidate line has expired. TC-Strong opts to evict the L2 line imme-
### 4.2. TC-Strong Coherence

Table 4.1: L1 States for TC-Strong Protocol. Shaded regions indicate additions to non-coherent protocol.

<table>
<thead>
<tr>
<th>State</th>
<th>Processor Request</th>
<th>L1 Action</th>
<th>From L2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Load</td>
<td>Store</td>
<td>Atomic</td>
</tr>
<tr>
<td>I</td>
<td>GETS</td>
<td>GETX</td>
<td>ATOMIC</td>
</tr>
<tr>
<td>V</td>
<td>hit</td>
<td>UPGR</td>
<td>ATOMIC</td>
</tr>
<tr>
<td>V_M (upgrade)</td>
<td>hit</td>
<td>UPGR</td>
<td>ATOMIC</td>
</tr>
<tr>
<td>L_V (Rd-miss)</td>
<td>×</td>
<td>GETX</td>
<td>ATOMIC</td>
</tr>
<tr>
<td>L_I (Wr-miss)</td>
<td>GETS</td>
<td>GETX</td>
<td>ATOMIC</td>
</tr>
</tbody>
</table>

**L1⇒L2 messages:** GETS (load), GETX (store), ATOMIC, UPGR (upgrade).

**L2 Triggered Events @ L1:** Data (valid data), Write Ack (store complete from L2).

**L1 Conditionals:** load/store/atomic? (response to GETS/GETX/ATOMIC?). pending 0? (all pending requests satisfied?).
### 4.2. TC-Strong Coherence

Table 4.2: L2 States for TC-Strong Protocol. Shaded regions indicate additions to non-coherent protocol.

<table>
<thead>
<tr>
<th>State</th>
<th>GETS</th>
<th>GETX</th>
<th>UPGR</th>
<th>ATOMIC</th>
<th>L2 Action</th>
<th>From Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>FETCH → L_S</td>
<td>FETCH → L_M</td>
<td>FETCH → L_M</td>
<td>FETCH → L_M</td>
<td>× ×</td>
<td>×</td>
</tr>
<tr>
<td>P (Private)</td>
<td>extend TS DATA</td>
<td>stall (TS==?)</td>
<td>stall</td>
<td>(dirty?) → E WB</td>
<td>×</td>
<td></td>
</tr>
<tr>
<td>S (Shared)</td>
<td>extend TS DATA</td>
<td>stall</td>
<td>stall</td>
<td>stall</td>
<td>(dirty?) → E WB</td>
<td>×</td>
</tr>
<tr>
<td>E (Expired)</td>
<td>extend TS DATA</td>
<td>ACK</td>
<td>ACK</td>
<td>ACK</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>L_S (Read miss)</td>
<td>merge</td>
<td>stall</td>
<td>stall</td>
<td>stall</td>
<td>×</td>
<td>DATA (dirty?)</td>
</tr>
<tr>
<td>L_M (Write miss)</td>
<td>stall</td>
<td>stall</td>
<td>stall</td>
<td>stall</td>
<td>×</td>
<td>DATA (write?)</td>
</tr>
<tr>
<td>M_J (Evicted)</td>
<td>stall</td>
<td>stall</td>
<td>stall</td>
<td>stall</td>
<td>× → I</td>
<td>×</td>
</tr>
</tbody>
</table>

**L2⇒L1 messages:** ACK (store done). DATA (data response).

**L2⇒MEM messages:** FETCH (fetch data from memory). WB (writeback data to memory).

**L2 Conditionals:** TS==? (requester’s L1 timestamp matches current L2 timestamp?). dirty? (L2 data modified?). multiple? (multiple load requests merged?).

**L2 Timestamp Actions:** extend TS (extend L2 timestamp according to request).
4.2. TC-Strong Coherence

diately to prevent stalling, and instead tracks the unexpired timestamps in an external structure. It uses unused L2 MSHR entries to store unexpired timestamps. No modifications to the L2 MSHR table are needed specifically for this optimization. Note that evicted timestamps may use all the free MSHR entries and introduce resource stalls. Unlike the resource stalls in traditional coherence protocols like MESI and GPU-VI, these stalls cannot lead to protocol deadlocks. This is because TC protocols do not have the cyclic resource dependency issues that are introduced by coherence messages. These stalls are akin to the MSHR stalls that can occur even in the baseline non-coherent GPU.

Private Write Optimization. TC-Strong implements an optimization to eliminate write-stalling for private data. It differentiates the single valid L2 state into two stable states, \( P \) and \( S \). The \( P \) state indicates private data while the \( S \) state indicates shared data. An L2 line read only once exists in \( P \). Stores to L2 lines in \( P \) are private writes if they are from the core that originally performed the read. In TC-Strong, store requests carry the local timestamp at the L1, if it exists, to the L2. This timestamp is matched to the global timestamp at the L2 to check that the core that originally performed the read is performing a private write. The two requirements that a single core has this line (the line being in \( P \) state) and that the timestamp at the L1 matches that at the L2 ensure that the original core is performing a private write. This optimization is visible in Table 4.2. An UPGR event in the \( P \) state completes without stalling if there is a timestamp match between the L1 and L2 lines.

4.2.2 TC-Strong and LCC comparison

Both LCC and TC-Strong use time-based self-invalidation and require synchronized counters and timestamps in L1 and L2. Both protocols stall stores at the last level cache to unexpired timestamps.

TC-Strong requires minimal hardware modifications to the baseline non-coherent GPU architecture. It supports multiple outstanding store requests per GPU wavefront. In contrast, LCC assumes only one outstanding store request per core. LCC implements Sequential Consistency and this restriction eases an SC implementation. The restriction also permits the use of a memory system that can aggressively reorder memory requests. TC-Strong implements a relaxed memory model, does not restrict each GPU core to a single in-flight request, and provides much greater memory-level parallelism for the thousands of concurrent scalar threads per GPU core. It relies on
4.3 TC-Weak Coherence

This section describes TC-Weak. TC-Weak relaxes the write atomicity of TC-Strong. As we show in Section 6.4, doing so improves performance by 28% and lowers interconnect traffic by 26% compared to TC-Strong.

TC-Strong and LCC enforce coherence across all data by stalling stores. This presents two problems. First, it makes performance more sensitive to the lifetime prediction mechanism. A misprediction where a large lifetime is predicted for an actually small lifetime will cause the store operation to stall unnecessarily. Second, since our baseline GPU implements an in-order memory system, stalled store operations will block all L2 requests behind them. This hurts overall system performance as requests from one GPU core may impede requests from other GPU cores. TC-Weak uses the insight that by relaxing the write-atomicity property a store operation does not need to be stalled. Instead, it relies on a novel fence mechanism that guarantees all preceding store operations from a wavefront have completed before the fence instruction completes. This provides two main benefits. First, it eliminates expensive stalling at the shared L2 cache controllers, which affects all cores and wavefronts, and shifts it to scheduling of individual wavefronts at memory fences. A wavefront descheduled due to a memory fence does not affect the performance of other wavefronts. Second, it enforces coherence only when required and specified by the program through memory fences. It relies on the data-race free requirement of the SC for DRF paradigm to ensure that the required orderings are enforced through fence operations. It implements, RCpc [35], the variant of Release Consistency that does not require write atomicity.

In TC-Weak, stores to unexpired global timestamps at the L2 do not stall. Instead, the store data is immediately written to the L2 line and the store response returns with the L2 line’s global timestamp. The returned global timestamp is the guaranteed time by which the store will become
visible to all cores in the system. This is because by this time all cores will have invalidated their privately cached stale copies. TC-Weak tracks the global timestamps returned by stores, called Global Write Completion Time (GWCT), for each wavefront. A memory fence operation uses this information to deschedule the wavefront sufficiently long enough to guarantee that all previous stores from the wavefront have become globally visible.

As illustrated in Figure 4.2(a), TC-Weak adds a small GWCT table to each GPU core. The GWCT table contains 48 entries, one for each wavefront in a GPU core. Each entry holds a timestamp value which corresponds to the maximum of all GWCTs observed for that wavefront.

### 4.3.1 TC-Weak Operation

A memory fence in TC-Weak deschedules a wavefront until all pending store requests from the wavefront have returned acknowledgments, and until the wavefront’s timestamp in the GWCT table has expired. The latter ensures that all previous stores have become visible to the system by fence completion.

Figure 4.3(c) illustrates how coherence is maintained in TC-Weak by showing the execution of C1’s memory instructions from Figure 4.3(a). C1 executes S1 and sends a store request to the L2 for data (1). Subsequently, C1 issues a memory fence operation (2) that defers scheduling of the wavefront because the instruction S1’s memory request is outstanding. The L2 receives the store request (3) and returns the current global timestamp stored in the L2 for data. In this case, the value returned is 30 and corresponds to C2’s initially cached copy. The L2 does not stall the store, immediately updates the data in the L2 line, and sends back an acknowledgment with the GWCT, which updates the C1’s GWCT entry for this wavefront. After C1 receives the acknowledgment (4), no memory requests are outstanding. The scheduling of the wavefront is now deferred because the GWCT entry of this wavefront containing a timestamp of 30 has not yet expired. As data self-invalidates in C2’s cache (5), the wavefront’s GWCT expires and the fence is allowed to complete (6). The next store instruction, S2, sends a store request (6) to the L2 for flag. The L2 returns a GWCT time of 60 (7), corresponding to the copy cached by C2.

Comparing Figure 4.3(c) to 4.3(b) shows that TC-Weak performs better than TC-Strong because it only stalls at explicit memory fence operations. Relaxing write atomicity eliminated the unnecessary stall to the update to flag and improved the performance of the code sequence in Figure 4.3(a).
### 4.3. TC-Weak Coherence

Table 4.3: L1 States for TC-Weak Protocol. Shaded regions indicate additions to non-coherent protocol.

<table>
<thead>
<tr>
<th>State</th>
<th>Processor Request</th>
<th>L1 Action</th>
<th>From L2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Load</td>
<td>Store</td>
<td>Atomic</td>
</tr>
<tr>
<td>I</td>
<td>GETS</td>
<td>GETX</td>
<td>ATOMIC</td>
</tr>
<tr>
<td>V</td>
<td>hit</td>
<td>UPGR</td>
<td>ATOMIC</td>
</tr>
<tr>
<td>V_M  (upgrade)</td>
<td>hit</td>
<td>UPGR</td>
<td>ATOMIC</td>
</tr>
<tr>
<td>L_V  (Rd-miss)</td>
<td>×</td>
<td>GETX</td>
<td>ATOMIC</td>
</tr>
<tr>
<td>L_4  (Wr-miss)</td>
<td>GETS</td>
<td>GETX</td>
<td>ATOMIC</td>
</tr>
</tbody>
</table>

**L1⇒L2 messages:** GETS (load). GETX (store). ATOMIC. UPGR (upgrade).

**L2 Triggered Events @ L1:** Data (valid data). Write Ack (store complete from L2).

**L1 Conditionals:** load/store/atomic? (response to GETS/GETX/ATOMIC?). global? (response includes GWCT?). pending 0? (all pending requests satisfied?).
### 4.3. TC-Weak Coherence

#### Table 4.4: L2 States for TC-Weak Protocol. Shaded regions indicate additions to non-coherent protocol.

<table>
<thead>
<tr>
<th>State</th>
<th>L1 Request</th>
<th>L2 Action</th>
<th>From Mem Mem Data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>GETS → Ls</td>
<td>GETX → LM</td>
<td>Evict</td>
</tr>
<tr>
<td>I</td>
<td>FETCH</td>
<td>FETCH → LM</td>
<td>×</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Fetch X → LM</td>
<td></td>
</tr>
<tr>
<td>P</td>
<td>extend TS</td>
<td>TS++ (TS==?)</td>
<td>(dirty?)</td>
</tr>
<tr>
<td></td>
<td>DATA</td>
<td>ACK-G</td>
<td>ACK-G</td>
</tr>
<tr>
<td></td>
<td>→ S</td>
<td></td>
<td>else - DATA-G</td>
</tr>
<tr>
<td>S</td>
<td>extend TS</td>
<td>TS++</td>
<td>TS++ (TS==?)</td>
</tr>
<tr>
<td></td>
<td>DATA</td>
<td>ACK-G</td>
<td>ACK-G</td>
</tr>
<tr>
<td></td>
<td>→ P</td>
<td></td>
<td>else - DATA-G</td>
</tr>
<tr>
<td>E</td>
<td>extend TS</td>
<td>TS++</td>
<td>TS++</td>
</tr>
<tr>
<td></td>
<td>DATA</td>
<td>ACK</td>
<td>ACK</td>
</tr>
<tr>
<td></td>
<td>→ P</td>
<td></td>
<td>else -</td>
</tr>
<tr>
<td>Ls</td>
<td>merge</td>
<td>stall</td>
<td>stall</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lm</td>
<td>stall</td>
<td>stall</td>
<td>stall</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mj</td>
<td>FETCH → Ls</td>
<td>FETCH → LM</td>
<td>FETCH → LM</td>
</tr>
</tbody>
</table>


**L2⇒MEM messages:** FETCH (fetch data from memory). WB (writeback data to memory).

**L2 Conditionals:** TS==? (requester’s L1 timestamp matches pre-incremented L2 timestamp?). dirty? (L2 data modified?). multiple? (multiple load requests merged?).

**L2 Timestamp Actions:** extend TS (extend L2 timestamp according to request). TS++ (increment L2 timestamp).
4.4. Lifetime Prediction

Tables 4.3 and 4.4 present TC-Weak’s complete L1 and L2 state machines, respectively. Each table entry lists the actions carried out and the final cache line state for a given event (column heading) and an initial cache line state (row heading). The 4 stable L2 states, I, P, S and E, correspond to invalid lines, lines with one reader, lines with multiple readers, and lines with expired global timestamps, respectively. The I\_S and I\_M L2 transient cache states track misses at the L2 for load and store requests. The M\_I transient coherent state tracks evicted L2 lines with unexpired global timestamps. Note the lack of transient states and stalling at the L2 for stores to valid (P, S and E) lines. At the L1, the stable I state indicates invalid lines or lines with expired local timestamps, and the stable V state indicates valid local timestamps. The I\_V and I\_I transient cache states are used to track load and store misses, while the V\_M transient coherent state tracks stores requests to valid lines.

Private Write Optimization. To ensure that memory fences are not stalled by stores to private data, TC-Weak uses a private write optimization similar to the one employed by TC-Strong and described in Section 4.2.1. Store requests to L2 lines in the P state where the L1 local timestamp matches the L2 global timestamp indicate private writes and do not return a GWCT. Since TC-Weak does not stall stores at the L2, an L2 line in P may correspond to multiple unexpired but stale L1 lines. Stores in TC-Weak always modify the global timestamp by incrementing it by one. This ensures that a store request from another L1 cache with stale data carries a local timestamp that mismatches with the global timestamp at the L2, and that the store response replies with the updated data.

4.4 Lifetime Prediction

Predicted lifetimes should not be too short that L1 lines are self-invalidated too early, and not too long that storing evicted timestamps wastes L2 cache resources and potentially introduces resource stalls. In Section 6.6 we show that a single lifetime value for all accesses performs well. Moreover, this value is application dependent. Based on this insight, we propose a simple lifetime predictor that maintains a single lifetime prediction value at each L2 cache bank, and adjusts it based on application behaviour. A load obtains its lifetime prediction at the L2 bank.

The predictor updates the predicted lifetime based on events local to the L2 bank. First, the local prediction is decreased by \( t_{\text{evict}} \) cycles if an L2 line with an unexpired timestamp is evicted. This reduces the number
4.5. Timestamp Rollover

of timestamps that need to be stored past an L2 eviction. Second, the local prediction is increased by $t_{hit}$ cycles if a load request misses at the L1 due to an expired L1 line. This helps reduce L1 misses due to early self-invalidation. The lifetime is also increased by $t_{hit}$ cycles if the L2 receives a load request to a valid line with an expired global timestamp. This ensures that the prediction is increased even if L1 lines are quickly evicted. Third, the lifetime is decreased by $t_{write}$ cycles if a store operation writes to an unexpired line at the L2. This helps reduce the amount of time that fence operations wait for the GWCT to expire, i.e., for stores to become globally visible. This third mechanism is disabled for applications not using fences to keep it from unnecessarily harming the L1 miss rate. Table 5.1 lists the parameter values used in our evaluation; we found these to yield the best performance across all applications.

Also note that the lifetime prediction mechanism is completely independent of the protocol’s correctness. TC-Strong and TC-Weak do not place any restrictions on the values that can be predicted. In traditional coherence protocols, optimizations are generally introduced at the cost of additional states and complexity. The choice of prediction mechanism has no effect on the complexity of a TC protocol. TC can allow performance improvements through better prediction mechanisms and without having to modify and re-verify the coherence implementation.

4.5 Timestamp Rollover

L1 lines in the valid state but with expired timestamps may become unexpired when the global time counters rollover. This could be handled by simply flash invalidating the valid bits in the L1 cache [79]. More sophisticated approaches are possible, but beyond the scope of this work. None of the benchmarks we evaluate execute long enough to trigger an L1 flush with 32-bit timestamps.
Chapter 5

Methodology

This chapter describes the simulation infrastructure and benchmarks that we use in this study.

5.1 Simulation Infrastructure

We modeled a cache coherent GPU architecture by extending GPGPU-Sim version 3.1.2 [12] with the Ruby memory system model from GEMS [62]. Ruby was selected because it provides the easy-to-use SLICC interface to specify coherence protocols. Interfacing Ruby with GPGPU-Sim required changes to the core infrastructure of Ruby.

First, Ruby was designed to support only a single outstanding request per address per core. This assumption was in place because CMP systems are mostly evaluated with one thread per core and multiple requests to the same address from the same thread are rare. This is not true for GPUs where the hundreds of threads in each core may access the same memory address. To properly model a GPU memory system, we modified Ruby’s back-end to support multiple accesses to the same address from the same core. The original implementation in Ruby tracked requests through their (address, core ID) pairs. This was sufficient to track every unique in-flight request due to the one request per address per core restriction. Removing the restriction means two different requests from different wavefronts may alias to the same (address, core ID) pair. We modified Ruby to track requests using unique IDs that are different for each request. Tracking via unique IDs removes any theoretical restrictions on the number of in-flight requests.

Second, Ruby did not support a way to allow certain requests to skip L1 caches, as is needed for the NO-L1 configuration in our study. We supported this by introducing additional queues at the interface between GPGPU-Sim and Ruby that allow requests to skip Ruby’s internal structures. Our L1 state machine for our baseline non-coherent protocol handles these requests differently by forwarding them directly to the L2 cache.

Finally, the original DRAM model in Ruby was idealistic and only modeled the now obsolete DDR-400 memory chips. We modified Ruby to sup-
port the GDDR5 DRAM model present in GPGPU-Sim. We did so by implementing a second interface whereby requests inside Ruby travelling from the L2 cache to off-chip memory are returned back to GPGPU-Sim, processed by GPGPU-Sim’s GDDR5 DRAM model, and sent back to Ruby’s L2 controller. Ruby’s code had 3 core components: L1 caches, L2 caches and main memory. All three connect through a single interconnection network. Since the interconnection network in our baseline GPU does not connect to DRAM controllers, we removed the memory components from Ruby and limited the interconnection network to only L1 and L2 caches. This provided more accurate power data by removing the unused physical links to DRAM controllers.

The baseline non-coherent memory system and all coherence protocols are implemented in SLICC. The MESI cache coherence protocol is acquired from gem5 [14]. Our GPGPU-Sim extended with Ruby is configured to model a generic NVIDIA Fermi GPU [68]. We use Orion 2.0 [47] to estimate the interconnect power consumption. The interconnection network is modelled using the detailed fixed-pipeline network model in Garnet [8]. Two crossbars, one per direction, connect the GPU cores to the memory partitions. Each crossbar can transfer one 32-byte flit per interconnect cycle to/from each memory partition for a peak bandwidth of $\sim 175$GB/s per direction. GPU cores connect to the interconnection network through private ports. The baseline non-coherent and all coherence protocols use the detailed GDDR5 DRAM model from GPGPU-Sim. Minimum L2 latency of 340 cycles and minimum DRAM latency of 460 cycles (in core cycles) is modelled to match the latencies observed on Fermi GPU hardware via microbenchmarks released by Wong et al. [89]. Table 5.1 lists other major configuration parameters. All of the code use in this study, including our benchmarks, is available on the web [4].

5.2 Benchmarks

We used two sets of benchmarks for evaluation: one set contains inter-workgroup communication and requires coherent caches for correctness, and the other only contains intra-workgroup communication. While coherence can be disabled for the latter set, we kept coherence enabled and used this set as a proxy for future workloads which contain both data needing coherence and data not needing it. The following benchmarks fall into the former set:
5.2. Benchmarks

Table 5.1: Simulation Configuration.

<table>
<thead>
<tr>
<th>GPGPU-Sim Core Model</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td># GPU Cores</td>
<td>16</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Ruby Memory Model</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 Private Data $</td>
<td>32KB, 4way, 128B line, 4-way assoc. 128 MSHRs.</td>
</tr>
<tr>
<td># Mem. Partitions</td>
<td>8</td>
</tr>
<tr>
<td>Virtual Channels</td>
<td>8-flit buffer per VC.</td>
</tr>
<tr>
<td>GDDR Clock</td>
<td>1400 MHz</td>
</tr>
<tr>
<td>Memory Channel BW</td>
<td>8 (Bytes/Cycle) (175GB/s peak). Minimum Latency: 460 cycles.</td>
</tr>
<tr>
<td>DRAM Queue Capacity</td>
<td>32, Out-of-Order (FR-FCFS).</td>
</tr>
<tr>
<td>GDDR5 Memory Timing</td>
<td>$t_{CL}$=12, $t_{RP}$=12, $t_{RC}=40$, $t_{RAS}=28$, $t_{CCD}=2$, $t_{WL}=4$, $t_{RD}=12$, $t_{RCD}=12$, $t_{DRQ}=6$, $t_{CDL}=5$, $t_{WR}=12$, $t_{CCDL}=3$, $t_{RTPL}=2$.</td>
</tr>
<tr>
<td>TC-Weak Parameters</td>
<td></td>
</tr>
<tr>
<td>Timestamp Size</td>
<td>32 bits</td>
</tr>
<tr>
<td>Predictor Constants</td>
<td>$t_{evict}=8$ cycles, $t_{hit}=4$ cycle, $t_{write}=8$ cycles.</td>
</tr>
</tbody>
</table>
Table 5.2: Benchmarks.

<table>
<thead>
<tr>
<th>Inter-workgroup communication</th>
<th>Intra-workgroup communication</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Name</td>
</tr>
<tr>
<td>Barnes Hut [18]</td>
<td>HotSpot [21]</td>
</tr>
<tr>
<td>Cloth Physics [17]</td>
<td>3D Laplace Solver [12]</td>
</tr>
<tr>
<td>Stencil (Wave Propagation)</td>
<td>Gaussian Filter [1]</td>
</tr>
<tr>
<td>Versatile Place and Route</td>
<td>Anisotropic Diffusion [21]</td>
</tr>
</tbody>
</table>

**Barnes Hut (BH)** implements the Barnes Hut n-body algorithm, widely used in simulating the motion of galaxies, in CUDA [18]. The Barnes Hut algorithm makes the problem of calculating forces between bodies tractable by dividing space into cells, computing summary information for bodies in each cell, and using the summary information to approximate the forces on each body. The division of space into cells is accomplished by building an octree where the leaf nodes are bodies and each parent node is the larger cell that contains child nodes corresponding to smaller cells or bodies. In this thesis we focus on the octree-building kernel which uses fine-grained locking to insert nodes. Coherence is needed to ensure that updates to the tree become visible to all threads that may be caching the stale state. We report data for the octree-building kernel with an input of 30000 bodies.

**CudaCuts (CC)** implements the maxflow/mincut algorithm for image segmentation in CUDA [88]. The algorithm is used in Computer Vision problems like video segmentation and multi-camera scene reconstruction. This implementation uses the push-relabel algorithm to find the minimum cut in a 2D stencil graph where each node represents a pixel in an image. The push operation pushes excess flow at each node to its neighbouring nodes with lower heights, limited by the edge capacities. The relabel operation increases nodes’ heights to enable push operations at nodes with excess flow. The original implementation by Vineet et al. [88] splits the push operation into push and pull kernels, citing the lack of inter-workgroup synchronization needed for communication between nodes in different workgroups. We optimized CC by utilizing a coherent memory space to combine the push, pull and relabel operations into a single kernel, improving performance by 30% as a result.
5.2. Benchmarks

**Cloth Physics (CL)** is a cloth physics simulation based on “RopaDemo” [17]. Each time step in the simulation consists of two computational steps: calculating new particle positions by solving equations of motion, and adjusting the particle positions to fit constraints defined by the cloth’s properties. Each constraint defines the maximum and minimum distance between two particles. The constraint-solving kernel solves all constraints in parallel, using fine-grained locks to ensure that two constraints moving the same particle are serialized. Coherence is needed to ensure that updates to particles become visible to all threads that may be caching the stale position.

**Dynamic Load Balancing (DLB)** implements task-stealing in CUDA [19]. It uses non-blocking task queues to load balance the partitioning of an octree. Each task takes in a set of nodes as input, partitions the nodes into 8 subsets according to their spacial positions, and inserts into a task queue 8 new tasks to further partition each of the 8 subsets. Each workgroup works on a single task at a time, dividing computation among its threads. Coherence is needed to ensure that updates to the shared task queues and the buffers used to track sets of nodes are visible across all threads. We report data for an input graph size of 100000 nodes.

**Stencil (STN)** uses stencil computation to implement a finite difference solver for 3D wave propagation, which is used in applications like seismic imaging and fluid dynamics. At each time step, a node reads the latest value from its 24 adjacent neighbours (8 neighbours in each of the 3 dimensions), and produces a new value of its own. Each workgroup processes a subset of the stencil nodes. A coherent memory space ensures that updates to neighbours in a different subset become visible across GPU cores. The persistent thread model is used to eliminate the need for launching a new kernel for each time step. STN uses fast barriers [90] to synchronize workgroups between time steps. Coherence is also needed for the inter-workgroup synchronization.

**Versatile Place and Route (VPR)** is a placement tool for FPGAs. We ported the simulated annealing based placement algorithm from VTR 1.0 [76] to CUDA. The algorithm swaps two randomly selected blocks if the operation decreases the placement cost function. We parallelized this algorithm on two fronts. First, multiple swaps are processes in parallel, one per wavefront. Fine-grained locks ensure that the same location is not processed by two parallel swap operations. Second, threads within a wavefront compute the bounding-box cost function in parallel. Combined, the parallel
VPR performs 4x faster on GPU hardware (with L1 caches disabled) over the serial CPU version. Coherent memory is needed to ensure that swapped locations become visible to all threads. We simulate one iteration in the annealing schedule for the bgm circuit.

The set of benchmarks with intra-workgroup communication is chosen from the Rodinia benchmark suite [21], benchmarks used by Bakhoda et al. [12] and the CUDA SDK [1]. These benchmarks were selected to highlight a variety of behaviours; we did not exclude any benchmarks where TC-Weak performed worse than other protocols. All benchmarks we evaluate are listed in Table 5.2. The code for the applications created or modified for this study can be found online [4].
Chapter 6

Results

This section compares the performance of the coherence protocols on a GPU. Section 6.1 presents performance data, Section 6.2 presents traffic data, and Section 6.3 presents power and energy data. Section 6.4 compares TC-Weak to TC-Strong. Section 6.5 evaluates the two optimizations we introduce in TC-Strong. Section 6.6 presents performance data for TC-Weak with a sweep of different lifetimes. Section 6.7 explores the impact of scaling GPU-VIni’s directory size on performance and traffic. Finally, Section 6.8 presents a sensitivity analysis for our proposed predictor. The TCW configuration implements TC-Weak with the lifetime predictor described in Section 4.4.

6.1 Performance

Figure 6.1(a) compares the performance of coherence protocols against a baseline GPU with L1 caches disabled (NO-L1) for applications with inter-workgroup communication. Figure 6.1(b) compares them against the non-coherent baseline protocol with L1 caches enabled (NO-COH) for applications with intra-workgroup communication. In Figure 6.1(a), on average the coherence protocols have similar performance to each other and better performance than the baseline non-coherent GPU with L1 caches disabled. TCW performs 85% better than the baseline GPU for this set of applications.

MESI performs slightly better than the write-through protocols for the applications BH and CL. Both BH and CL make extensive use of fine-grained locking, which uses atomic or read-modify-write operations. In MESI, we model read-modify-write operations at the L1. In the write-through protocols, they are modelled in an atomic operation unit at the L2. BH and CL benefit from the merging of atomic operations at the L1 in MESI. VPR also uses locks, but has very low contention, contains fewer atomic accesses, and does not gain a performance benefit from L1 atomics.

On CC and VPR, TCW does not perform as well as the other write-through protocols, GPU-VI and GPU-VIni. Here, predictor settings that are optimal for average performance across all benchmarks are suboptimal.
6.1. Performance

![Graph showing performance comparison]

(a) Inter-workgroup comm.  
(b) Intra-workgroup comm.

Figure 6.1: Performance of coherent and non-coherent GPUs. HM=harmonic mean.

for these two applications. Section 6.8 shows that using a different set of parameters for the predictor brings the performance of TCW up to match the performance of the write-through protocols. Specifically, CC benefits from smaller increments in lifetime prediction with load misses at the L1 and VPR benefits from larger increments in lifetime. A more flexible predictor would be beneficial for these two applications.

DLB achieves a significant speedup with TCW over other coherence protocols. This is because DLB’s performance is sensitive to invalidation latency. TCW eliminates invalidations, and hence the invalidation latency, and achieves $\sim 2x$ speedup. Figure 6.2 shows DLB’s performance with varying interconnect latencies for IDEAL-VI, GPU-VI and TCW. IDEAL-VI implements GPU-VI with ideal invalidations and recalls, i.e., invalidations and recalls have no latency and do not introduce interconnect traffic. TCW performs as well as IDEAL-VI, while GPU-VI’s performance suffers at higher interconnect latencies. GPU-VI’s performance improves as the invalidation latency is decreased. DLB’s sensitivity to invalidation latency exists for two reasons. First, each workgroup fetches for execution a single task at a time. This task is divided among all threads within the workgroup. The task is fetched from shared task queues; updating the task queues may generate invalidation messages. The invalidation latency is on the critical path of an entire workgroup, i.e., all threads in the workgroup are idle when a task queue is being updated. Second, DLB builds an octree by partitioning a set
6.1. Performance

Figure 6.2: Performance of DLB with varying interconnect latencies. Latencies decrease to the right.

of nodes into 8 octants and generating 8 new tasks to recursively partition the 8 subsets. When a child task starts on a different GPU core than its parent, it sends an invalidation with every data access because its data is cached on the parent’s L1. The excessive number of invalidations in such a case hurts performance.

In Figure 6.1(b), the write-through protocols perform as well as the non-coherent baseline when it comes to applications that do not require coherence. MESI suffers a 25% slowdown compared to the baseline. This is a result of MESI’s L1 writeback write-allocate policy which favours store locality but introduces unnecessary traffic for write-once access patterns common in existing GPU applications. Section 6.2 shows the large traffic overhead in MESI for this set of applications. On the application HSP, the traditional coherence protocols suffer a performance loss while TCW achieves the same performance as the non-coherent baseline. HSP contains false sharing, which introduces unnecessary invalidations that lower the L1 hit rate and introduce additional traffic. TCW does not suffer from false invalidations.

Both Figures 6.1(a) and 6.1(b) show that the potentially larger effective cache capacity in non-inclusive GPU-VIni adds no performance benefit over the inclusive GPU-VI. Section 6.7 describes how the performance also remains unaffected as the directory size is increased in GPU-VIni.
6.2 Interconnect Traffic

Figures 6.3(a) and 6.3(b) show the breakdown of interconnect traffic between different coherence protocols. LD, ST, and ATO are the data traffic from load, store, and atomic requests. MESI performs atomic operations at the L1 cache and this traffic is included in ST. REQ refers to control traffic for all protocols. INV and RCL are invalidation and recall traffic, respectively.

MESI’s write-allocate policy at the L1 significantly increases store traffic due to unnecessary refills of write-once data. On average, MESI increases interconnect traffic over the baseline non-coherent GPU by 75% across all applications. The write-through GPU-VI and GPU-VIni introduce unnecessary invalidation and recall traffic, averaging to a traffic overhead of 31% and 30% for applications without inter-workgroup communication. TCW removes all invalidations and recalls and as a result reduces interconnect traffic by 56% over MESI, 23% over GPU-VI and 23% over GPU-VIni for this set of applications.

BH and CL in Figure 6.3(a) have significantly greater store traffic on MESI than they do on other protocols. The store traffic in MESI includes the atomic traffic since atomic operations are performed in the L1. The store traffic is higher in MESI because the entire cache line is brought into the L1, even if the atomic operation is for a single word within the cache line. The traffic for TCW in CC and VPR is higher than other write-through protocols. This is due to suboptimal predictor settings for these two applications that cause more load misses at the L1. Figure 6.3(a) shows that coherence traffic is not a large overhead for GPU-VI on applications that require coherence. These applications have considerate data locality, as indicated by the performance gain from enabling L1 caches. As a result, contention at the directory is reduced and recall traffic is low.

In Figure 6.3(b), the HSP benchmark observes significant reduction in traffic on TCW. While HSP does not contain inter-workgroup communication, it contains false sharing as indicated by the large amount of INV traffic on MESI and the two GPU-VIs. TC-Weak eliminates the negative impacts of false sharing, such as invalidation traffic, on data that does not require coherence. For the applications in Figure 6.3(b), the coherence traffic overhead is greater because data locality is lower and more requests thrash the coherence directory. This results in the recall traffic overhead. TCW eliminates this traffic overhead.

To provide further insight into the increased store traffic usage in MESI, Figure 6.4 shows the detailed breakdown of MESI’s store traffic. The two ST_PUT portions refer to traffic used to transfer evicted L1 lines to the
6.2. Interconnect Traffic

Figure 6.3: Breakdown of interconnect traffic for coherent and non-coherent GPU memory systems.
6.2. Interconnect Traffic

![Detailed breakdown of interconnect traffic for MESI.](image)

Figure 6.4: Detailed breakdown of interconnect traffic for MESI.

L2. ST_PUT_M indicates eviction of L1 data that was modified, while ST_PUT_C refers to the portion of the L1 line that was not modified. The two ST_RFL portions refer to the refill traffic used to allocate L1 lines on stores misses at the L1. ST_RFL_OW indicates data that will be overwritten when the store is completed at the L1, while ST_RFL_C indicates the portion of ST_RFL data that will not be overwritten.

Of these 4 portions, ST_RFL_C, ST_RFL_OW and ST_PUT_C are the potentially unnecessary traffic overheads in MESI. ST_RFL_C is the data not referenced by a store operation but brought into the L1 to allocate a complete cache line. When store locality is low, this data is not used and represents a waste of traffic. ST_RFL_OW is the data that is referenced by a store operation and brought into the L1 during a store allocate. It will be overwritten after the store completes at the L1. A single GPU store instruction often overwrites the complete L1 line, but MESI still wastes traffic bringing the line data into the L1. ST_PUT_C are the unmodified words of an L1 cache line sent to the L2 during an L1 eviction. This is unnecessary as the L2 contains the unmodified data. Figure 6.4 shows that these overheads account for a large portion of MESI’s store traffic.

These store traffic overheads result from the store locality optimizing features of MESI. One example of such a feature is the write allocate nature of MESI. Another example is the Exclusive, or E, state that optimistically acquires write permissions to a line when a core is the first to read it. In both cases MESI sacrifices additional traffic to capture store locality at the L1. We observe that store locality of GPU applications is low. Hence, MESI’s
additional store traffic cannot be amortized over future stores requests. One reason for the GPU’s reduced store locality is the flexible register allocation scheme that avoids spilling registers by reducing the number of concurrently executing wavefronts. Secondly, thousands of GPU threads share a small L1 cache. The resulting high contention at the L1 makes it difficult to keep store-allocated lines in the L1. Lastly, many GPU applications optimize for intra- and inter-thread communication through shared memory. This further reduces store locality in the cached global memory.

6.3 Power

Figure 6.5 shows the breakdown of interconnect power and energy usage. The breakdown shows the dynamic and static power used by the routers, and the dynamic and static power used in link traversal. TCW lowers the interconnect power usage by 21%, 10% and 8%, and interconnect energy usage by 36%, 13% and 8% over MESI, GPU-VI and GPU-VIni, respectively. Two factors contribute to the power savings observed in TCW. First, dynamic power is reduced in TCW due to the lower interconnect traffic. This saving comes through link power and router power as both experience less activity. Second, TC protocols require only 2 virtual channels, compared to the 5 required by MESI and 4 required by GPU-VI and GPU-VIni. This leads to fewer virtual channel buffers in the routers and lower static router power. Static link power remains the same across configurations as it has
6.4. TC-Weak vs. TC-Strong

Figure 6.6: (a) Harmonic mean speedup and (b) normalized average interconnect traffic for single fixed timestamp prediction across all applications for TC-Strong and TC-Weak. TCW uses dynamic predictor instead of single fixed timestamps.

no relation to the number of virtual channels. Note that the baseline GPU with L1 caches disabled for inter-workgroup application experiences lower power usage than the configurations with coherence and L1 caches enabled. The average power is skewed in this case because the execution time for the baseline GPU is much greater than the cache coherent configurations. The energy usage is a better indicator in this case and shows that the coherent configurations have lower energy usage.

6.4 TC-Weak vs. TC-Strong

Figures 6.6(a) and 6.6(b) compare the harmonic mean performance and average interconnect traffic, respectively, for configurations of TC-Strong and TC-Weak that use a single fixed lifetime prediction for all applications. This is equivalent to the FIXED-DELTA prediction scheme proposed in LCC [57, 82], which selects a single fixed lifetime that works best across all applications. TCSUO-FIXED implements TC-Strong without the two optimizations we propose. TCS-FIXED implements TC-Strong with those two optimizations. TCSOO-FIXED allows memory operations to be reordered at the L2 cache to reduce the performance impact of stalled store requests impeding other requests. All three TCS configurations use a fixed lifetime prediction of 400 core cycles, which was found to yield the best harmonic mean performance over other lifetime values. TCW-FIXED implements TC-Weak and uses a fixed lifetime prediction of 1600 core cycles, which
was found to be best performing over other fixed lifetime values. TCW implements TC-Weak with the proposed predictor, as before.

The two optimizations we propose for TC-Strong improve performance by an average of 4% in TCS-FIXED over TCSUO-FIXED. Allowing re-ordering of requests at the L2 controller in TC-Strong further improves performance by 5% in TCSOO-FIXED. TCW-FIXED has the same predictor as TCS-FIXED but outperforms it by 15% while reducing traffic by 13%. TC-Strong has a trade-off between additional write stalls with higher lifetimes and additional L1 misses with lower lifetimes. TC-Weak avoids this trade-off by not stalling stores. This permits longer lifetimes and fewer L1 misses, improving performance and reducing traffic over TC-Strong. With the novel predictor proposed in this work, TCW achieves a 28% improvement in performance over TCS-FIXED and reduces interconnect traffic by 26%. The dynamic prediction mechanism is able to select different lifetimes for different applications and improves performance over the static fixed lifetime prediction. Section 6.6 shows that each application prefers a different lifetime for better performance.

Figures 6.7(a) and 6.7(b) show the performance and interconnect traffic for TC-Strong and TC-Weak configurations, but this time selecting the fixed lifetime on a per-application basis. They select the fixed lifetime prediction that gives the best performance for each application. TCW performs as well as TCW-BEST, indicating that the predictor we propose is able to dynamically select the best lifetime for each application. TCW-BEST outperforms
6.5. TC-Strong Optimizations

Figure 6.8: (a) Harmonic mean speedup and (b) normalized average interconnect traffic for TC-Strong with and without proposed optimizations. The X-axis lists the single fixed lifetime prediction values used across all applications.

TC-BEST by 12% and reduces interconnect traffic by 17%. Even with good lifetime prediction in TC-Strong, a performance gap between TC-Strong and TC-Weak remains.

6.5 TC-Strong Optimizations

Figures 6.8(a) and 6.8(b) show the impact of the two optimizations proposed for TC-Strong described in Section 4.2.1. As before, TCSUO-FIXED implements TC-Strong without the optimizations and TCS-FIXED with them. Both use the FIXED-DELTA \[57, 82\] prediction scheme which uses a single fixed value for all lifetime predictions. The X-axes in Figures 6.8(a) and 6.8(b) list the lifetime prediction value used for each configuration. In Figure 6.8(a), at the optimal fixed prediction of 400 cycles, the proposed optimizations improve performance only by 4%. With higher lifetime values, the performance gap increases and the optimizations make TC-Strong more resilient to performance loss. The first optimization removed stalling when unexpired L2 lines are evicted. The second optimization reduced stalling of store operations to private data. Figure 6.8(a) shows that these two optimizations benefit performance when over-prediction increases the frequency of these stalls. Figure 6.8(b) shows that the optimizations do not impact interconnect traffic usage. Both optimizations do not add or remove any messages, so this result is expected.
6.6 TC-Weak Performance Profile

Figure 6.9 presents the performance of TC-Weak with various fixed lifetime prediction values for the entire duration of the application. The downward arrows in Figure 6.9 indicate the average lifetime predictions in TCW. An increase in performance with increasing lifetimes results from an improved L1 hit rate. A decrease in performance with larger lifetimes is a result of stalling fences and L2 resource stalls induced by storage of evicted but unexpired timestamps. Note that in DLB, TCW-FIXED with a lifetime of 0 is 3x faster than NO-L1 because use of L1 MSHRs in TCW-FIXED reduces load requests by 50% by merging redundant requests across wavefronts. The performance profile yields two main observations. First, each application prefers a different fixed lifetime. For example, NDL’s streaming access pattern benefits from a short lifetime, or an effectively disabled L1. Conversely, HSP prefers a large lifetime to fully utilize the L1 cache. Second, the arrows indicating TCW’s average lifetime lie close to the peak performance lifetimes for each application. Hence, our simple predictor can effectively locate the best fixed lifetime for each benchmark for these applications.

6.7 Directory Size Scaling

Figures 6.10(a) and 6.10(b) compare the performance and traffic of TCW to GPU-VIni with directories ranging from 8-way associative and 2x the number of entries as total L1 lines (VIni-2x-8w) to 32-ways and 16x the number of L1 lines (VIni-16x-32w). In Figure 6.10(a), directory size and associativity have no impact on performance of GPU applications. On CMP...
architectures, larger directories reduce the frequency of directory misses, which add significant latency to the memory operation. Many CMP coherence designs \cite{34, 50, 78, 92-94} have been proposed to increase the effective directory capacity. For GPUs, a larger directory does not translate to better performance. This is because the latency tolerant GPU architecture is able to tolerate the additional latency introduced by directory misses.

Larger directory size does impact the interconnect traffic usage on GPUs. In Figure 6.10(b), a larger directory reduces the coherence traffic overheads for applications with only intra-workgroup communication. A larger directory requires fewer recalls. However, even a very large and highly associative (16x-32way) directory is unable to completely eliminate the coherence traffic overheads for this set of applications. Figure 6.10(c) shows the breakdown in RG’s traffic for these directory configurations. As the directory size is increased from 2x to 16x, the reduction in recall traffic is offset by the increase in invalidation traffic. As the directory size becomes large enough to capture entire working sets, subsequent kernels need invalidations when accessing the working set from a previous kernel. Hence, while larger directories may reduce recall traffic, the coherence traffic cost of true communication cannot be eliminated. TCW is able to eliminate both sources of coherence traffic overheads by using synchronized time to facilitate communication.
6.8. TC-Weak Predictor Parameters

Figure 6.11: TC-Weak predictor performance with varying parameters.

(a) Vary $t_{\text{write}}$. Fix $t_{\text{evict}} = 8$ and $t_{\text{hit}} = 4$.

(b) Vary $t_{\text{evict}}$. Fix $t_{\text{write}} = 8$ and $t_{\text{hit}} = 4$.

(c) Vary $t_{\text{hit}}$. Fix $t_{\text{write}} = 8$ and $t_{\text{evict}} = 8$. 

Figure 6.11: TC-Weak predictor performance with varying parameters.
6.8 TC-Weak Predictor Parameters

Figure 6.11 evaluates the effects of varying the predictor constants. Figure 6.11(a) varies $t_{\text{write}}$, Figure 6.11(b) varies $t_{\text{evict}}$ and Figure 6.11(c) varies $t_{\text{hit}}$. In each figure the performance is normalized to the predictor configuration selected for TC-Weak in prior evaluations ($t_{\text{write}} = 8$, $t_{\text{evict}} = 8$, $t_{\text{hit}} = 4$).

In Figure 6.11(a), the performance of applications with intra-workgroup communication does not change by varying $t_{\text{write}}$ because it is disabled for applications that do not have memory fences. CC’s performance benefits with higher $t_{\text{write}}$ and $t_{\text{evict}}$. These parameters decrease lifetimes to prevent excessive stalling at the core due to memory fences and at the L2 due to MSHRs occupied by evicted timestamps, respectively. In Figure 6.11(c), CC benefits further from lower lifetimes that result from a lower $t_{\text{hit}}$. Conversely, STN and VPR benefit from higher lifetime predictions from a higher $t_{\text{hit}}$. Performance is most sensitive to variations in $t_{\text{hit}}$ because the ‘load to expired line’ event is more frequent than the ‘store to unexpired lines’ and ‘eviction of unexpired L2 lines’. As can be seen in Figures 6.11(a), (b) and (c), the predictor parameters yielding highest harmonic mean performance were chosen for TC-Weak. A more flexible predictor can benefit performance of applications like CC and VPR.

6.9 Summary of Results

This chapter presented detailed evaluations of various hardware coherence protocols on a GPU. It showed that hardware coherence enabled significant performance improvements by allowing GPU applications that contain inter-workgroup communication to use L1 caching. The traditional coherence protocols MESI and GPU-VI introduce a 127% and 31% traffic overhead, respectively, for a set of applications that do not require coherence. We showed that the majority of MESI’s large traffic overhead resulted from its write-allocate nature, while GPU-VI’s traffic overhead was due to excessive recalls. We showed that a large directory does not address this challenge of GPU coherence. TC-Weak eliminates the overhead of coherence traffic. This results in interconnect energy savings of 36% and 13% over MESI and GPU-VI. This chapter also showed that relaxing write-atomicity in TC-Weak can benefit performance and traffic over TC-Strong independently of other features such as the lifetime predictor.
Chapter 7

Correctness

In this chapter, we show that TC-Strong and TC-Weak implement the required invariants needed by the respective consistency models. Namely, TC-Strong implements the variant of RC with write-atomicity, while TC-Weak implements the variant of RC without write-atomicity.

7.1 TC-Strong Correctness

In this section we show that our TC-Strong based GPU implements a Relaxed Consistency model with atomic stores [85], similarly to other existing consistency models [7, 35, 84, 86]. We first show that TC-Strong satisfies the Single-Writer Multiple-Reader (SWMR) and Data-Value cache coherence invariants [85, Chapter 2] described in Section 2.2. Then we show that our system enforces the ordering rules required for relaxed consistency execution [85, Chapter 5].

7.1.1 TC-Strong Coherence Invariants

Temporal Coherence uses global timestamps to ensure safety without requiring invalidation messages. TC-Strong enforces the following rules. TIME refers to the current system or wall-clock time. GT refers to the global timestamp of a given line stored in the L2 cache. LT_i refers to the local timestamp of a given line cached in the L1 cache at i^{th} GPU core.

Rule #1 - Independent Time. Each component has instantaneous access to TIME. The timestamp counters shown in Figure 4.2(a) provide each component with the system time.

Rule #2 - Local Timestamp. At all times and for all lines in the L2, GT >= LT_i for every i^{th} L1 that caches the line. This is guaranteed because the L1 only updates LT_i with a timestamp returned by the L2. The L2 only responds with timestamps that are less than or equal to the line’s GT.
Rule #3 - Local Data. An L1 cache will only cache the data and update \( LT_i \) from a load response if the updated \( LT_i \geq TIME \).

Rule #4 - Read Rule. A cache line can only be read at the \( i^{th} \) L1 cache if \( TIME \leq LT_i \) and can only be read at the L2 cache if \( TIME \leq GT \). The L1 cache treats expired lines as invalid. The L2 always extends GT for an expired line past \( TIME \) before performing the read.

Rule #5 - Write Rule. A store completes at the L2 only if \( TIME > GT \) for the line being accessed. The L2 stalls a store until \( TIME \) is greater than \( GT \). \( GT \) for a line will not be modified in that time as no other requests to that cache line can be processed during a stall at the L2.

Rule #1 provides the guarantee that all components operate synchronously. The above rules ensure the following properties will hold during system execution.

Property #1. A load and a store to the same line cannot have the same \( TIME \) of completion. Rules #4 and #5 ensure that a load and store occurring at the L2 cache do not occur at the same \( TIME \) because of the conditions that \( TIME \leq GT \) for a load and \( TIME > GT \) for a store. Adding Rule #2 also enforces this property at the L1s as loads at the L1s also obey the \( TIME \leq GT \) condition.

Property #2. No L1 cache will be holding a line at the time a store to that line completes. Rules #2 and #5 together imply that when a store to a line occurs at the L2, \( TIME > LT_i \) at all L1s that cache the line. Therefore, the line is effectively invalid in all L1 caches.

Property #3. All unexpired L1 lines, i.e., lines for which \( TIME \leq LT_i \), will have seen all stores that completed before the current \( TIME \). Consider the latest load response for a line that updated an L1 cache. The load response carries with it data for all stores to the line up to the time when it left the L2 cache. Rules #3 and #5 imply that no store could have occurred to that line between the time that the load response left the L2 and arrived at the L1. As this was the latest load response for this line, and the \( LT_i \leq GT \) condition currently holds for the line, according to Rule #5 no store could have occurred between when the load response arrived at the L1 cache and now. Therefore, no stores have occurred since the latest load response left the L2 and so the L1 holds the value from the all stores to this line that have completed so far.

The three properties stated above enforce the SWMR and Data-Value coherence invariants. Property #1 ensures that a load and store cannot be overlapped in time. Along with the condition that L2 processes one access at a time, Property #1 ensures the SWMR invariant holds. Property #2 and #3 ensure the Data-Value invariant which specifies that each load must
return the value corresponding to the latest store in global memory order. Property #2 ensures that all stale data is invalidated, while Property #3 guarantees that any read that hits at the L1 cache will see all stores to the line that completed prior to the current TIME. These coherence invariants imply a total memory order and that stores occur atomically.

### 7.1.2 Relaxed Consistency Implementation

Section 7.1.1 showed that TC-Strong implements a total memory order. The following ordering constraints [85, Chapter 5] are required to ensure that the total order corresponds to a valid memory order as required by Relaxed Consistency.

1. **Any two memory accesses from a single wavefront separated by a FENCE obey program order in the total memory order.** The in-order GPU cores enforce this ordering by stalling a FENCE instruction until all previous load and store operations from the wavefront have completed. Atomic stores in TC-Strong ensure that stores complete before an acknowledgment is returned to the GPU core.

2. **Any two memory accesses from a single wavefront to the same address obey program order in the total memory order.** The memory stage in the in-order GPU cores does not reorder memory requests, so they are inserted in program order to the memory system. The point-to-point ordering at the interconnect and all cache controllers ensures that requests to the same address from the same wavefront are processed in-order.

3. **A load reads the value from the latest store to the same address.** The SWMR and Data-Value invariants enforced by TC-Strong provide this guarantee.

### 7.2 TC-Weak Correctness

In this section we show that a TC-Weak based GPU architecture implements a Relaxed Consistency model without write atomicity, similarly to existing consistency models [2, 35, 80]. TC-Weak allows RCpc [35] executions, which is the RC model without write atomicity. TC-Weak unifies the ACQUIRE and RELEASE fences of RCpc into a stronger FENCE operation that orders all requests, similarly to the FENCE in TC-Strong. Informally, correct RCpc execution requires that FENCES stall until all memory accesses preceding in program order have performed. As defined by Dubois et al. [32], a load is considered performed at a point in time when any store issued by any thread...
7.2. TC-Weak Correctness

(wavefront for a GPU) after this time cannot affect the value returned by the load. A store is considered performed at a point in time after which no thread (wavefront for a GPU) can read the stale value that was overwritten by this store. As per RCpc, the following conditions enforce correctness. The definitions of TIME, GT and LT remain unchanged from Section 7.1.1. Additionally, we define GWCT as the Global Write Completion Time, the timestamp returned by the L2 informing a GPU core of the time by when this store will be performed.

1. All previous load and store accesses from a wavefront must be performed before a later FENCE can complete.

This guarantee is trivially enforced for load accesses as a FENCE operation stalls until all pending loads have completed. It is non-trivial for stores because in TC-Weak an acknowledgment from the L2 that the store request was completed does not imply that a store has been performed and is visible to all wavefronts. This is because TC-Weak breaks Rule #5 from Section 7.1.1 and allows a store request to update the L2 cache even if \( \text{TIME} < \text{GT} \). Rules #1 - #4, however, still apply to TC-Weak. To show correctness for this case, we use the following properties of TC-Weak.

**Property #1’.** Given a GWCT returned by a store and the GT of a line when the store updated the L2 cache, \( \text{GWCT} > \text{GT} \). All stores occur at the L2 and the write completion time returned is one plus \( \text{GT} \) of the line.

**Property #2’.** At all times, any line in the \( i^{th} \) L1 cache with \( \text{TIME} \leq \text{LT} \) has seen all the stores that performed up to the time that the latest response (that updated the local timestamp) for this line from the L2 left the L2 cache. All load responses from the L2 contain data, and hence they contain all stores to that address up to the time the response leaves the L2. The load responses update both the data and the local timestamps at the L1 as per Rule #3.

Using the above two properties we can show that each store is performed by its GWCT. Assume that a store is processed at the L2 at \( \text{TIME} = 5 \) and a \( \text{GWCT} = 10 \) is returned. Further assume that a load hit at an L1 occurs at \( \text{TIME} = 10 \) and the previously mentioned store was not seen by this load (implying incorrect behaviour). From Property #2’, this implies that no load response (that updated this L1’s line) left the L2 between \( \text{TIME} = 6 \) and \( \text{TIME} = 10 \), inclusive. This further implies that at \( \text{TIME} = 5 \) the L1 line had a timestamp of \( \text{LT} \geq 10 \) because an L1 hit occurred at \( \text{TIME} = 10 \) and this line was not modified after \( \text{TIME} = 5 \). From Property #1’ we know that at \( \text{TIME} = 5 \) the timestamp at the L2 for the line was \( \text{GT} \leq 9 \) because a \( \text{GWCT} = 10 \) was returned by the store. This is a contradiction with Rule #2 in Section 7.1.1 which states that \( \text{GT} \geq \text{LT} \).
at all times. Therefore, by $TIME = 10$ the load hit at the L1 must have seen the store, and hence the store was performed by its $GWCT = 10$.

TC-Weak stores in each GPU core a per-wavefront $GWCT$ entry containing the maximum of the $GWCT$ returned by all previous stores from the wavefront. A TC-Weak FENCE stalls until all pending stores have updated the wavefront’s $GWCT$ entry and until $TIME \geq GWCT$ entry. This guarantees that all stores prior to this FENCE have performed.

2. All prior FENCE operations should be completed before any later loads or stores can be performed.

The in-order GPU core does not schedule any instructions from a wavefront whose FENCE operation has not yet completed.

3. Synchronization accesses are write atomic and sequentially consistent with respect to other synchronization accesses.

By default, the read-modify-write, or atomic, operations on a GPU used for synchronization purposes are not sequentially consistent. Sequentially consistent atomic operations can be provided by simply enclosing an atomic access between two FENCE instructions.

The preceding three conditions show that TC-Weak supports RCpc executions. Supporting RCpc executions means that TC-Weak supports SC execution of Data Race Free (SC for DRF) programs [5].

### 7.3 Towards Full Verification

This chapter presented the TC-Strong and TC-Weak as a set of rules. It then showed that these sets of rules are equivalent to the invariants required by specific implementations of coherence or consistency models. It also briefly discussed how our protocol implements each rule. The discussion here does not imply that the protocols we present in this thesis are fully verified. Future work towards verifying the protocols needs to show that the specific implementations correctly implement the rules this chapter presented. The detailed state machines for both TC-Strong and TC-Weak that we presented in this thesis should aid this final step in full protocol verification.
Chapter 8

Related Work

This chapter discusses research related to this work.

8.1 Timestamp-Based Software Coherence

The use of timestamps has been explored in software coherence. Cheong and Viedenbaum [22] proposed a version control based software coherence scheme that keeps a version number for each variable in a table. At the beginning of a phase, it stores the version number in an extra field in the cache line. At the end of the phase, it increments the software version number for all variables that may have been modified in that phase. In the next phase a cache line is only considered valid if the variable’s software version is less than the cache line’s version, otherwise the data may be stale. This scheme introduces a large storage overhead to track versions for each variable in software, and a performance overhead due comparing version numbers at every load.

Min et al. [66] extend this technique with more sophisticated compiler analysis to determine which variable may be modified in a phase. Chiueh [23] reduced the storage overhead of tracking a version or timestamp in software for each variable by using one version number for all variables. However, the runtime overhead of comparing and maintaining the cache line’s version number remains. The TS1 [31] software coherence scheme eliminates the software overhead of tracking version or timestamps. It adds one bit to each cache line that is set when a variable is referenced in the current phase. At the end of each phase it invalidates all cache lines where the bit is not set and that may have been modified by other processors. It however has a run-time overhead of invalidating lines at the end of each phase. TPI [25] uses the phase id instead of version numbers as the timestamp. TBSIS [91] combines TS1 and TPI to reduce both runtime overhead of invalidations and the storage costs of timestamps. TBSIS however shifts the overhead to hardware by requiring an instruction that can quickly invalidate all lines where the timestamp equals a given value. All of these schemes use compiler-directed analysis and add some form of run-time overhead for software coherence.
They are also designed for task-based parallel programming models that are even more restrictive than current GPU programming models. TC-Weak does not require any software or compiler support.

8.2 Timestamp-Based Hardware Coherence

Nandy et al. [67] first considered timestamps for hardware coherence. LCC [57, 82] is a time-based hardware coherence proposal that stores timestamps in a directory structure and delays stores to unexpired lines to enforce Sequential Consistency on CMPs. The TC-Strong implementation of the TC framework is similar to LCC as both enforce write atomicity by stalling stores at the shared last level cache. Unlike LCC, TC-Strong supports multiple outstanding stores from a core and implements a relaxed consistency model. TC-Strong also includes optimizations to eliminate stalls due to private writes and L2 evictions. Despite these changes, we find that the stalling of stores in TC-Strong causes poor performance on a GPU. We propose TC-Weak and a novel time-based memory fence mechanism to eliminate all write-stalling, improve performance, and reduce interconnect traffic compared to TC-Strong. We also show that unlike for CMP applications [57, 82], the fixed timestamp prediction proposed by LCC is not suited for GPU applications. We propose a simple yet effective lifetime predictor that can accommodate a range of GPU applications. Lastly, we present a full description of our proposed protocol, including state transition tables that describe the implementation in detail.

8.3 Self-Invalidation Based Coherence

Self invalidation of lines in a private cache has also been previously explored in the context of cache coherence. Dynamic Self-Invalidation (DSI) [56] reduces critical path latency due to invalidation by speculatively self-invalidating lines in private caches before the next exclusive request for the line is received. In the sequentially consistent implementation, DSI requires explicit messages to the directory at self-invalidation and would not alleviate the traffic problem on a GPU. In its relaxed consistency implementation, DSI can reduce traffic through the use of tear-off lines, which are self-invalidated at synchronization points. Recently, Ros et al. [75] proposed extending tear-off lines to all cache lines to eliminate coherence directories entirely, reducing implementation complexity and traffic for CMP coherence. Their protocol requires self-invalidation of all shared data at synchroniza-
8.4 Scalable Coherence

Synchronization events, however, are much more frequent on a GPU. Thousands of scalar threads share a single L1 cache and would cause frequent self-invalidations. Their protocol also requires blocking and buffering atomic operations at the last level cache. GPUs support thousands of concurrent atomic operations; buffering these would be very expensive. A Last Touch Predictor [53] allows selective invalidation of cache lines, but would only move the coherence traffic from invalidation to recall. TC-Weak eliminates all invalidation and recall messages through shared timestamps.

8.4 Scalable Coherence

Denovo [24] simplifies the coherence directory but uses a restrictive programming model that requires user annotated code. TC-Weak does not force restrictions onto the GPU programming model. Kelm et al. [49] discovered that some applications contain data regions that benefit from hardware coherence and other regions that benefit with coherence disabled. They propose a hybrid coherence system that requires code modifications and additional hardware to disable coherence at a fine granularity for such applications. In contrast, TC-Weak effectively disables coherence for data regions that do not require coherence but does not require any code modifications. Recent coherence proposals [50, 78, 94] simplify tracking sharer state for 1000s of cores. GPUs have tens of cores; exact sharer representation is not an issue.

ARM recently announced a cache coherent architecture for their Mali GPUs [3]. Their architecture uses a MESI directory protocol to provide coherence within the GPU. AMD’s roadmap [10] for their fused CPU-GPU architectures includes fully coherent GPUs. Intel’s manycore MIC architecture [26] already includes coherent caches.

8.5 Consistency Models for Throughput-Oriented Processors

Recently, Hechtman and Sorin [39] explored different consistently models on a GPU-like massively multithreaded architecture. Their findings suggest that strong consistency models like Sequential Consistency may be well suited for these architectures because these architectures implement simpler pipelines. Their results also support our thesis that optimizing for store latency is far less useful for GPU applications than it is for CPU applications.
Chapter 9

Conclusion and Future Work

This chapter provides a concluding summary of this thesis and presents future work.

9.1 Conclusions

In this thesis, we motivated hardware cache coherence for GPUs and explored its implications. We showed that coherence can improve GPU programming and improve performance of GPU applications by enabling the use of private caches. Our evaluations found that some assumptions of CMP coherence do not apply to GPUs. On GPUs, write-through protocols are preferable to writeback protocols. Larger directories are not as beneficial to GPU performance as they are to CMPs.

We found that conventional coherence implementations introduce a new set of challenges when applied to GPU architectures. The management of transient state for thousands of in-flight memory accesses adds hardware and complexity overhead. Coherence adds unnecessary traffic overheads to existing GPU applications. Accelerating applications with both coherent and non-coherent data requires that the latter introduce minimal coherence overheads. We presented Temporal Coherence, a timestamp based coherence framework that addresses the overheads of GPU coherence. We proposed an implementation of this framework, TC-Weak, which uses novel timestamp based memory fences to reduce these overheads.

Our evaluations showed that TC-Weak can successfully address the new concerns that arise with hardware coherence for highly parallel architectures. It reduces the traffic of the conventional coherence protocols MESI, GPU-VI and GPU-VIni by 56%, 23% and 22% across a set of applications without coherent data. This leads to lower energy overheads in introducing coherence. It provides a 85% speedup over disabling the non-coherent L1’s for a set of applications that require coherent caches. This makes hardware coherence an attractive feature that allows a new class of applications to benefit from GPU hardware. Lastly, TC-Weak provides a 28% performance improvement over TC-Strong. This shows that relaxing some coherence constraints, such
as write atomicity, can be beneficial over considering coherence as a fixed black-box system.

There has been a recent push towards properly defining consistency models for CMPs. The C++ \[15\] and Java \[58\] memory models have gone through revisions to more precisely define their requirements from these models. Hardware manufacturers are urged \[81, 85\] to more properly define the behaviour of their memory systems. GPU hardware, however, has moved slowly in this aspect by failing to reveal many details of its memory system. We encourage GPU manufacturers to support well-defined consistency models and the SC for DRF paradigm so that GPU programmers too can utilize the benefits of high-level languages.

9.2 Future Work

9.2.1 Lifetime Prediction

In this thesis we proposed a simple lifetime prediction mechanism. More advanced prediction mechanism may further improve the performance of TC protocols. For example, lifetime may be correlated with instructions and a Program Counter based prediction might yield better results. Studying the relationship between code and lifetimes may yield hints on possibly making more informed predictions through compiler analysis.

9.2.2 Graphics Applications

While we studied the effects of coherence on GPU applications that don’t need coherence, we did not explore its impacts on graphics applications. Since graphics are still the major driving force for advancements in GPU architectures, it is important that coherence does not negatively impact these applications. With these applications, the issue of how coherence should handle graphics pipeline specific features, such as texture caches and Z-Buffer Units \[45\], arises and needs to be explored.

9.2.3 Cache Management

Recent research \[74\] shows that thread scheduling and cache management policies can greatly impact the performance of GPU applications. These have direct impact on the coherence protocol, and especially the lifetime of cache lines for TC protocols. Future work should explore GPU cache management and GPU coherence in conjunction.
9.2.4 CPU-GPU Coherence

This thesis showed that the requirements for CPU coherence and GPU coherence are fundamentally different. Fused CPU-GPU architectures \[16, 48\] may need to separately address these challenges. One approach may be to use a framework \[13\] that allows combining of different coherence protocols through a single interface. Future work in this domain also includes studying the impact CPU-GPU coherence has on shared resources, like the interconnection network.
Bibliography


Bibliography


