# Efficient Synchronization Mechanisms for Scalable GPU Architectures

by

Xiaowei Ren

M.Sc., Xi'an Jiaotong University, 2015

B.Sc., Xi'an Jiaotong University, 2012

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

**Doctor of Philosophy**

in

THE FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES
(Electrical and Computer Engineering)

The University of British Columbia
(Vancouver)

October 2020

The following individuals certify that they have read, and recommend to the Faculty of Graduate and Postdoctoral Studies for acceptance, the dissertation entitled:

**Efficient Synchronization Mechanisms for Scalable GPU Architectures**

submitted by **Xiaowei Ren** in partial fulfillment of the requirements for the degree of **Doctor of Philosophy** in **Electrical and Computer Engineering**.

**Examining Committee:**

Mieszko Lis, Electrical and Computer Engineering
*Supervisor*

Steve Wilton, Electrical and Computer Engineering
*Supervisory Committee Member*

Konrad Walus, Electrical and Computer Engineering
*University Examiner*

Ivan Beschastnikh, Computer Science
*University Examiner*

Vijay Nagarajan, School of Informatics, University of Edinburgh
*External Examiner*

**Additional Supervisory Committee Members:**

Tor Aamodt, Electrical and Computer Engineering
*Supervisory Committee Member*

# Abstract

The Graphics Processing Unit (GPU) has become a mainstream computing platform for a wide range of applications. Unlike latency-critical Central Processing Units (CPUs), throughput-oriented GPUs provide high performance by exploiting massive application parallelism.

In parallel programming, synchronization is necessary to exchange information for inter-thread dependency. However, inefficient synchronization support can serialize thread execution and restrict parallelism significantly. Considering parallelism is key to GPU performance, we aim to provide efficient and reliable synchronization support for both single-GPU and multi-GPU systems. To achieve this target, this dissertation explores multiple abstraction layers of computer systems, including programming models, memory consistency models, cache coherence protocols, and application specific knowledges of graphics rendering.

First, to reduce programming burden without introducing data-races, we propose Relativistic Cache Coherence (RCC) to enforce Sequential Consistency (SC). By avoiding stalls of write permission acquisition with logical timestamps, RCC is 30% faster than the best prior SC proposal, and only 7% slower than the best non-SC design. Second, we introduce GETM, the first GPU Hardware Transactional Memory (HTM) with eager conflict detection, to help programmers implement deadlock-free, yet aggressively parallel code. Compared to the best prior GPU HTM, GETM is up to 2.1× (1.2× gmean) faster, area overheads are 3.6× lower, and power overheads are 2.2× lower. Third, we design HMG, a hierarchical cache coherence protocol for multi-GPU systems. By leveraging the latest scoped memory model, HMG not only can avoid full cache invalidation of software coherence protocol, but also filters out write invalidation acknowledgments and transient

coherence states. Despite minimal hardware overhead, HMG can achieve 97% of the performance of an idealized caching system. Finally, we propose CHOPIN, a novel Split Frame Rendering (SFR) scheme by taking advantage of the parallelism of image composition. CHOPIN can eliminate the performance overheads of primitive duplication and sequential primitive distribution that exist in previous work. CHOPIN outperforms the best prior SFR implementation by up to 56% (25% gmean) in an 8-GPU system.

# Lay Summary

This dissertation proposes architectural supports for efficient synchronizations in both single-GPU and multi-GPU systems. The innovations span across multiple abstraction layers of the computing system, including the programming model, memory consistency model, cache coherence protocol, and application specific knowledge of graphics processing. This can simplify GPU programming, increase performance, and extend hardware scalability to large-scale systems, thereby attracting more programmers and extending GPU to a wider range of application domains.

# Preface

The following is a list of my publications during the PhD program in chronological order:

[C1] Xiaowei Ren, and Mieszko Lis. "Efficient Sequential Consistency in GPUs via Relativistic Cache Coherence". In Proceedings of the 23rd International Symposium on High Performance Computer Architecture (HPCA), pages 625–636. IEEE, 2017.

[C2] Xiaowei Ren, and Mieszko Lis. "High-Performance GPU Transactional Memory via Eager Conflict Detection". In Proceedings of the 24th International Symposium on High Performance Computer Architecture (HPCA), pages 235–246. IEEE, 2018.

[C3] Xiaowei Ren, Daniel Lustig, Evgeny Bolotin, Aamer Jaleel, Oreste Villa, and David Nellans. "HMG: Extending Cache Coherence Protocols Across Modern Hierarchical Multi-GPU Systems". In Proceedings of the 26th International Symposium on High Performance Computer Architecture (HPCA), pages 582–595. IEEE, 2020.

[C4] Xiaowei Ren, and Mieszko Lis. "CHOPIN: Scalable Graphics Rendering in Multi-GPU Systems via Parallel Image Composition". (Under Submission)

The publications are incorporated into this dissertation as follows:

- Chapter 2 uses background section materials from [C1], [C2], [C3], and [C4].
- Chapter 3 presents a version of the material published in [C1]. In this work, Xiaowei Ren was the leading researcher, he designed the Relativistic Cache Coherence (RCC), implemented and evaluated RCC in simulation framework, analyzed simulation results, and contributed to the article writing. This work

was done under the supervision of Professor Mieszko Lis, he finished most of the article writing and provided lost of helpful guidance for this work.

- Chapter 4 presents a version of the material published in [C2]. In this work, Xiaowei Ren was the leading researcher, he proposed the eager conflict detection mechanism and the metadata tracking hardware structure for GETM, implemented and evaluated GETM in simulation framework, analyzed simulation results, and contributed to the article writing. This work was done under the supervision of Professor Mieszko Lis, he finished most of the article writing and provided lots of helpful guidance for this work.

- Chapter 5 presents a version of the material published in [C3]. In this work, Xiaowei Ren was the leading researcher, he extended the cache coherence protocol to hierarchical multi-GPU systems, deeply optimized the protocol by leveraging the characteristics of scoped memory consistency model, implemented and evaluated the proposal in simulation framework, analyzed simulation results, and finished article writing. This work was done under the mentoring of Daniel Lustig at NVIDIA, he provided lots of helpful guidance for this work. Other coauthors also offered many helpful comments to this work.

- Chapter 6 presents a version of the material described in [C4]. In this work, Xiaowei Ren was the leading researcher, he proposed the scalable Split Frame Rendering (SFR) scheme by leveraging parallel image composition, optimized the proposal by designing a draw command scheduler and an image composition scheduler, implemented and evaluated the proposal in simulation framework, analyzed simulation results, and finished article writing. This work was done under the supervision of Professor Mieszko Lis, he provided lots of helpful guidance for this work.

- Chapter 7 uses the related work sections in [C1], [C2], [C3], and [C4].

- Chapter 8 uses the conclusion text from [C1], [C2], [C3], and [C4].

# Table of Contents

# List of Tables

# List of Figures

# List of Abbreviations

| | |
|---|---|
| **AFR** | Alternate Frame Rendering |
| **AI** | Artificial Intelligence |
| **API** | Application Programming Interface |
| **AR** | Augmented Reality |
| **CMP** | Chip Multiprocessor |
| **CPU** | Central Processing Unit |
| **CTA** | Thread Block Array |
| **DLP** | Data Level Parallelism |
| **DLSS** | Deep Learning Super Sampling |
| **DRF** | Data-Race-Free |
| **FB** | Framebuffer |
| **GPC** | Graphics Processing Cluster |
| **GPM** | GPU Module |
| **GPU** | Graphics Processing Unit |
| **HRF** | Heterogeneous-Race-Free |
| **HTM** | Hardware Transactional Memory |
| **ILP** | Instruction Level Parallelism |
| **IMR** | Immediate Mode Rendering |
| **LLC** | Last Level Cache |

**LRU** Least Recently Used

**MCM** Multi-Chip-Module

**MIMD** Multiple-Instruction, Multiple-Data

**MLP** Memory Level Parallelism

**NUMA** Non-Uniform Memory Access

**PME** PolyMorph Engine

**PSO** Partial Store Ordering

**RC** Release Consistency

**RCC** Relativistic Cache Coherence

**ROP** Rendering Output Unit

**RT** Render Target

**SC** Sequential Consistency

**SIMD** Single-Instruction, Multiple-Data

**SIMT** Single-Instruction, Multiple-Thread

**SM** Streaming Multiprocessor

**SFR** Split Frame Rendering

**SWMR** Single Writer Multiple Reader

**TLP** Thread Level Parallelism

**TM** Transactional Memory

**TPC** Texture Processing Cluster

**TSO** Total Store Ordering

**VR** Virtual Reality

**VU** Validation Unit

**WO** Weak Ordering

# Acknowledgments

It has been a long journey since I started my PhD study. Lots of great people have made this journey unforgettable and worth taking. Without their help, the work in this dissertation would have not been possible.

First and foremost, I would like to thank my dear supervisor, Professor Mieszko Lis. It has been my great honour of working with him throughout my PhD program. His dedication to the research and kindness to the students have truly inspired me both professionally and personally. He was always there for me whenever I needed advice, whether it be technical, professional, personal, or otherwise. He also encouraged me to explore various research topics, which have enormously enriched my knowledge beyond this dissertation. He accepted nothing less than my best, and I will be forever grateful for his mentorship.

I also would like to thank my qualifying, department, and university examination committee members: Professor Tor Aamodt, Professor Steve Wilton, Professor Sudip Shekhar, Professor Guy Lemieux, Professor Shahriar Mirabbasi, Professor Konrad Walus, Professor Ivan Beschastnikh, Professor Ryozo Nagamune, and Professor Vijay Nagarajan. I am grateful for their insightful feedback, which has immensely improved my research work.

I also would like to thank all the people who helped me during my internships in NVIDIA and MPI-SWS, especially Daniel Lustig, David Nellans, Evgeny Bolotin, Aamer Jaleel, Oreste Villa, Viktor Vafeiadis, and Michalis Kokologiannakis. Daniel Lustig was a great mentor during my two internships in NVIDIA, I really appreciate his technical and personal support during and after my time there. Viktor Vafeiadis was an excellent mentor while I was at MPI-SWS, I am extremely thankful that he offered me a precious opportunity to explore a totally new research topic.

I also would like to thank all my lab colleagues: Amin Ghasemazar, Ellis Su, Mohammad Ewais, Maximilian Golub, Khaled E. Ahmed, Mohamed Omran Matar, Dingqing Yang, John Deppe, Peter Deutsch, Mohammad Olyaiy, Christopher Ng, Muchen He, Winnie Gong, and Avilash Mukherjee. I have learned a lot from all of them. Special thanks to Ellis Su, Dingqing Yang, Amin Ghasemazar, and John Deppe, it was my great pleasure to cooperate with them.

Finally, I would like to extend my huge thank you to my parents and grandparents. They have fully supported my decision to pursue my PhD degree. Even though they did not understand my research, they firmly believed me and offered me their heartwarming encouragements. I am regretful that I could not go home and visit them very often during the past several years. I wish they stay happy and healthy forever.

*To my parents.*

# Chapter 1

# Introduction

During past few decades, semiconductor technology has largely benefited from Moore's Law scaling, which has enabled an exponentially increasing number of transistors in a single chip. With abundant transistors, processor designers have greatly improved single-thread performance by maximizing the Instruction Level Parallelism (ILP). Multi-core architecture, such as Chip Multiprocessor (CMP), has also been designed to exploit the Data Level Parallelism (DLP) and Thread Level Parallelism (TLP). However, the failure of Dennard Scaling has made these architectures hit the power wall, and the problem of dark silicon may significantly limit their scalability [73]. What makes the situation even worse is the fact that the Moore's Law is also approaching the end [210, 232]. In response to these observations, much attention has been focused on more cost-efficient alternatives. The massively parallel GPU architecture has been proven to be a promising candidate.

CMP is a widely adopted multi-core CPU architecture. Individual cores are optimized to reduce single-thread latency, but computing throughput is usually limited by the relatively small number of processor cores. In contrast, GPU tradeoffs single-thread latency for system throughput, a massive number of threads are run on simple shader cores in parallel. The cost of instruction fetching, decoding, and scheduling is amortized by executing threads in Single-Instruction, Multiple-Data (SIMD) fashion.

Thanks to the abundant parallelism, GPU architecture is much more cost-efficient for applications that perform identical operations on enormous amounts of data (i.e.,

regular parallelism). A typical application is graphics processing, the application that GPU was originally built for. In recent years, with enhancements in both software and hardware, academia researchers and industry vendors have successfully extended GPUs into a much wider range of application domains, such as graph algorithms [91], scientific computing [101], machine learning [226], and so on. GPU has become one of the major platforms in computing society.

In parallel architectures, efficient synchronization mechanisms are critical to guarantee high performance. Otherwise, synchronization can significantly restrict available parallelism and reduce performance. The massive number of concurrent threads in GPUs makes this problem far more prominent than CPUs. Given the importance of GPU computing, this dissertation explores architectural support for robust and efficient synchronization in GPUs.

## 1.1 The Extensive Usage of the GPU Platform

Unlike CMP, GPU expects applications to expose a large amount of parallelism. For example, 3D graphics rendering contains plenty of data-level parallelism; all attributes of primitives and fragments are computed independently. The code that specifies the detailed operations is called a shader, which is programmed with graphics Application Programming Interface (API), such DirectX [7] and Vulkan [10]. Conventional GPUs built fixed pipelines for different shaders. Although this can reduce the hardware design complexity, the flexibility and utilization of hardware components are sacrificed. To address this problem, NVIDIA's Tesla [132] and AMD's TeraScale [142] architectures replaced the fixed pipeline with the unified shader model, which has largely enriched the programmability of GPU hardware.

The increased programmability also created opportunities for other non-graphics applications to take advantage of the computing capacity of GPUs. Therefore, in addition to graphics APIs, industry has also developed CUDA [177] and OpenCL [112] for general-purpose applications. With the introduction of CUDA and OpenCL, the resultant programming model is called Single-Instruction, Multiple-Thread (SIMT). SIMT can efficiently hide the complexity of SIMD hardware, because it allows programmers to think about their code in the same way as single-thread execution. SIMT has seen widespread interest and greatly extended the usage of GPU platform.

So far, the high-level domain specific libraries built in CUDA have existed in a broad range of applications, including ray tracing, medical imaging, machine learning, autonomous driving, robotics, smart cities, and so on [100].

In addition to graphics, machine learning has become another critical GPU application in recent years. Hence, NVIDIA integrated Tensor Cores [163] in their GPUs to accelerate matrix convolution, a common operation of neural networks. They also designed the RTX platform [161] for ray tracing acceleration. Deep Learning Super Sampling (DLSS) is one of the latest graphics technologies enabled by Tensors Cores [178]. With DLSS, NVIDIA set out to redefine real-time rendering through AI-based super resolution – rendering few pixels and then using AI to construct sharp, higher resolution images. Together with RTX platform, DLSS gives gamers the performance headroom to maximize ray tracing settings and increase output resolutions.

Some recent applications require computing power beyond single-GPU system, so multi-GPU systems have been developed to further scaling performance [170, 173, 174]. In multi-GPU systems, individual GPUs are connected with the advanced networking technologies, such as NVLink [160] and NVSwitch [162]. GPUs have also been deployed in datacenters or cloud systems to accelerate supercomputing applications, such as scientific computing [101], genome sequencing [159], weather forecasting [175] etc. Cloud gaming systems, like GeForce Now, have also been built to provide game players a high-quality experience without a substantial hardware investment [158].

## 1.2 Challenges of GPU Synchronization

Along with the diversification of GPU applications, data sharing and synchronization patterns have become more and more complex. Therefore, GPUs have shifted away from a simple bulk-synchronous model to a more traditional shared memory programming model. Industrial vendors have exposed an abstraction of Unified Memory (i.e, unified virtual address space) to software programmers [112, 168]. The NVIDIA Volta GPU pushed this abstraction even further by enabling independent thread scheduling – each thread can execute independently with explicit forward progress guarantee [69]. This modification allows general-purpose Multiple-Instruction,

Multiple-Data (MIMD) execution in SIMD architecture of GPUs [72]. Recently, scoped memory models have also been formalized to support flexible fine-grained synchronizations [98, 135]. Although lots of innovations have been proposed, many challenges still exist in the synchronization of GPUs as follows.

- GPU memory models allow many weak behaviours, so programmers need to insert fences to enforce necessary memory order. However, correctly inserting fences is difficult and bug-prone; the authors of [16] found missing fences in a variety of peer-reviewed publications, and even vendor guides [204]. In contrast, Sequential Consistency (SC) is the most intuitive memory model for programmers. However, constrained by strong order requirements, the performance of SC enforcement with MESI-like and timestamp-based cache coherence protocols is limited by the stalls for acquiring write permissions [94, 221]. Therefore, it's desirable to propose an efficient cache coherence protocol to enforce SC without write stalls.

- Although the lock-based synchronization has been widely used in CPU systems, the massive number of threads in GPUs makes it much more difficult to get optimal performance and guarantee deadlock-free execution. Transactional Memory (TM) is an alternative solution, which can potentially avoid these problems by relying on the underlying mechanism to detect conflicts (data-races) automatically in deadlock-free manner [96]. Unfortunately, the excessive conflict detection latency leaves the performance of prior GPU TM designs far away behind fine-grained locks [76, 77]. This has significantly demotivated the usage of TM in GPU systems.

- More recently, there are two prominent changes for GPU systems: hierarchical architecture and scoped memory model. To keep scaling performance, GPU vendors have built ever-larger GPU systems by connecting multiple chip modules (MCM-GPU) [29] and GPUs (multi-GPUs) [143, 253]. Due to physical constraints, the bandwidth of the latest inter-GPU link [1, 160, 162] is still one order of magnitude lower than intra-GPU connections [187], resulting in severe Non-Uniform Memory Access (NUMA) effect that can bottleneck performance often. In addition, GPUs have also changed from conventional

bulk-synchronous towards scoped memory models for flexible synchronization support [98, 135]. Caching has been widely implemented to mitigate NUMA effect. However, none of existing cache coherence protocols can support efficient caching in multi-GPU systems, since they do not consider the changes in both computer architecture and memory model.

- Apart from above challenges that are related to the synchronization support for general-purpose applications, it's also critical to improve the synchronization for conventional graphics applications. In principle, the latest multi-GPU systems are promising to substantially improve performance and offer a visual experience with much higher quality. Unfortunately, it is not clear how to adequately parallelize the rendering pipeline to take advantage of these resources while maintaining low rendering latencies. Current implementations of Split Frame Rendering (SFR) are bottlenecked by the redundant computation and the sequential inter-GPU synchronizations [20, 114, 166], so their performance does not scale. To fully release the performance potential of multi-GPU systems, a mechanism which can eliminate these bottlenecks is urgently needed for graphics applications.

## 1.3 Thesis Statement

This dissertation aims to address above challenges. It enhances the ability of providing high-performance synchronizations in both single-GPU and multi-GPU systems. The major proposed enhancements include efficient cache coherence protocols, GPU Hardware Transactional Memory (HTM), and scalable Split Frame Rendering (SFR). By offering robust and efficient synchronization mechanisms, the proposals in this dissertation can potentially attract more programmers and extend GPU to a wider range of application domains.

First, this dissertation proposes Relativistic Cache Coherence (RCC) to enforce SC in logical time. In RCC, all Streaming Multiprocessor (SM) cores and cachelines have separate timestamps. Therefore, SM cores can see different cachelines based on their own timestamps, and SM core timestamps are advanced independently according to different cachelines they accessed. Instead of stalling write requests until all sharers become invalid, RCC chooses to advance the timestamp of writing SM

core instantly. Although instant timestamp advancement can potentially invalidate other cachelines in the same SM core, the memory load latency is well known to be tolerated by GPU architecture. By enforcing SC without write stalls, RCC can avoid the complexity of fence inserting and provide high performance to programmers at the same time.

Second, this dissertation proposes GETM, a novel GPU Hardware Transactional Memory (HTM) system, which can reduce the excessive latency of prior value-based lazy conflict detection [76, 77] with a logical-timestamp-based eager mechanism. GETM detects conflicts by comparing the timestamps of transactions and the data they accessed. If the transaction timestamp is smaller than the data timestamps which have been set by other transactions, it indicates that the current transaction has a conflict with other executed ones. While conflicts are detected, GETM eagerly aborts the current transaction, advances its timestamp, and restarts it later. With eager conflict detection, transactions that have reached commit point are guaranteed to be conflict-free, so their results can be committed to the memory without additional validation. Even though eager conflict detection can slightly increase transaction abort rate, the dramatically faster aborts and commits of individual transactions can transfer to substantial performance improvement.

Third, this dissertation proposes HMG, a hierarchical cache coherence protocol for multi-GPU systems. Similar to GPU-VI [221], HMG is a simple two-state hardware cache coherence protocol, but adds a coherence hierarchy to exploit intra-GPU data locality and reduce the bandwidth overhead on inter-GPU links. HMG has also been deeply optimized by leveraging the non-multi-copy-atomicity of scoped memory models – non-synchronization stores are processed instantly without waiting for invalidation acknowledgments, and only synchronization stores are stalled to enforce correct data visibility. Since there are no stalls for most store requests and GPU architecture is latency-tolerant, HMG can eschew the complexity of transient coherence states and extra hardware complexities that are necessary for latency-critical CPUs. With efficient cache coherence protocol support, applications which have fine-grained synchronizations [91, 118, 258] can be accelerated substantially in multi-GPU systems.

Finally, this dissertation proposes CHOPIN, a scalable SFR technique. Considering various properties of draw commands in a single frame, CHOPIN first divides

them into multiple groups. Each draw command is distributed to a specific GPU, so no redundant computing exists in CHOPIN. At group boundaries, sub-images generated in each GPU are composed in parallel. Sub-images of opaque objects are composed out-of-order by retaining pixels that are closer to the camera. Although sub-image composition of transparent objects needs to respect the depth order, CHOPIN leverages the associativity of pixel blending [33] to maximize the parallelism – neighbouring sub-images start to compose with each other once they are ready. Therefore, CHOPIN can eliminate the sequential inter-GPU communication overhead. A draw command scheduler and an image composition scheduler are designed to address the problems of load-imbalance and network congestion. By leveraging parallel image composition, CHOPIN is more scalable than prior SFR solutions [20, 114, 166].

## 1.4 Contributions

This dissertation makes the following contributions:

1. It traces the cost of Sequential Consistency (SC) enforcement in realistic GPUs to the need to acquire write permissions, proposes Relativistic Cache Coherence (RCC) that improves store performance by enforcing SC in logical time, and demonstrates that RCC is faster than the best prior GPU SC proposal by 29% and within 7% of the performance of the best non-SC design.

2. It traces the inefficiency of prior GPU Hardware Transactional Memory (HTM) proposals to the unamortized latencies of value-based lazy conflict detection, proposes the first GPU HTM called GETM that detects conflicts with a logical-timestamp-based eager mechanism, and demonstrates that GETM is up to 2.1× (1.2× gmean) faster than the best prior proposal.

3. It identifies the necessity of coherence hierarchy for performance scaling in multi-GPU systems, proposes a hierarchical cache coherence protocol called HMG for multi-GPUs, eliminates the complexity of transient coherence states and invalidation acknowledgments by leveraging the non-multi-copy-atomicity of scoped memory models, and demonstrates that HMG can achieve 97% of the performance of an idealized caching system.

4. It traces the main performance cost of existing Split Frame Rendering (SFR) mechanisms to redundant computation and sequential inter-GPU communication requirements, proposes a novel SFR technique called CHOPIN that takes advantage of the parallel image composition to remove overheads of prior solutions, develops a draw command scheduler and an image composition scheduler to address the problems of load-imbalance and network congestion, and demonstrates that CHOPIN outperforms the best prior SFR proposal by up to 56% (25% gmean) in an 8-GPU system.

## 1.5 Organization

The rest of this dissertation is organized as follows:

- Chapter 2 gives the background on GPU architecture, synchronizations, memory consistency model, cache coherence protocol, and graphics processing.
- Chapter 3 proposes Relativistic Cache Coherence (RCC), a logical-timestamp-based cache coherence protocol which can efficiently enforce Sequential Consistency (SC) in GPUs by reducing stalls for write permission acquisition.
- Chapter 4 presents GETM, the first GPU Hardware Transactional Memory (HTM) which reduces excessive latency of conflict detection by eagerly checking conflicts when the initial memory request is made.
- Chapter 5 proposes HMG, a hardware-managed cache coherence protocol designed for forward-looking hierarchical multi-GPU systems with the enforcement of scoped memory consistency model.
- Chapter 6 describes CHOPIN, a novel Split Frame Rendering (SFR) scheme which can eliminate the performance overheads of redundant computing and sequential primitive exchanging that exist in prior solutions by leveraging parallel image composition.
- Chapter 7 discusses related work.
- Chapter 8 concludes the dissertation and discusses directions for potential future work.

# Chapter 2

# Background

This chapter reviews the necessary background materials for the rest of this dissertation. Chapter 2.1 presents a high-level view of the contemporary GPU architecture, it also introduces the recently explored multi-GPU systems. Chapter 2.2 summarizes the background on program synchronizations, including lock mechanism and transactional memory. Chapter 2.3 and 2.4 briefly explain a set of concepts in memory consistency models and cache coherence protocols. Finally, Chapter 2.5 describes the 3D graphics pipeline and the corresponding architecture support.

## 2.1 GPU Architectures

### 2.1.1 High-level Architecture and Programming Model

The high-level architecture of a GPU is shown in Fig. 2.1. A GPU application starts on the CPU; the operations to be executed by the GPU are packaged in the kernel functions. Every time the kernel function is called, CPU will launch it onto the GPU through a compute acceleration API, such as CUDA [177] or OpenCL [112]. Each kernel function is composed of many threads which perform the same operation on different data in parallel; this programming model is called Single-Instruction, Multiple-Thread (SIMT). The thread hierarchy (Fig. 2.2) organizes threads into *thread block arrays* (CTAs) in NVIDIA GPUs or *workgroups* in AMD GPUs, each of which is dispatched to one of the Streaming Multiprocessor (SM) cores in the GPU

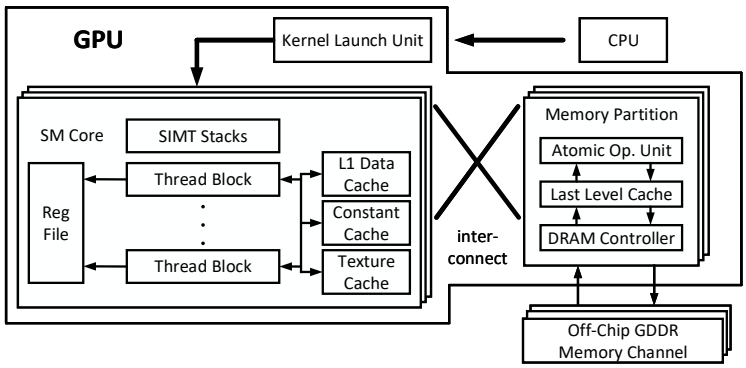



**Figure 2.1:** High-level GPU Architecture as seen by programmers [31].

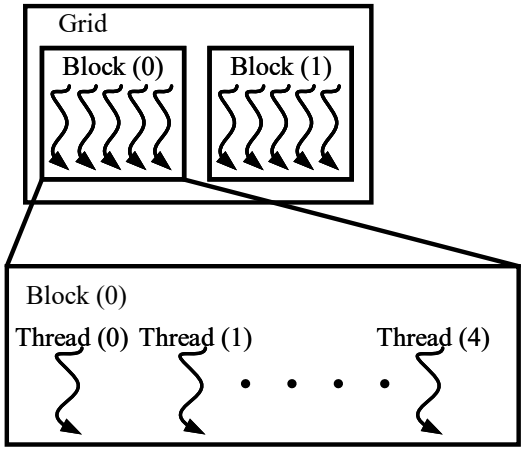



**Figure 2.2:** GPU programming model.

architecture. Threads are scheduled in batches (called *warps* in NVIDIA GPUs and *wavefronts* in AMD GPUs) in each SM core, each with 32–64 threads. A SIMT stack is used to handle branch divergence among threads in the same batch [127]. Threads within a thread block can communicate via an on-chip scratchpad memory called the *shared memory* in CUDA (or *local memory* in OpenCL), and can synchronize via hardware barriers. The SM cores access a distributed, shared last-level cache and off-chip DRAM via an on-chip interconnection network. From a programming

perspective, GPU memory is divided into several spaces, each with its own semantics and performance characteristics: for example, data which is shared by the threads of a thread block is stored in the *shared memory*, while data which is shared by all thread blocks is stored in the *global memory*.

The GPU launch unit automatically dispatches as many thread blocks as the GPU on-chip resources can handle in parallel. If the number of thread blocks is larger than what the GPU can support, the remaining threads will be launched when previous thread blocks have finished and released sufficient resources. This allows GPU applications to generate as many threads as necessary without introducing significant overhead.

As we all know, GPU was originally built as an accelerator for graphics processing, so there are many specific architecture components for rendering. We introduce them in Chapter 2.5.2.

### 2.1.2 Hierarchical Multi-Module and Multi-GPU Systems

Because the end of Moore's Law is limiting continued transistor density improvement, scaling GPU performance by simply integrating more resources in a monolithic chip has gradually reached the limit. Future GPU architectures will consist of a hierarchy in which each GPU is split into multiple GPU Modules (GPMs) [62]. Recent work has demonstrated the benefits of creating GPMs in the form of single-package multi-chip modules (MCM-GPU) [29]. Researchers have also explored the possibility of presenting a large hierarchical multi-GPU system to users as if it were a single larger GPU [143], but mainstream GPU platforms today largely just expose the hierarchy directly to users so that they can manually optimize data and thread placement and migration decisions.

The constrained bandwidth of the inter-GPM/GPU links is the main performance bottleneck on hierarchical GPU systems. To mitigate this, both MCM-GPU and Multi-GPU have scheduled adjacent CTAs to the same GPM/GPU to exploit inter-CTA data locality [29, 143]. These proposals map each memory page to the first GPM/GPU that touches it to increase the likelihood of capturing data locality. They also extended a conventional GPU software coherence protocol to the L2 caches, and they showed that it worked well for traditional bulk-synchronous programs.

More recently, CARVE [253] proposed to allocate a fraction of local GPU DRAM as cache for remote GPU data, and enforced coherence using a simple protocol filtered by tracking whether data was private, read-only, or read-write shared. However, as CARVE does not track sharers, CARVE broadcasts invalidation messages to all caches for read-write shared data.

Apart from the above work that focused on general-purpose applications, researchers have also explored multi-GPU systems for graphics processing. GPUpd [114] reduced redundant geometry processing in conventional Split Frame Rendering (SFR) by exchanging primitives through high-speed inter-GPU links, such as NVIDIA's NVLink [160] and AMD's XGMI [1]. Xie et al. proposed OO-VR framework to accelerate VR by improving data locality [246].

## 2.2 Synchronization

GPU is a computing platform for highly parallel applications. In multithreaded programs, synchronizations are necessary to protect shared data accesses from data-races, thereby guaranteeing correct data communication among threads. Generally, synchronization orders are specified by programmers with different primitives, such as barriers, locks, and so on. This section first introduces the widely used locks. Then, an alternative, Transactional Memory (TM), is described to address the potential problems of lock in GPU programming.

### 2.2.1 Locks

A *lock* is a software data object that creates mutual exclusion (i.e., critical section) of shared data in memory. If a thread wants to access the shared data, it has to acquire the associated lock firstly. At any time instant, a lock only can be acquired by at most one thread, so the thread holding the lock has exclusive access to the shared data. The lock should be released after operations on shared data are finished, so that other threads can acquire the lock if needed. Multiple locks which have cyclic dependence can potentially create deadlocks. Assuming both thread T1 and T2 need to acquire both locks L1 and L2 to progress, but each thread is holding one of the two locks and cannot release its own lock until both locks are acquired. In this case, thread T1 and T2 will be blocked indefinitely. One way to avoid this deadlock is to

maintain a global order of lock acquisition.

To simplify programming, we can associate all shared data to one or very few lock(s) – known as *coarse-grained locks*. Even though this approach can reduce programming complexity and largely avoid deadlocks, it will potentially serialize most executions and significantly hurt performance. Dividing shared data into multiple smaller critical sections – *fine-grained locks* – can maximize parallelism, but a larger number of locks can complicate programming and increase the risk of deadlocks. With locks, programmers usually need to make conservative assumptions that different threads will interfere during execution, although they may not actually interfere at runtime. This can throttle available parallelism and reduce performance. Hence, the difficulty in using locks for both correct synchronization and high performance will restrict them to experienced programmers.

### 2.2.2 Transactional Memory

The massive number of threads in GPU programs can make above situation of lock even more challenging. Transactional Memory (TM) can mitigate this problem by decoupling the functional behavior of code (i.e., operation performed by each transaction) from its performance (i.e., how the transactions are executed). In TM systems, programmers need to change lock-based critical section to transaction, a code fragment which should be performed atomically (i.e., either all operations in a transaction complete successfully or none of them appear to start executing). Conflicts (data-races) are detected automatically in a deadlock-free manner with underlying software or hardware mechanisms. TM also keeps restarting aborted transactions until they finish without any conflicts.

TM designs can be categorized along two axes: conflict detection and version management. In eager conflict detection (e.g., LogTM [149, 252]), an inconsistent read or update attempt by a transaction is detected when the access is made, and one of the conflicting transactions is aborted. Lazy conflict detection (e.g., TCC [89]) defers this until later: often, the entire transaction log is validated during the commit process, and conflicts are discovered only then. In principle, the lazy technique can make better conflict resolution decisions because the entire transaction is known, but has longer commit/abort latencies because the entire transaction must be verified

atomically. Typically, eager conflict detection leverages an existing CPU coherence protocol. However, GPUs lack an efficient coherence protocol that can be leveraged for eager conflict detection [196].

Version management can also be eager or lazy. Lazily-versioned TMs (e.g., TCC [89]) add transactional accesses to a *redo log*, which is only written to memory when the transaction has been validated and commits; if the transaction aborts, the redo log is discarded. In eager version management (e.g., LogTM [149, 252]), the transaction writes the new value directly to the memory hierarchy, but keeps the old value in an *undo log*; if a transaction aborts, the undo log is written to memory.

## 2.3 Memory Consistency Model

A *memory consistency model* defines which sequences of values may be legally returned from the sequence of load operations in each program thread. For example, the following code snippet from [155, 224] represents a common synchronization pattern found in many workloads:

| core C0 | core C1 |
|---|---|
| `data = new` | |
| `done = true` | `while (!done) {` |
| *weakly ordered models* | `}` *// wait for new data value* |
| *need a memory fence here* | *. . . use new data. . .* |

The question is, should core C1 be allowed to see `done=true` even if `data=old`? This is clearly not the intended behaviour, since C1 could see a stale copy of `data`; nevertheless, it is allowed by many commercial CPUs and all extant GPUs [16].

Sequential Consistency (SC) [123] most closely corresponds to most programmers' intuition: it requires that (a) memory operations appear to execute and complete in program order, and (b) all threads observe stores in the same global sequence. In SC, an execution where `done=true` when `data=old` is illegal because either (a) the writes to `data` and `done` were executed out of order by core C0, or (b) they were executed in one order by C0 but observed in a different order by C1.

Weak consistency models, on the other hand, allow near-unrestricted reordering of loads and stores in the program, provided that data dependencies are respected; such reordering typically occurs during compilation and during execution in the processor. Special *memory fence* instructions must be used to restrict reordering and restore sequentially consistent behaviour: in the example above, a fence is needed to ensure that the store to `data` completes before the store to `done`. However, missing fences can be very difficult to find in a massively multithreaded setting like a GPU; conversely, adding too many fences compromises performance.

Since compilers can reorder or elide memory references (e.g., via register allocation), a programming language must also define a memory model. Due to the range of consistency models present in extant CPUs, languages like Java [137] or C++ [38] guarantee SC semantics *only* for programs that are Data-Race-Free (DRF) (i.e., properly synchronized and fenced); this is known as DRF-0 [11]. The Heterogeneous-Race-Free (HRF) model recently proposed for hybrid CPU/GPU architectures further constrains DRF-0 by requiring proper scoping [98]. NVIDIA has also formalized their PTX memory model with scope annotations [135].

## 2.4 Cache Coherence Protocol

In systems with private caches, a *cache coherence protocol* ensures that writes to a *single* location are ordered and become visible in the same order to all cores [45]; the aim is to make caches logically transparent. Since caches are ubiquitous, providing coherence is a fundamental part of implementing any memory consistency model.

There are two critical invariants for coherence protocol definition: Single Writer Multiple Reader (SWMR) invariant and Data-Value invariant [155, 224]. For any given memory location, at any given moment in time[1], SWMR requires that there is only a single core that may write it (and that may also read it) or some number of cores that may read it. In addition to SWMR, Data-Value invariant requires that the value of a given memory location is propagated correctly. This invariant states that the value of a memory location at the start of an epoch is the same as the value of

[1]The SWMR invariant only needs to be enforced in *logical time*, not physical time. This subtle issue enables many optimizations that appear to – but do not – violate this invariant. The proposed Relativistic Cache Coherence (RCC) in Chapter 3 leverages this insight and enforces SC in logical time.

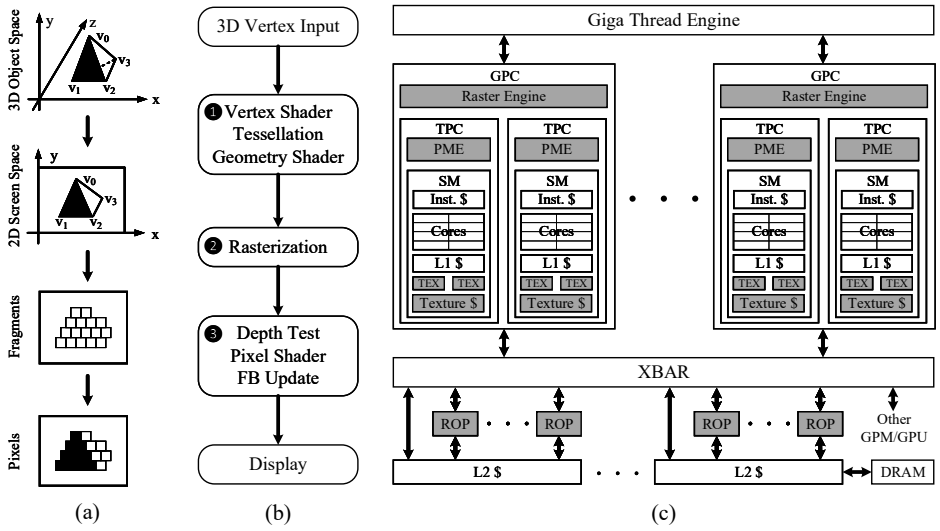



**Figure 2.3:** (a) A simple illustration of graphics pipeline. (b) General operations of a graphics pipeline. (c) GPU microarchitecture (shaded components are specific to graphics rendering).

the memory location at the end of its last read-write epoch.

The vast majority of coherence protocols, called "invalidation protocols", are designed explicitly to maintain these invariants. A core cannot overwrite a location until all other sharers get invalidated. If a core wants to read a location, it has to guarantee that no other cores have cached that location in read-write state. Typical examples are MESI-like coherence protocols, which have been widely used in CPUs.

## 2.5 Graphics Processing

### 2.5.1 The 3D Rendering Pipeline

The function of the 3D graphics pipeline, illustrated in Figure 2.3(a), is to project a 3D scene, full of objects that often consist of thousands of primitives (usually triangles), onto a 2D screen space. On the screen, primitives end up as thousands of pixels; the pixels are accumulated in a Framebuffer (FB) and sent to the display unit once rendering is complete. Producing a single frame typically involves hundreds or thousands of draw commands to render all objects in the scene, all of which must

go through the graphics pipeline.

Figure 2.3(b) shows the graphics pipeline defined by DirectX [39]; other pipelines (e.g., OpenGL [207]) are similar. The key pipeline stages are geometry processing, rasterization, and fragment processing. Geometry processing ❶ first reads vertex attributes (e.g., 3D coordinates) from memory and projects them onto the 2D screen space using vertex shader. Projected vertices are grouped into primitives (typically triangles); some primitives may then be split into multiple triangles through tessellation, which creates smoother object surfaces for a higher level of visual detail. Generated primitives that are outside of the current viewport are then culled by geometry-related shaders. The next stage, called rasterization ❷, converts primitives into fragments, which will in turn be composed to produce pixels on the screen. Fragment processing ❸ has two main tasks: performing the depth test (Z test) and computing fragment attributes (e.g., colour). The depth test discards fragments whose depth value (i.e., the distance from the camera) is bigger than prior ones. Fragments that passed the depth test have their attributes computed using a pixel shader. Finally, the shaded fragments (which maybe opaque or semi-transparent) are composed to generate the pixels, which are written to the FB and eventually sent to the display unit.

### 2.5.2 The Graphics GPU Architecture

Figure 2.3(c) illustrates the overall microarchitecture of NVIDIA's Turing GPU [164]. It consists of multiple Graphics Processing Clusters (GPCs) connected to multiple Render Output Units (ROPs) and L2 cache slices via a high-throughput crossbar. A Giga Thread Engine distributes the rendering workload to the GPCs based on resource availability.

GPCs perform rasterization in dedicated Raster Engines, and organize resources into multiple Texture Processing Clusters (TPCs). Within each TPC, the PolyMorph Engine (PME) performs most non-programmable operations of the graphics pipeline except rasterization (e.g., vertex fetching, tessellation, etc). The Streaming Multiprocessor (SM), consisting of hundreds of shader cores, executes the programmable parts of the rendering pipeline, such as the vertex shaders, pixel shaders, etc.; to reduce hardware complexity, SMs schedule and execute threads in SIMD fashion

(i.e., warps of 32 or 64 threads). Texture Unit (TEX) is a hardware component that does sampling. Sampling is the process of computing a color from an image texture and texture coordinates.

On the other side of the interconnect, ROPs perform fragment-granularity tasks such as the depth test, anti-aliasing, pixel compression, pixel blending, and pixel output to the FB. A shared L2 cache, accessed through the crossbar, buffers the data read from off-chip DRAM.

# Chapter 3

# Efficient Sequential Consistency via Relativistic Cache Coherence

This chapter proposes Relativistic Cache Coherence (RCC) which can efficiently enforce Sequential Consistency (SC) in GPUs. In contrast with prior GPU SC work [220], RCC does not explicitly classify read-only/private data: instead, a predictor naturally learns to assign short cache lifetimes to frequently written shared data. Unlike prior GPU coherence work [221], RCC operates in logical time; as a result, stores acquire write permissions instantly but still maintain SC. RCC underpins a sequentially coherent GPU memory system that outperforms all previous proposals and closes the gap between SC and weak consistency in GPUs. RCC is 29% faster than the best prior SC proposal for GPUs, and within 7% of the best non-SC design.

Sequential consistency — the most intuitive model — requires that (a) all memory accesses appear to execute in program order and (b) all threads observe writes in the same sequence [123]. To ensure in-order load/store execution, a thread must delay issuing some memory operations until preceding writes complete; we refer to these delays as SC stalls. Moreover, since all cores must observe writes in the same order, stores cannot complete until they are guaranteed to be visible to all other threads and cores. Because of these restrictions, few modern commercial CPUs have supported SC [251]; typically SC is relaxed to permit limited [181, 218] or near-arbitrary reordering [26, 104, 205, 230]; programmers must then insert

*memory fences* for specific memory operations, in essence manually reintroducing SC stalls. GPUs manufacturers have followed suit: both NVIDIA and AMD GPUs exhibit weak consistency [16] similar to Weak Ordering (WO) [68] or Release Consistency (RC) [78] models.

Correctly inserting fences is difficult, however, especially in GPUs where all practical programs are concurrent and performance-sensitive. The authors of [16] found missing fences in a variety of peer-reviewed publications, and even vendor guides [204]. Such bugs are very difficult to detect: some occurred in as few as 4 out of 100,000 executions in real hardware, and most occurred in fewer than 1% of executions [16]. Code fenced properly for a specific GPU may not even work correctly on other GPUs from the same vendor: some of these bugs were observable in Fermi and Kepler but not in older or newer microarchitectures [16].

SC hardware is desirable, then, if it can be implemented without significant performance loss. Recent work [94, 220] has argued that this is possible in GPUs: unlike CPUs, which lack enough Instruction Level Parallelism (ILP) to cover the additional latency of SC stalls, GPUs can leverage abundant Thread Level Parallelism (TLP) to cover most SC stalls. The authors of [220] propose reducing the frequency of the remaining SC stalls by relaxing SC for read-only and private data; classifying these at runtime, however, requires complex changes to GPU core microarchitecture and carries an area overhead in devices where silicon is already at a premium. Moreover, both studies focused on SC built using CPU coherence protocols (MOESI and MESI) with write-back L1 caches. In GPUs, however, write-through L1s perform better [221]: GPU L1 caches have very little space per thread, so a write-back policy brings infrequently written data into the L1 only to write it back soon afterwards. Commercial GPUs have write-through L1s and require bypassing/flushing L1 caches to ensure intra-GPU coherence [21, 165, 167].[1] Compared to the best GPU relaxed consistency design, the performance cost of implementing SC appears to be closer to 30% [221].

In the rest of this chapter, we describe RCC and demonstrate how it achieves

[1]GPU vendor literature and some prior work use "coherence" to describe automatic page-granularity data transfer between the host CPU and the GPU's shared L2; some academic proposals use "system coherence" for the same concept [188]. To the best of our knowledge, no existing GPU product implements hardware-level intra-GPU coherence.

both easy programming and high performance by efficiently enforcing SC with logical timestamps.

## 3.1 GPUs vs. CPUs: A Consistency and Coherence Perspective

**Consistency.** Modern multi-core CPUs have largely settled on weak memory models to enable reordering in-flight memory operations [104, 181, 205, 218, 230]: because CPUs support at most a few hardware threads, the Memory Level Parallelism (MLP) obtained from reordering memory operations is key to performance. GPUs, on the other hand, buffer many tens of warps (e.g., 48–64 [21, 165, 167]) of 32–64 threads in each GPU Streaming Multiprocessor (SM) core, and when one warp is stalled (because of an L1 cache miss, for example), the core simply executes another.

With fine-grained multithreading, GPUs can amortize hundreds of cycles of latency *without* reordering memory operations; recent work [94, 220] has suggested that the same mechanism can cover the ordering stalls required by SC. Indeed, hardware techniques that reorder accesses — such as store buffers — are either too expensive or ineffective in GPUs, so leaving them out does not hurt performance [220].

**Coherence.** CPU caches are generally kept coherent by tracking each block's sharers and invalidating all copies before writing the block. Most protocols in commercial products are quite similar: they have slightly different states (MESI, MESIF, MOESI, etc.) or sharer tracking methods, but the basic operation relies on request-reply communication between cores and an ordering point such as a directory.

All commercial GPUs we are aware of lack automatic coherence among private L1 caches: in GPU vendor literature, "coherence" refers only to the boundary between the host CPU and the GPU. NVIDIA Pascal allows the GPU to initiate page faults and synchronize GPU and CPU memory spaces [92], but intra-GPU coherence requires bypassing the L1 caches [16]. AMD Kaveri APUs bypass and flush the L1 cache for intra-GPU coherence, and bypass the L2 for CPU-GPU sharing [222]. Details for ARM Mali GPUs are scant, but it appears that the coherence boundary terminates at the GPU shared L2 cache and does not include the L1s [198].

Efficient intra-GPU coherence implementations are subject to different constraints than CPUs. GPUs have 15, 32, or even 56 SM cores [21, 92, 165, 167], simultaneously executing around 100,000 threads. While some prior studies [94, 220] (and our motivation study) have assumed CPU-like MESI coherence, a realistic implementation could face simultaneous coherence requests from tens of thousands of threads; just the buffering requirements would be prohibitive [221].

The other coherence protocol work for GPUs leveraged two observations: (a) that write-through caches provide a natural ordering point at the L2, and (b) that inter-core synchronization can be implicit via a shared on-chip clock [221]. A cache that requests read permissions receives a read-only copy with a limited-time *lease*; this copy may be read until the shared clock has ticked past the lease time. Two protocols are proposed in [221]: TC-STRONG (TCS) can support SC if the core does not reorder accesses, but stalls stores at the L2 to ensure that all leases for the address have expired; TC-WEAK (TCW) allows stores to proceed without stalling, but compromises write atomicity and cannot support SC.

## 3.2 Bottlenecks of Enforcing Sequential Consistency

To trace the roots of performance loss created by enforcing SC, we evaluated an SC implementation similar to prior work [94, 220] but with GPU-style write-through L1 caches (see Chapter 3.5 for simulation setup). We examined memory-intensive workloads with and without inter-workgroup sharing previously used to evaluate GPU cache coherence [221]; the inter-workgroup benchmarks rely on inter-core coherence traffic, while the intra-workgroup benchmarks communicate only within each GPU core. We found SC stalls to be relatively infrequent (Figure 3.1a): in only one case were more than 20% memory operations ever stalled because of SC; this supports prior arguments [94] that the massive parallelism available in GPUs can cover most ordering stalls introduced by SC.

We next examined the cause of each stall — i.e., the type of the preceding memory operation from the same thread. Figure 3.1b shows that most SC stall cycles are spent waiting for a previous store (or atomic) instruction to complete; indeed, in most cases, nearly all stall delays are due to waiting for prior writes. This is because average store latencies are very long: for workloads with inter-workgroup


**Figure 3.1:** SC stalls are (a) infrequent, but (b) mostly due to preceding stores; (c) average store latencies are much longer than load latencies; (d) zero invalidate latency leads to substantial speedup for inter-workgroup sharing workloads.

communication, store latencies are often much longer than load latencies (2.4× gmean), and up to 3.7× longer (Figure 3.1c).

This makes sense: to maintain SC, each store must receive an ACK before completing to ensure that the new value has become visible to all cores. There are two parts to this latency: one — the round-trip to L2 — is unavoidable with the write-through L1 caches found in GPUs. The other part is ensuring exclusive coherence permissions: in our MESI-based experiment the write waits until other sharers have invalidated their copies, while in timestamp-based GPU coherence protocols like TC-STRONG [221] the store waits for all read leases to expire. Long-latency stores can affect performance not only by delaying SC stall resolution, but also by occupying buffer space or stalling same-cacheline stores from other threads in MSHRs until the ACK is received.

To find out whether coherence delays are significant, we implemented an idealized variant of SC where acquiring read and write permissions is instant (SC-IDEAL). Figure 3.1d shows the speedup of SC-IDEAL over realistic SC: for workloads with inter-workgroup sharing, idealizing coherence yields a substantial performance improvement (1.6× gmean); workloads with only intra-workgroup sharing see no benefit. In the next section, we address the store latency and SC stall problems by maintaining SC in logical time.

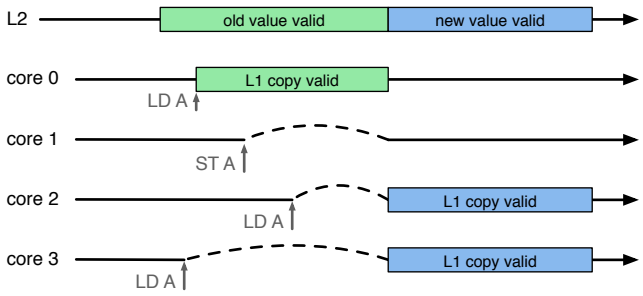



**Figure 3.2:** High-level view of enforcing SC in logical time. Logical time increases left to right; all cores that observe the new value of A must advance their logical times past that of the store.

## 3.3 Enforcing Sequential Consistency in Logical Time

To address the problems identified above, we leverage Lamport's observation that ordering constraints need to be maintained only in *logical time* [122], prior observations that SC can be maintained logically [84, 131], and the recent insight that logical timestamps can be used directly to implement a coherence protocol [255]. We propose Relativistic Cache Coherence (RCC), a simple, two-state GPU coherence protocol where each core maintains — and independently advances — its own logical time. The L2 keeps track of the last logical write time for each cache block; whenever a core accesses the L2, it must ensure that its own logical time exceeds the last write time of the relevant block. Data may be cached in L1s for a limited (logical) time, after which the block self-invalidates.

Figure 3.2 shows how RCC maintains SC in logical time. First, core 0 loads address A, and receives a fixed-time lease for A from the L2, which records the lease duration; core 0 may then read its L1 copy until its logical time exceeds the lease expiration time. Core 1 writes to A, but to do this it must advance its own logical time to past the lease given out for A; this step (dashed line) is equivalent to establishing write permissions in other protocols, but occurs instantly in RCC. Core 2 loads A from L2 and advances its logical time past the time of core 1's write. Finally, core 3 also reads A. The load is *logically* before the store to A (because core 3's logical clock is earlier than A's), but *physically* the write to A has already happened, and only the new value of A is available at the L2. Core 3 thus receives

| | MESI | TCS | TCW | **RCC** |
|---|---|---|---|---|
| SC support? | yes | yes | no | **yes** |
| stall-free store permissions ? | no (invalidate sharers) | no (wait until lease expires) | yes (but stall for fences) | **yes** |

**Table 3.1:** SC and coherence protocol proposals for GPUs.

the new value of A, but must also advance its logical time to that of A's write.

Naturally, the cost of synchronization does not entirely disappear: advancing a core's logical time may cause other L1 cache blocks to expire. In essence, we are exchanging a reduction in store latency for A for potentially some additional L1 misses on other addresses. While this would be problematic for latency-sensitive CPUs, throughput-focused GPUs were explicitly designed to amortize this kind of cost; we will show that in GPUs this tradeoff is worth making.

Lamport's logical time has recently been proposed as a coherence mechanism for CPUs [255, 257]. Performance, however, was subpar even compared to the much simpler MSI protocol, even though the proposed protocol was more complex than RCC and relied on a complex speculation-and-rollback mechanism. RCC is not only much simpler, but actually outperforms the best existing GPU protocols.

Next, we describe Relativistic Cache Coherence, a new GPU coherence protocol that supports SC (like TCS) but allows stores to execute without waiting for write permissions (like TCW). Table 3.1 compares RCC with prior protocols proposed for GPUs in the context of SC.

## 3.4 Relativistic Cache Coherence (RCC)

Relativistic Cache Coherence leverages the observation by Lamport [122] that consistency need only be maintained *in logical time*. Two threads may see the memory as it was at two different logical times, as long as each *only* observes all writes logically before — and never sees any writes logically after — its own logical "now." In RCC, cores maintain separate logical times, which become synchronized only when read-write data is shared.

Like all library coherence protocols [119, 211, 221, 255, 257], RCC allows L1 caches to keep private copies of data only for limited-time "leases" granted for each

requested block; when a lease expires, the block self-invalidates in L1 without the need for any coherence traffic. Writes to a block must ensure that no valid copies are present in any L1s by ensuring that the write time exceeds the expiration time of all outstanding leases. In RCC, leases are granted and maintained in logical time, so writes can complete instantly by advancing the writing core's logical clock.

### 3.4.1 Logical Clocks, Versions, and Leases

In relativistic coherence, each core maintains, and independently advances, its own logical clock (*now*). Similarly, each shared cache (L2) block maintains it own logical version (*ver*), equal to the logical time of the last write to this block.

Since the L2 grants per-block read leases to private L1 caches, it keeps track of when the last lease for a given block will expire (*exp*). Each L1 cache also keeps track of the *exp* it was given by the L2. Different L1s may have different *exp*s for the same block, but none will exceed the latest *exp* in L2. Because L1s are write-through, they do not need to record *ver* for each block.

A unique, global SC ordering of memory accesses is maintained in logical time by applying three rules:

1. Core C reading cache block B must advance its logical time *now* to match B's current version *ver* if $B.ver > C.now$. This ensures that C cannot use B to compute new data values with logical times $< B.ver$, i.e., that C does not observe a value of B "from the future."

2. Core C writing cache block B must advance B's *ver* to C's *now* if $B.ver < C.now$, and advance its own *now* to B's *ver* if $B.ver > C.now$. This ensures the new value of B cannot be used for computation in cores whose *now* is earlier, i.e., that B is not "sent back in time."

3. Core C writing cache block B must advance its *now* as well as the new $B.ver$ beyond the expiration time *exp* of the last outstanding lease for B. This ensures that the new value of B does not "leak:" i.e., that any values computed from the new value of B by other cores cannot coexist in their L1s with the old value of B.

| core | | C0 | | | C1 | | | shared | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| C0 | C1 | L1 cache | | | L1 cache | | | L2 cache | | | |
| memory op | | now | A.exp | B.exp | now | A.exp | B.exp | A.ver | A.exp | B.ver | B.exp |
| — | — | 20 | ~~10~~ | ~~10~~ | 0 | 10 | 10 | 0 | 10 | 30 | 10 |
| ST A | — | 20 | ~~10~~ | ~~10~~ | 0 | 10 | 10 | **20** | 10 | 30 | 10 |
| LD B | — | **30** | ~~10~~ | **40** | 0 | 10 | 10 | 20 | 10 | 30 | **40** |
| — | ST B | 30 | ~~10~~ | 40 | **41** | ~~10~~ | ~~10~~ | 20 | 10 | **41** | 40 |
| — | LD A | 30 | ~~10~~ | 40 | 41 | **51** | ~~10~~ | 20 | **51** | 41 | 40 |
| ST B | — | **41** | ~~10~~ | ~~40~~ | 41 | 51 | ~~10~~ | 20 | 51 | 41 | 40 |
| ST A | — | **52** | ~~10~~ | ~~40~~ | 41 | 51 | ~~10~~ | **52** | 51 | 41 | 40 |
| — | LD A | 52 | ~~10~~ | ~~40~~ | 41 | 51 | ~~10~~ | 52 | 51 | 41 | 40 |

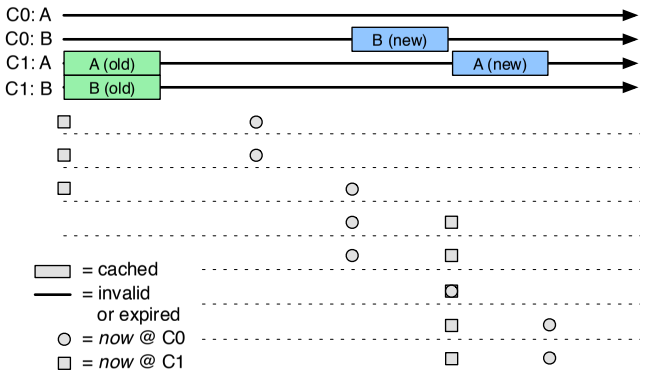



**Figure 3.3:** RCC executing accesses to two addresses (A and B) from two cores (C0 and C1). The table (left) tracks each core's logical time (*now*), and each cache block's version (*ver*) and read lease expiration (*exp*) after each instruction has executed; the rows represent the order of instructions as executed in physical time. The diagram (right) illustrates the lease durations in each cache (top) and how the logical time *now* advances in each core as the corresponding operations from the table execute (bottom); logical time flows left to right while physical time flows top to bottom. Bold values denote changes since the last step; crossed-out leases have expired.

The logical *now* times of memory operations provide a sequentially consistent ordering. Provided the core scheduler is modified to ensure that only one global memory access per warp is issued at any given time, RCC supports SC.[2]

### 3.4.2 Example Walkthrough

Figure 3.3 shows how RCC operates on a sequence of instructions from two different cores. Initially, C0's cache has neither A and B (since $now > exp$) and core C1 has both. In the shared L2 cache, B has since been written by a third core and has $ver = 30$; because C1's *now* has not advanced past 10, however, it may still read its cached copy of B.

First, core C0 writes A, which updates the *A.ver* in the L2 (rule 2); C1 still has $now = 0$ and can read its old copy of A. C0 then reads B, which receives a new lease (until logical time 40) but must advance its *now* past *B.ver* (rule 1).

Next, C1 writes B, which updates *B.ver* and *C1.now* to 41, past the last outstanding lease for B (rule 3). This step enforces SC ordering between the two cores: C1 next reads A, and is forced to pick up the value written by C0.

Finally, C0 writes B, advancing its *now* past the previous write to B (rule 2), and then A, advancing past the last lease for A (rule 3). Because *C1.now* is earlier, however, C1's next load will happen logically before C0's write to A, and will not observe the new value. Note that SC has been maintained, as the overall behaviour is explained by the following sequential interleaving: `C0: ST A, LD B; C1: ST B, LD A, LD A; C0: ST B, ST A`.

### 3.4.3 Coherence Protocol: States and Transitions

The full state transition diagram for RCC, including both stable and transient states, is shown in Figure 3.4.

**Stable states.** RCC has two stable states: V (VALID) and I (INVALID). Blocks loaded into the L1 transition to the V state, and may be read until they are evicted, written, or until their leases expire, at which point they self-invalidate and transition to the I state. Stores (and atomic read-modify-write operations) may occur in both

[2]The proof that RCC supports SC is essentially the same as for Tardis [256], we refer the interested reader there. The main difference is that RCC permits a sequence of unobserved stores to share the same logical version; the SC ordering in that case is provided by the physical arrival times at the L2.

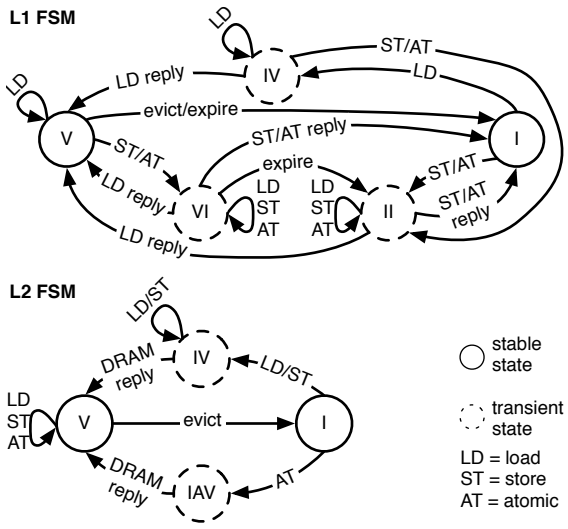



**Figure 3.4:** Full L1 and L2 coherence FSMs of RCC (stable and transient states).

V and I states; the request is forwarded to the L2 (GPU L1s are write-through, write-no-allocate), and the block eventually transitions to I after the store ACK is received. Expired blocks in V state ($exp < now$) are treated exactly the same way as blocks in I state for memory operations and cache replacement purposes.

The L2 also only has V and I states. L2 misses retrieve the value from memory and transition to V. Because the L2 is write-back (like in commercial GPUs ), the V state allows reads, writes, and atomic operations; a block transitions to I only when evicted by the L2 cache replacement algorithm.

**Transient states**. L1 blocks also have three transient states: IV, II, and VI; the first two are required for correctness, while the third is a GPU-specific optimization.

IV indicates that a load request missed in the L1 and a GETS request has been sent; further load requests for the same cache block will be stored in the MSHR without more GETS requests, and the block will transition to V once the DATA response has been received. Stores received while in IV state cause a transition to II.

II indicates that a store (or atomic) request has been sent to the L2, and the cache is waiting for an ACK message with the logical time at which the write was

executed (i.e., the new *ver*); this is necessary to maintain SC. While in II state, any DATA response from the L2 will be forwarded to the core, but the block will stay in II.

VI is an optimization of the II state when the block was valid before the write; in VI, the block can still be read by other warps until the ACK message with the new *ver* is received from the L2 cache; this is important in GPUs because round-trip access latencies to L2 can be hundreds of cycles [241].

To permit non-blocking misses, the L2 coherence controller has two transient states:

IV buffers new GETS and WRITE requests in the relevant MSHR, keeping track of the maximum *now* times from the reading and writing processors. Once the data arrives from DRAM, the block's version is updated to reflect any writes in the MSHR and a new lease is generated to satisfy any readers.

IAV indicates an ATOMIC operation received in an invalid state; this stalls any further L1 requests until the block has been retrieved from DRAM, its version has been established, and the atomic operation has completed.

Figure 3.5 shows the complete state transition table, including the generated messages and MSHR management details.

RCC has fewer states and transitions than prior art. Earlier logical timestamp coherence work [255] requires three stable states each for L1 and L2 (transient states are not described), as well as MESI-like recall and downgrade mechanisms to implement a private writable state; such inter-core communication is precisely the source of the SC store latencies we wish to avoid. Prior GPU coherence work also has more states (13 total) and transitions than RCC. In the SC-capable variant, a private state is used to avoid store stalls for private data; in the weakly ordered version, non-fenced stores do not stall but SC support is not possible. RCC employs logical timestamps to acquire store permissions instantly, and does not require private or exclusive states.

| L1 state | requests from processor core | | | L1 events | | L2 responses | | |
|---|---|---|---|---|---|---|---|---|
| | load | store | atomic | evict | expiry | DATA | RENEW | ACK |
| I | GETS {now = L1.now, exp = D.exp} → **IV** | WRITE {now = L1.now} → **II** | ATOMIC {now = L1.now} → **II** | — | — | — | — | — |
| V | cache hit | WRITE {now = L1.now} → **VI** | ATOMIC {now = L1.now} → **VI** | → **I** | → **I** | — | — | — |
| IV | add to MSHR | WRITE {now = L1.now} → **II** | ATOMIC {now = L1.now} → **II** | stall | — | L1.now = max(L1.now, M.ver) D.exp = M.exp → **V** | D.exp = M.exp → **V** | — |
| II | GETS {now = L1.now, exp = D.exp} | WRITE {now = L1.now} | ATOMIC {now = L1.now} | stall | — | L1.now = max(L1.now, M.ver) read resp? D.exp = M.exp MSHR.empty? → **V**, else → **VI** atomic resp? MSHR.empty? → **I**, else → **II** | D.exp = M.exp → **VI** | L1.now = max(L1.now, M.ver) MSHR.empty? → **I** |
| VI | cache hit | WRITE {now = L1.now} | ATOMIC {now = L1.now} | stall | → **II** | L1.now = max(L1.now, M.ver) read resp? D.exp = M.exp MSHR.empty? → **V**, else → **VI** atomic resp? MSHR.empty? → **I**, else → **II** | — | L1.now = max(L1.now, M.ver) MSHR.empty? → **I** else → **II** |

**(a)** L1 state transition table for RCC.

| L2 state | requests from L1 | | | L2 events | memory responses |
|---|---|---|---|---|---|
| | GETS | WRITE | ATOMIC | evict | DATA |
| I | DRAM FETCH MSHR.lastrd = M.now → **IV** | DRAM FETCH MSHR.lastwr = M.now → **IV** | DRAM FETCH MSHR.lastwr = M.now → **IAV** | — | — |
| V | D.exp = max(D.exp, D.ver+lease, M.now+lease) M.exp > D.ver? RENEW {exp=D.exp} else DATA {exp = D.exp, ver = D.ver} | D.ver = max(M.now, D.ver, D.exp+1) ACK {ver = D.ver} | D.ver = max(M.now, D.ver, D.exp+1) DATA {exp = D.exp, ver = D.ver} | mnow = max(mnow, D.exp, D.ver) dirty? WBACK → **I** | — |
| IV | add to MSHR MSHR.lastrd = max(MSHR.lastrd, M.now) | write to MSHR MSHR.lastwr = max(MSHR.lastwr, M.now) ACK {ver = max(MSHR.lastwr, mnow)} | stall | stall | D.exp = D.ver = mnow MSHR.haswrite? D.ver = max(MSHR.lastwr, mnow) MSHR.hasread? D.exp = max(D.ver+lease, MSHR.lastrd+lease) DATA {exp = D.exp, ver = D.ver} → **V** |
| IAV | stall | stall | stall | stall | D.exp = mnow, D.ver = max(MSHR.lastwr, mnow) DATA {exp=D.ver, ver = D.ver} → **V** |

**(b)** L2 state transition table for RCC.

**Figure 3.5:** State transition tables for RCC. D is the cache block (e.g., D.exp is the expiration time for the block), M represents a received message (e.g., M.ver in an ACK indicates the time when a write will become visible). Arrows signify state transitions. V and I are stable states; IV, VI, II (L1 only) and IAV (L2 only) are transient states. Braces denote coherence message contents; cache block data are included as appropriate. Shaded areas highlight protocol changes required for lease extensions.

| name | granularity | semantics |
|---|---|---|
| *now* | GPU core | logical time seen by this core |
| *exp* | cache block | lease expiration time |
| *ver* | cache block | data version (last write time) |
| *mnow* | mem. partition | max(*exp,ver*) evicted to DRAM |
| *lastrd* | L2 MSHR | latest *now* of any reading core |
| *lastwr* | L2 MSHR | latest *now* of any writing core |

**Table 3.2:** Timestamps used in RCC.

### 3.4.4 L2 Evictions and Timestamp Rollover

Table 3.2 lists all timestamps maintained in RCC and their semantics. Core logical clock *now*, data write version *ver*, and lease expiration time *exp* were described in Chapter 3.4.1.

**L2 evictions.** Because data copies in L1 automatically expire, RCC allows caches to be non-inclusive without requiring the usual RECALL messages, as in prior GPU coherence work [221]. Care must be taken, however, to maintain logical ordering when evicting blocks from L2: if a block were naïvely evicted and then re-fetched without preserving its *ver* and *exp*, it could then be read logically before it was written, or could be written before all leases expire. Singh et al [221] handle this by using an MSHR entry to store the evicted block until the timestamp expires, which limits the number of MSHR entries available for L2 misses.

RCC instead allows the eviction but ensures that, if the block is reloaded from DRAM, reading or writing it will cause any outstanding leases for it to expire. To enforce this, we could keep track of *ver* and *exp* for each block in DRAM, but this would require additional storage provisions in main memory. Instead, we store the maximum *ver* or *exp* of any evicted block as the "memory time" *mnow*, one in each memory partition. To maintain logical ordering, a block loaded from DRAM will have its *ver* and *exp* set to *mnow*: any cores that read or write this block will have to advance their logical time to prevent the issue described above.

Since the L2 is write-back (like in extant GPUs [21, 165, 167]), a WRITE request that misses in L2 will be stored in MSHR while the block is set to IV state and retrieved from DRAM, and any additional write requests are merged into the MSHR. To maintain correct logical write ordering, each MSHR keeps track of *lastwr*, the

highest write time (originating core *now* value) of any WRITE requests received in IV state. WRITE requests with $now \geq lastwr$ update the MSHR data and *lastwr*; write requests with $now < lastwr$ do not change *lastwr* but must be tracked until the final write time is known. The larger of *lastwr* and *mnow* will become the block's *ver*; since this is the logical write time, the store can be acknowledged without waiting for the DRAM response. The store data will remain in the MSHR until the DRAM response arrives.

A similar case arises for read requests that miss in L2. MSHRs keep track of *lastrd*, the latest *now* of any reading cores; this is used to calculate the lease expiration (*exp*) once the block is available, and can be elided to save space (*lastwr* would be used instead).

**Timestamp rollover.** Because timestamps have finite exact representations and keep increasing, they are subject to arithmetic rollover. In our experiments, 32-bit logical timestamps advanced on average once for every 1073 core clock cycles; this corresponds to approximately one rollover per hour at clock speeds found in high-performance GPUs.

In principle, this can be handled simply by setting core *now* clocks to 0, flushing all L1s, setting all L2 *ver* and *exp* entries to 0, and setting all *mnow* values to 0; SRAMs that support flash-clearing [206] make this easy. However, rollover must be processed atomically in the presence of in-flight messages, transient cache states, and independent L2 banks. To implement this correctly, we observe that the L2 is the only coherence actor that actually *increases* timestamps (L1s only copy timestamps received from L2); therefore, the L2 will be the first component to know that rollover is required.

When an L2 partition needs to roll over a timestamp, it first ensures that all other L2 partitions have stalled and set their timestamps to 0. This can be done in many ways, perhaps using a narrow unidirectional ring with the rollover L2 partition sending a STALL flit and all other cores stalling before allowing the flit to continue; when STALL returns to the originating core, all cores will have stalled (in case of concurrent stall requests, lowest L2 partition ID wins). All stalling partitions must set all of their timestamps (including *lastwr* and *lastrd*) to 0; queued requests and MSHR entries are retained, with all timestamps reset to 0. The rollover partition then sends a FLUSH request to all L1s, and waits for responses from all; once these

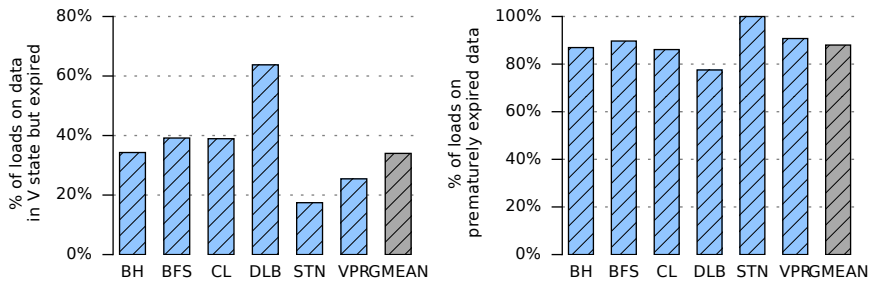



**Figure 3.6:** Left: fraction of loads that find data in V state but expired (either for coherence reasons or prematurely); expiration rate is negligible for intra-workgroup benchmarks. Right: Fraction of expired loads whose blocks that have not changed in L2 (and can be renewed).

have been received, a RESUME flit is sent on the inter-partition ring, and all L2 partitions resume processing requests. An L1 that receives a FLUSH request sets its *now* to 0 and invalidates all entries before replying to L2; addresses with MSHR entries enter the II state, while the remaining addresses transition to I.

### 3.4.5 Lease Time Extension, and Prediction

When the L2 receives a GETS request, it generates a read lease for the block and sends the logical expiration time *exp* back to the requesting L1. So far, we have assumed all leases have the same duration (of 10 in Chapter 3.4.2); intuitively, however, read-only data should receive very long leases to avoid expiration, whereas data shared frequently should receive short leases to avoid advancing the logical time too much when they are written (and thus causing other cache blocks to expire).

When a lease is too short, a load request finds the L1 block in V state but with an expired lease ($now > exp$). Figure 3.6 (left) shows how many L1 cache blocks are in V state but expired when accessed. Sometimes, this is the coherence protocol working as intended and indicates a transitive logically-before relation; at other times, the expiration reflects imperfect lease assignment. Figure 3.6 (right) shows that most such expirations are premature (i.e., the block's L2 entry has not changed).

**Lease extension.** Every such block generates a GETS request and a DATA response from the L2. While the GETS is small, a DATA response includes the full cache block, which poses an unnecessary traffic overhead.

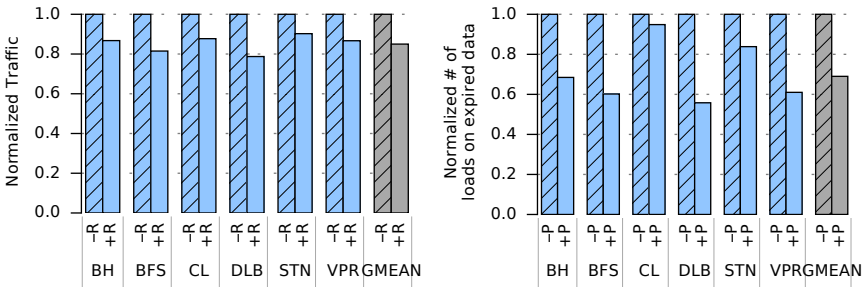



**Figure 3.7:** Left: interconnect traffic with (+R) and without (–R) the renew mechanism. Right: reduction in loads that find expired data in L1, with (+P) and without (–P) the lease prediction.

Since the L2 knows when the block was last written (*ver*), it could potentially *renew* the lease by sending the new lease expiration time but no data (which the L1 already has). Before deciding whether to send RENEW or the full DATA, the L2 needs to know whether the L1's previous lease is older than *ver*; if it is, the L1 may have incorrect data. To provide this information, we modify GETS requests to carry the *exp* time of the expired lease (tracked by the L1): if this is newer than the data version *ver* in the L2, a RENEW grant can be sent. The required protocol changes are shaded in Figure 3.5; note that the complexity cost is minimal, with no additional states and only two new transitions. Prior work [255] also features a lease extension mechanism, but the renew mechanism there relies on keeping track of data versions *ver* in the L1 caches.

Figure 3.7 (left) shows that the renewal mechanism is effective in reducing interconnect traffic for inter-workgroup sharing workloads by 15% (traffic is also reduced for the intra-workgroup benchmarks, but their expiration rates are negligible to begin with).

**Lease prediction**. Although lease extension reduces interconnect traffic, many expirations would not occur to begin with if each block received an optimal lease. We attempted to sweep a range of fixed leases, but found that the performance spread among them was negligible. This is because RCC operates in logical time and most operations advance time in lease-sized amounts; therefore choosing a single fixed lease merely changes the rate at which logical clocks run for everyone. Optimally choosing leases, however, is a non-trivial problem for read-write shared data partly

because the "correct" lease depends on the precise scheduling and interleaving of threads; while the correct lease is obvious for read-only data (= $\infty$), detecting read-only data at runtime requires microarchitectural changes [220].

Instead, we observe that GPU applications tend to work in synchronized phases, with most data being read at the beginning of a phase and written at the end. These (and read-only) data should receive fairly long leases, while data that is shared often (e.g., locks) should receive short leases.

To find the best lease, the L2 initially predicts the maximum lease (2048) for every block. When the block is written, the prediction drops to the minimum (8), and grows ($2\times$) every time a read lease is successfully renewed. This way the L2 quickly learns to predict short leases for frequently shared read-write blocks (such as those containing locks), but long leases for data that is mostly read and blocks that miss in the L2 (e.g., streaming reads). A similar per-block lease prediction mechanism has been proposed [257] for logical-time CPU coherence protocols; unlike our predictor, however, short leases are preferred, and the consistency model is relaxed (to TSO) to maintain performance. Figure 3.7 (right) shows that the predictor reduces expired reads by 31% for inter-workgroup workloads (again, intra-workgroup benchmarks benefit but start with negligible expiration rates).

**Potential livelock.** Because RCC allows cores to read cached data without advancing their logical clocks, a spinlock that only reads a synchronization variable may livelock unless other warps advance the logical time. This optimization is common in multi-core CPUs with invalidate-based coherence, but relies on implicit store-to-load synchronization that is not guaranteed by coherence or consistency requirements. To the best of our knowledge, these kinds of spinlocks are not used in GPUs, as most workloads have enough available parallelism to cover synchronization delays; spinning merely prevents other (potentially more productive) warps from executing (in general, synchronization in GPUs requires different optimizations than in CPUs [243]). Nevertheless, this potential livelock can be avoided by periodically incrementing the logical time *now* (say, by 1 every 10,000 cycles).

| | |
|---|---|
| GPU cores | 16 streaming multiprocessors (SMs) |
| core config | 1.4 GHz, 48 warps × 32 threads, 32 lanes |
| warp scheduler | loose round-robin |
| register file | 32,768 registers (32-bit) |
| scratchpad | 48 KB |
| per-core L1 | 32 KB, 4-way set-associative, 128-byte lines, 128 MSHRs |
| total L2 | 1024 MB = 8 partitions × 128 KB |
| L2 partition | 128 KB, 8-way set-associative, 128-byte lines, 128 MSHRs; 340-cycle minimum latency [241] |
| interconnect | one xbar/direction, one 32-bit flit/cycle/dir. @ 700 MHz (175 GB/s/dir.); 8-flit VCs (5 for MESI, 2 otherwise) |
| DRAM | 1400 MHz, GDDR, 8 bytes/cycle (175 GB/s peak), 460-cycle minimum latency, FR-FCFS queues, $t_{CL}$=12, $t_{RP}$=12, $t_{RC}$=40, $t_{RAS}$=28, $t_{CCD}$=2, $t_{WL}$=4, $t_{RCD}$=12, $t_{RRD}$=6, $t_{CDLR}$=5, $t_{WR}$=12, $t_{CCDL}$=3, $t_{WR}$=2 |
| lease times | 32 bits, predicted from 8–16–$\cdots$–1024–2048 |

**Table 3.3:** Simulated GPU and memory hierarchy for RCC.

### 3.4.6 RCC-WO: A Weakly Ordered Variant

Relative load and store ordering is effected through the per-core logical time *now*. Keeping track of two separate logical *now* times — the read view, consulted and updated by load operations, and the write view, consulted and updated by store operations — allows loads and stores to be reordered with respect to each other. In this scheme, full fence operations require only that the read view and write view *now* values be set to whichever is larger; performance can potentially improve because stores no longer expire cache data that do not have the same block address. The consistency model is WO [68]; work concurrent with ours [257] proposes a similar adaptation that supports RCsc [78].

## 3.5 Methodology

**Simulation Setup.** We follow the methodology used in previous GPU coherence work [220, 221]. GPGPU-Sim 3.x [31] is used to simulate the core, and combined with the Ruby memory hierarchy simulator from gem5 [34] to execute coherence transactions. For the sequentially consistent implementations (MESI, TCS, RCC), we altered the shader core model to execute global memory instructions sequentially,

| | | |
|---|---|---|
| inter-workgroup communication | | |
| BFS | breadth-first-search | graph traversal [31] |
| BH | Barnes-Hut | n-body simulation kernel [42] |
| CL | RopaDemo | cloth physics kernel [40] |
| DLB | dynamic load balancing | work stealing algorithm for octree partitioning [47] |
| STN | stencil | finite difference solver synchronized using fast barriers [243] |
| VPR | place & route | FPGA synthesis tool [199] |
| intra-workgroup communication | | |
| HSP | hotspot | 2D thermal simulation kernel [53] |
| KMN | k-means | iterative clustering algorithm [53] |
| LPS | Laplace solver | 3D Laplace Solver [31] |
| NDL | Needleman-Wunsch | DNA sequence alignment [53] |
| SR | anisotropic diffusion | speckle reduction for ultrasound images [53] |
| LUD | matrix LU | matrix LU decomposition [53] |

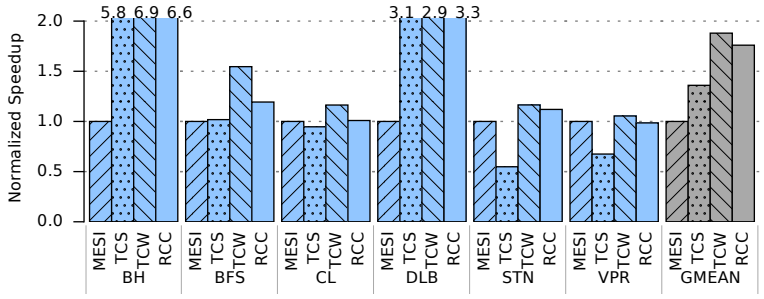
**Table 3.4:** Benchmarks used for RCC evaluation.

and stall local memory operations if there are outstanding global accesses; this matches the "naïve SC" baseline of [220]. We use Garnet [13] to simulate the NoC and ORION 2.0 [109] to estimate interconnect energy.

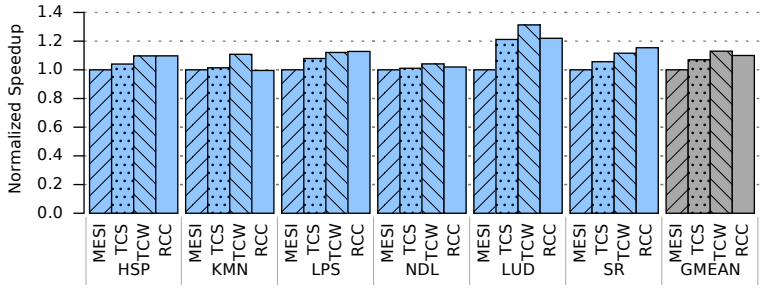
The simulated configuration is similar to NVIDIA's GTX 480 (Fermi [165]), with latencies derived from microbenchmark studies [241]; this matches the configurations used in prior work [220, 221]. Table 3.3 describes the details.

**Benchmarks.** We use benchmarks identified and classified into inter- and intra-workgroup communication categories in prior work on GPU coherence [221]. The intra-workgroup benchmarks execute correctly without coherence, but are used to quantify the impact of always-on cache coherence on traditional GPU workloads. For non-SC simulations, the inter-workgroup communication benchmarks rely on fences; for SC simulations fences act as no-ops in hardware, but were left in the sources to prevent the compiler from reordering operations.

Benchmark details and sources are listed in Table 3.4. Most were used in prior work on GPU coherence [221]; we dropped two because our sensitivity studies found them to be highly nondeterministic and unpredictably sensitive to small changes in architectural parameters (e.g., a few cycles' change in L2 latency). We added missing fences to DLB following [16], and altered tile dimensions in HSP to match GPU cache block sizes and avoid severe false sharing problems.




**(a)** inter-workgroup communication




**(b)** intra-workgroup communication

**Figure 3.8:** Speedup of RCC on inter- and intra-workgroup workloads.

## 3.6 Evaluation Results

### 3.6.1 Performance Analysis

SC on top of RCC performs substantially better than prior SC proposals for GPUs. Figure 3.8 shows that RCC is 76% faster than MESI and 29% faster than TCS on workloads with inter-workgroup sharing; in fact, performance is within 7% of TCW, the best prior non-SC proposal. On benchmarks with intra-workgroup communication patterns, RCC is 10% better than MESI and within 3% of both TCS and TCW.

RCC significantly reduces SC overheads compared to prior SC implementations for GPUs. Figure 3.9 (top) shows issue stall rates caused by enforcing SC: either direct SC memory ordering stalls or LSU pipeline stalls caused by waiting on store acknowledgements. RCC reduces these by 52% relative to MESI (largely because there are no invalidate delays) and by 25% relative to TCS (largely because stores in

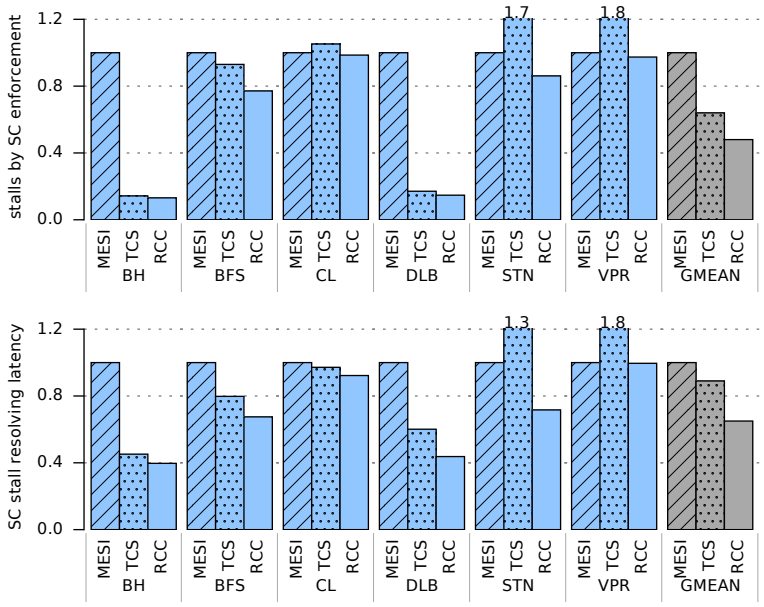
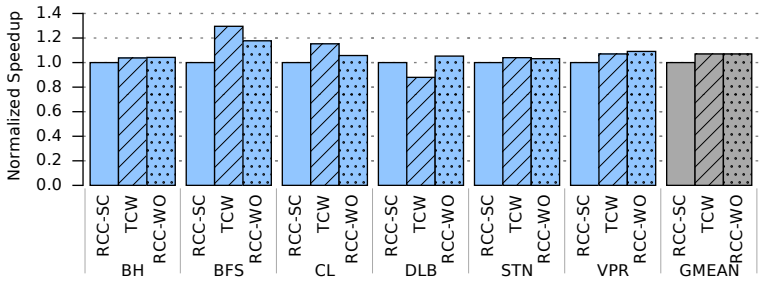



**Figure 3.9:** The reductions of SC stalls (Top) and SC stall resolving latency (Bottom) in RCC (results are normalized to MESI).

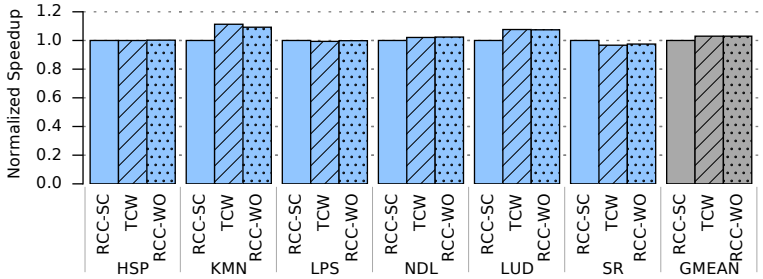
RCC acquire write permissions without stalling). Figure 3.9 (bottom) shows that SC ordering stalls in RCC are resolved 35% faster than in MESI and 11% faster relative to TCS. Both of these metrics directly correlate to performance.

TCW performs better than RCC for BFS because it benefits both from its weak memory model and from relaxing write atomicity. All threads share a "mask" vector, which identifies nodes to be visited in the next iteration (next level of the BFS tree); TCW allows different cores to modify parts of this vector without other cores observing the result, while RCC strictly enforces SC on cache block granularity and sees more L1 misses (73% vs. 52%).

Conversely, RCC outperforms TCW on DLB. In DLB, a per-workgroup work scheduler that completes its task steals tasks from a random other workgroup's scheduler. Since work could be stolen at any time, all per-workgroup queue accesses must be protected with fences; fences stall in TCW until a physical time when all stores have become globally visible. In actuality, however, work stealing events are rare, so most of these stalls are unnecessary. RCC allows cores to progress independently in their own epochs until actual sharing occurs. In addition, stores do




**(a)** inter-workgroup communication



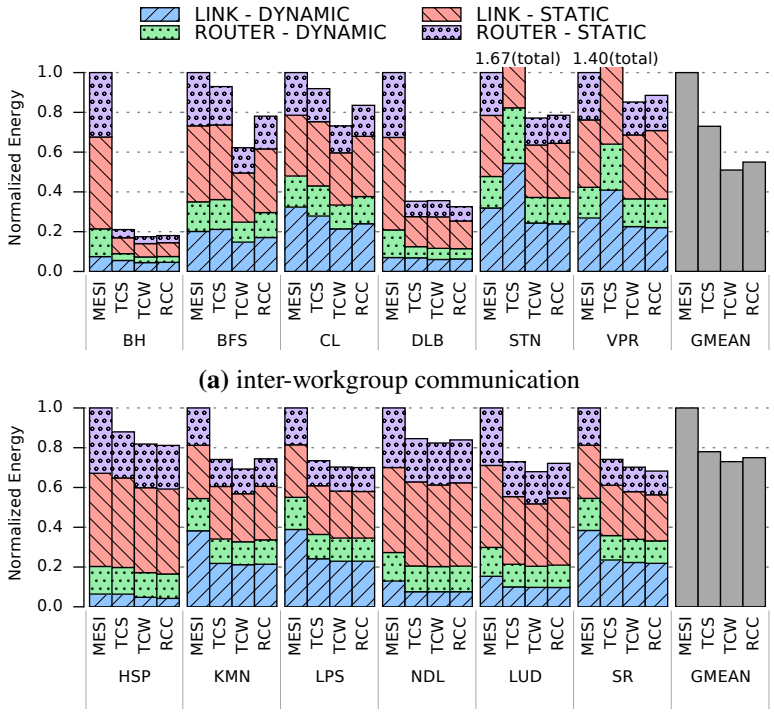

**(b)** intra-workgroup communication

**Figure 3.10:** Speedup of weak ordering implementations vs. RCC-SC on inter- and intra-workgroup workloads.

not stall even when sharing does occur because SC is enforced in logical time.

We also developed RCC-WO, a weakly ordered variant of RCC (Chapter 3.4.6) and compared it with both TCW (our implementation supports WO) and the default SC implementation of RCC. RCC-WO performs neck-to-neck with TCW, and both perform 7% better than RCC-SC (Figure 3.10).

One RCC implementation can support strong and weak consistency. The microarchitectural differences between weak and strong variants of RCC in GPUs consist of one additional scheduler signal per warp to order memops from one thread, and a small change in how stores update L2 metadata. This opens the possibility that the hardware memory model in GPUs could be chosen at boot time (as in, e.g., SPARCv9 [225]) or even at runtime.




**(a)** inter-workgroup communication

**(b)** intra-workgroup communication

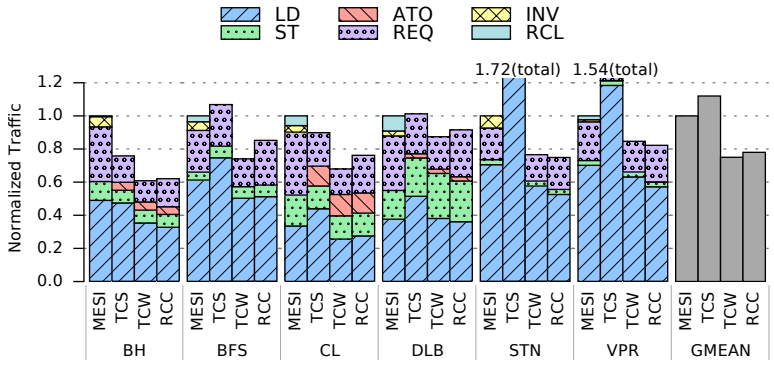
**Figure 3.11:** Energy cost of RCC on inter- and intra-workgroup workloads.

### 3.6.2 Energy Cost and Traffic Load

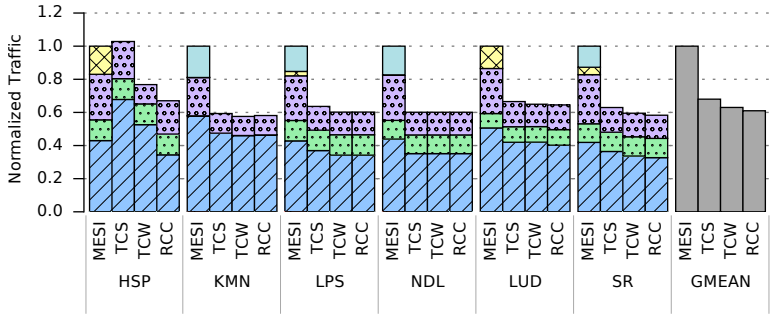
Interconnect energy is 45% lower than MESI, 25% lower than TCS, and only 7% below TCW on inter-workgroup workloads (Figure 3.11); on intra-workgroup programs, it is 25% better than MESI and on par with TCS/TCW. This is partly due to reductions in traffic (Figure 3.12) and partly due to RCC needing only two virtual networks to maintain deadlock-free operations vs. five for MESI. Interconnect energy expenditure is becoming more important as GPU core counts grow.

### 3.6.3 Coherence Protocol Complexity

RCC has fewer states than TCW and TCS (Table 3.5). This is important because coherence is notoriously difficult to verify: usually, validation involves very simplified formal models and extensive simulations [32, 242], but bugs survive despite extensive validation efforts [41, 61, 67, 185].




**(a)** inter-workgroup communication




**(b)** intra-workgroup communication

**Figure 3.12:** Traffic load of RCC on inter- and intra-workgroup workloads.

| | MESI | TCS | TCW | **RCC** |
|---|---|---|---|---|
| L1 states | 16 (5+11) | 5 (2+3) | 5 (2+3) | **5 (2+3)** |
| L1 transitions | 81 | 27 | 42 | **33** |
| L2 states | 15 (4+11) | 8 (4+4) | 8 (4+4) | **4 (2+2)** |
| L2 transitions | 50 | 23 | 34 | **14** |

**Table 3.5:** The number of states (stable+transient) and transitions for different coherence protocols.

### 3.6.4 Area Cost

RCC has reasonable silicon area overheads. For every L1 block, RCC only stores *exp*, and, for every L2 block, *exp* and *ver*. GPU cache blocks are 128 bytes, with perhaps 3-byte tags; with 32-bit timestamps this is 3% overhead for L1 and 6% area overhead for L2.

## 3.7 Summary

In this chapter we track the source of SC inefficiency in GPUs to long store latencies caused by coherence traffic; these severely exacerbate SC ordering and structural bottlenecks that GPUs could otherwise easily amortize. We address these by proposing RCC, a coherence protocol that uses logical timestamps to reduce store latency. When used as part of an SC implementation, RCC reduces SC-related stalls by 25%, and stall resolve latency by 11%, compared to the best coherence proposal for GPUs capable of supporting SC; as a result, performance is 29% better.

When used in RC mode, RCC matches the best prior RC proposal; because the hardware needed for RCC is similar for SC and RC, a single implementation can potentially allow runtime selection of the desired memory consistency model.

# Chapter 4

# Hardware Transactional Memory with Eager Conflict Detection

This chapter explores a simple and reliable programming model to implement efficient synchronization in GPUs. Instead of the lock mechanism, we choose Transactional Memory (TM) for simple and deadlock-free programming. We identify the excessive latency of value-based lazy conflict detection mechanism is critical performance bottleneck of prior GPU TM designs, so we propose GETM, the first GPU hardware TM with eager conflict detection. GETM relies on a logical-timestamp-based conflict detection mechanism: by comparing the timestamp of transaction with the timestamp of accessed data in memory, conflicts are detected eagerly when the initial memory access is made. Performance of GETM is up to 2.1× (1.2× gmean) better than the best prior work. Area overheads are 3.6× lower and power overheads are 2.2× lower.

While GPUs have traditionally focused on streaming applications with regular parallelism, irregular GPU applications with fine-grained synchronization are becoming increasingly important. Graph transformation [140, 249], dynamic programming [129], parallel data structures [150], and distributed hashtables [97] have all been accelerated on GPUs using fine-grained locks. Fine-grained parallel algorithms have recently become a hardware optimization focus for commercial GPUs [69].

Unfortunately, high-performance parallel applications with fine-grained locks

```
if (src > dst) { // acquire in-order to avoid deadlock
    outer = src; inner = dst;
} else {
    inner = src; outer = dst;
}
done = false;
while (!done) { // loop on flag to avoid SIMT deadlock
    if (atomicCAS(&locks[outer], 0, 1) == 0) {
        if (atomicCAS(&locks[inner], 0, 1) == 0) {
            accounts[src] -= amount;
            accounts[dst] += amount;
            locks[inner] = 0; // release
            locks[outer] = 0; // both locks
            done = true;
        } else { // acquired outer but not inner lock
            locks[outer] = 0; // release outer lock
        }
    }
}
```

```
txbegin
    accounts[src] -= amount;
    accounts[dst] += amount;
txcommit
```

**Figure 4.1:** CUDA ATM benchmark fragment using either locks or TM.

are challenging to program and debug. Indeed, reasoning about thread-based synchronized programs is difficult in general [28, 125], and even simple formal analyses that account for inter-thread synchronization are NP-hard [231] or undecidable [193]. In practice, the problem is exacerbated in accelerators like GPUs, because optimizing for performance is paramount — after all, if it weren't, the code would be running on a CPU. In GPUs, this problem is even worse, as the combination of lockstep warp execution and stack-based branch reconvergence can result in unexpected deadlocks in code that would be deadlock-free in CPUs [72].

Transactional memory (TM) [96, 228] offers an attractive solution. In contrast to the imperative style and global dependencies induced by locks, transactions enable a *declarative* programming style: the programmer specifies that a given code block constitutes an atomic transaction and leaves execution details to the runtime (see Figure 4.1). Typically, the runtime (hardware or software) attempts to execute transactions optimistically, only aborting and retrying them when conflicts are detected; writes performed by aborted transactions are not visible to transactions that

commit successfully. Because they maintain atomicity and isolation, transactions are *composable* [93], and substantially simplify code in complex codebases [200, 259], leading to many times lower error rates [201]. Recently, hardware-level transactional memory has appeared in production CPUs from major vendors [44, 90, 106, 107], as well as in designs and proposals from other significant industry players [52, 60].

Early proposals for hardware-level transactional memory for GPUs solved key problems of interacting with the SIMT stack [77] and coalescing transactions at warp level [76]. Both rely on value-based validation, which requires one core $\leftrightarrow$ LLC round trip to validate each transaction and another round-trip to finalize the commit. Combined with the massive concurrency present in GPU workloads, these long latencies create bottlenecks in the commit phase: even if transactional concurrency is restricted, 700 or more transactions may be queued in the commit phase on average [77].

Prior proposals have therefore limited transactional concurrency to very few warps per SIMT core [76, 77]. With few warps, however, the GPU can no longer effectively amortize commit latencies, so some performance is lost. Another proposal has been to proactively abort transactions by broadcasting conflict sets from the LLC back to the SIMT cores [55]; the bandwidth and latency of these broadcasts, however, limit this approach to extremely long transactions.

In this chapter, we instead propose to directly reduce commit costs by detecting conflicts eagerly. If conflict detection is performed separately for each memory access — a latency well within a GPU's capacity to amortize even with concurrency throttling — a transaction that arrives at the commit point is guaranteed to commit successfully. Because there is no need for time-consuming value-based conflict detection at commit time, the commit itself can be taken off the critical path while the warp continues execution. To the best of our knowledge, this is the first full GPU hardware TM proposal with eager conflict detection, and the first to leave transaction commits out of the critical path.

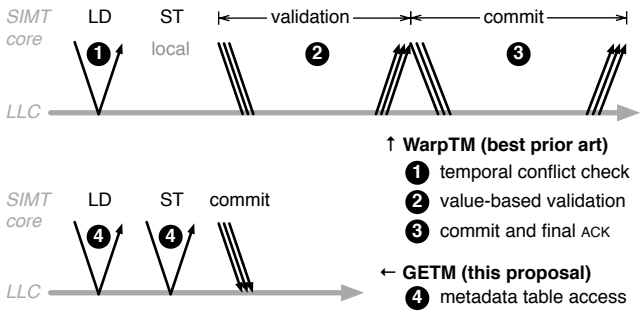



**Figure 4.2:** Messages required for transactional memory accesses and commits in WarpTM (top) and GETM (bottom).

## 4.1 GPU Transactional Memory

The state-of-the-art GPU TM, WarpTM [76, 77], combines lazy version management with lazy, value-based conflict detection.[1] Figure 4.2 (top) shows the access and commit timing.

Firstly, WarpTM modifies the SIMT stacks to allow aborting and restarting transactions at thread granularity. GPUs execute many (32–64) threads in lockstep as a single warp; transactions are a thread-level abstraction, however, so it is possible that some of the threads in the warp commit while other threads abort. WarpTM adds special Transaction and Retry stack entry types that track which threads aborted and should run again when the transaction is restarted.

As transactions execute, their memory accesses are sent to a redo log, stored in the SIMT core's local memory.[2] For each address, loads record the value that was observed (for later validation), and stores record the newly written value. When the warp reaches `txcommit`, a tx log unit traverses the redo log to record all threads wishing to access each address; this allows the SIMT core to resolve all intra-warp conflicts and coalesce the warp's surviving transactions.

At commit time, the read and write logs of the coalesced transaction are sent to validation/commit units (VUs/CUs) colocated with each LLC bank. Each validation unit verifies that the value observed by each read in the log corresponds to the

[1] We discuss other GPU proposals [55, 56, 234] in Chapter 7.3.

[2] In NVIDIA terminology, a GPU core's local memory is an address range of the global address space reserved for that core. As with the rest of the address space, local memory is cached in the GPU cache hierarchy.

current value in the LLC, and sends a success/failure message to the SIMT core. The core collects these to check whether any addresses failed validation, and sends a commit/abort confirmation back to the CUs. Each CU then sends the write log values to the LLC, and ACKs to the core. Once the core has collected ACKs from all CUs, the warp continues execution. Transactional consistency requires each transaction to be validated and committed atomically, so while one transaction goes through the two-round-trip validation/commit sequence, other transactions must wait.

WarpTM also includes a temporal conflict check mechanism (TCD) that allows read-only transactions to commit silently. A TCD table at the LLC that records the physical clock cycle number of the last store to each address; the cycle numbers are updated as transactions commit. Each transactional load is immediately sent from the SIMT core to this TCD table; if a read-only transaction has only read locations modified in the past, it is allowed to bypass value-based validation and commit silently.

Because GETM uses eager conflict detection, transactions that have reached `txcommit` are guaranteed to be free of conflicts, and commit without additional validation or ACKs.

GETM retains the SIMT stack modifications and warp-level transaction coalescing of WarpTM. However, it replaces the value-based validation and TCD read-only silent commits with an eager conflict detection scheme (see Chapter 4.4), which greatly simplifies the validation/commit unit and substantially reduces the hardware overhead (see Chapter 4.5 and Chapter 4.7).

## 4.2 Eager Conflict Detection and GPUs

Although eager conflict detection is more suitable for high-thread-count architectures (see Chapter 4.3), the lack of a natural conflict detection mechanism poses a challenge to implementing eager conflict detection in GPUs. Prior TMs with eager conflict detection (e.g., LogTM [149, 252]) have targeted CPUs, in which conflicts are naturally flagged when cache lines are invalidated by the coherence protocol. Unfortunately, extant GPUs lack hardware cache coherence, so another mechanism must be designed. Another challenge is scalability, since GPUs have large core

counts and many concurrent warps in each core. This precludes, for example, mechanisms that collect and broadcast read/write signatures.

To provide a scalable eager conflict detection mechanism, we take inspiration from the software transactional memory system TL2 [65]. TL2 uses a global version-clock that is incremented by every transaction which writes to memory, and maintains last-written version-clock values for every memory location. As the transaction accesses memory, it collects version-clocks for all referenced locations. At commit time, these clocks are checked to ensure that the transaction observed a consistent state of memory; if there are no violations, TL2 acquires locks for all locations it intends to modify and finally writes the memory.

In TL2, logical clocks are used to ensure consistency, but conflict detection is still performed lazily at commit time. In addition, the global version clock must be shared among multiple cores, which relies on the underlying cache coherence protocol. We leverage the idea of providing consistency via logical clocks, but use them to implement early conflict detection, and design a distributed logical clock protocol that does not need cache coherence.

We propose GPU Eager Transactional Memory (GETM), a novel GPU hardware TM design. Unlike prior eager TMs, GETM does not rely on coherence or signature broadcast. Instead, GETM combines encounter-time write reservations with a logical timestamp mechanism to detect conflicts as soon as they occur, and to allow off-critical-path commits.

## 4.3 GPUs Favour Eager Conflict Detection

In this section, we argue that eager conflict detection is particularly suited to the large number of threads concurrently executing in a GPU, because the long commit latencies inherent in lazy detection form a key bottleneck as concurrency grows. This is not the case for CPUs, where TMs with eager conflict detection, such as LogTM [149], are outperformed by lazy [51] or partially lazy [233] variants.

To test this intuition, we modified the state-of-the-art GPU TM design WarpTM [76] to emulate eager conflict detection (cf. Figure 4.2) and examined how it performs as the number of warps per SIMT core grows. WarpTM uses lazy conflict detection and lazy versioning (see Chapter 2.2.2 for details), and commits transactions via

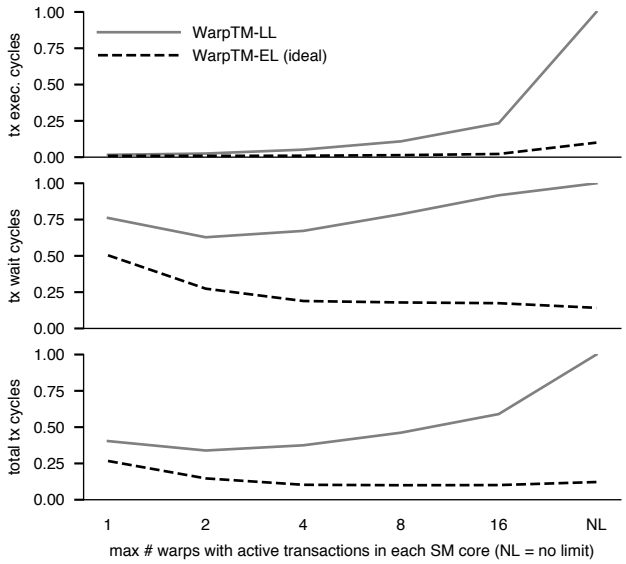
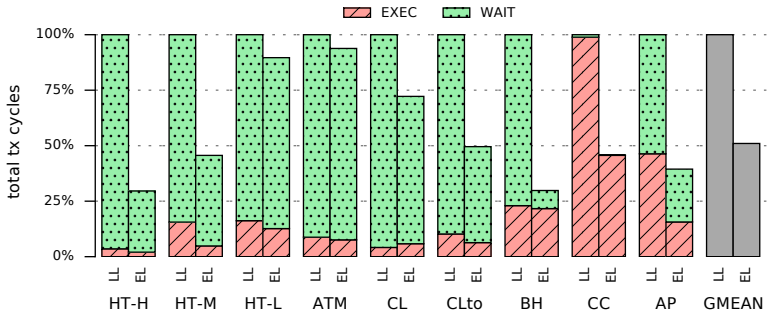


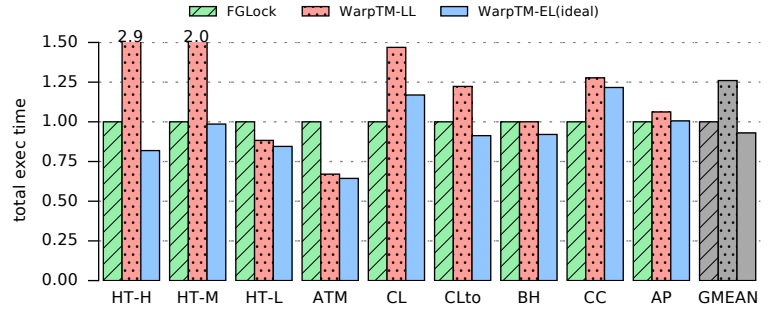

**Figure 4.3:** Time per transaction spent executing transactional code (top), waiting for aborting transactions in the same warp and concurrency limits (centre), and total time spent in transactions (bottom), as the number of warps allowed to concurrently run transactions grows. Measurements from the HT-H hashtable benchmark, normalized to the highest data point.

two core $\leftrightarrow$ LLC round trips: (i) the transaction log is sent to be value-validated at the LLC banks; (ii) the LLC sends back validation success/failure status; (iii) the core collects the responses and (if all banks reported success) instructs the LLC to start commit; (iv) the LLC banks acknowledge commit completion; (v) the core can resume executing the relevant warp. Eager conflict detection needs to check only the currently accessed memory location, but must be repeated for every access; therefore, to emulate an eager-lazy design, we hacked WarpTM to run validation (i)–(ii) for every transactional access, with no latency.

Figure 4.3 (top) shows how the original WarpTM (-LL) and idealized eager-lazy variant (-EL) perform as permitted concurrency grows on the hashtable insertion workload HT-H. With an increasing number of transactions, the number of cycles spent executing each transaction (including retries) grows much faster for the variant with lazy conflict detection than for the eager version. This is because increasing




**(a)** cycle breakdown of transactional segments only




**(b)** total execution cycles of whole application (tx and non-tx segments)

**Figure 4.4:** Benefits of eager conflict detection compared with lazy mechanism and hand-optimized find-grained lock implementations. (Optimal concurrency is used for all configurations)

concurrency increases conflicts and causes transactions to be retried more times. For each retry, WarpTM-LL incurs the two round-trip latency of lazy value-based validation, making each attempt far more expensive than in WarpTM-EL.

Figure 4.3 (centre) shows how long transactions wait to commit, either because of concurrency throttling or because of waiting for diverged threads in the same warp to abort the transaction. Because the value-based validations in WarpTM-LL are expensive, subsequent transactions wait longer than in WarpTM-EL. For WarpTM-EL, wait time decreases as more warps can execute and cover commit latency; for WarpTM-LL, however, increasing concurrency increases the commit queue backup and therefore the total wait cost.

The overall runtime spent in transactions is shown in Figure 4.3 (bottom). This explains why the optimal concurrency for WarpTM-LL is 2 transactional warps

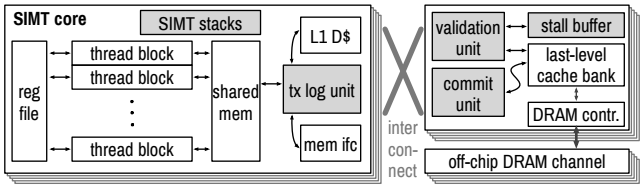



**Figure 4.5:** Overall architecture of a SIMT core with GETM. Shaded blocks are added for transactional memory support.

per SIMT core [76], and demonstrates that eager conflict detection can support substantially more concurrency with much lower overheads.

Note that this effect is peculiar to architectures with high thread-level concurrency, such as GPUs. Most CPUs run 1–2 threads per core, and have few cores per die. This places them on the left of Figure 4.3 (top), where the lazy and eager versions execute similar number of transactional cycles.

To quantify the overall performance potential of eager conflict detection, we simulated a range of TM benchmarks using the lazy and eager variants of WarpTM, as well as the equivalent non-TM versions using hand-optimized fine-grained locks. Figure 4.4 (top) shows that execution and wait cycles spent in transactions are substantially reduced in the eager variant, and Figure 4.4 (bottom) shows that this translates to faster overall execution time.

## 4.4 GETM Transactional Memory

In this section, we sketch an overview of how GETM provides transactional atomicity, consistency, and isolation, and describe how it tracks the necessary metadata.

The description here focuses on the GETM protocol, how transactions execute, and how metadata evolves. The high-level architecture is shown in Figure 4.5; implementation details, including the metadata and queueing data structures present at the LLC, are described in Chapter 4.5.

### 4.4.1 Atomicity, Consistency, and Isolation

We first describe the transaction logs that provide atomicity, and then the logical timestamp and access-time locking mechanisms used to ensure consistency and isolation.

| Tracked per warp | |
|---|---|
| *warpts* | the logical time at which transactions from this warp atomically execute |
| **Tracked per LLC cache line** | |
| *wts* | one higher than the logical time when this location was last written |
| *rts* | the logical time when this location was last read |
| *#writes* | # writes to this location (if non-zero, location is locked by a transaction) |
| *owner* | the owner of the write reservation (if *# writes* is non-zero) |

**Table 4.1:** Metadata tracked by GETM.

**Transaction logs.** As in prior work [76, 77], transactions are managed at warp level, and each warp keeps a redo log in the SIMT core's existing local memory.

In contrast to GETM, prior work required sending the entire log (reads and writes) to the commit units for validation at commit time. Because GETM uses eager conflict detection, transactions that have reached `txcommit` are guaranteed to succeed, and commit-time validation is not necessary. Instead, a committing transaction transmits only the transactional writes from the redo log (typically a fraction of the entire log), so that the write data can be stored in the LLC.

In addition to being logged, all transactional accesses are sent to the LLC for eager conflict detection, using the timestamp and lock mechanisms described below.

**Logical timestamps.** GETM uses distributed logical timestamps to provide transactional consistency, and each transaction executes at a specific logical timestamp. To guarantee consistency, GETM must ensure that a running transaction (a) does not observe stale values of locations changed by logically earlier transactions, (b) does not observe values written by logically later transactions, and (c) does not alter values already seen by logically later transactions.

The logical timestamps tracked by GETM are shown in Table 4.1. Firstly, each warp keeps a logical timestamp *warpts*, corresponding to the memory state observed by the last transaction. This timestamp starts at 0, and is advanced when transactions abort (as discussed below). All new transactions started by this warp execute at logical time *warpts*.

Each cache line in the shared LLC has a write timestamp *wts*, equal to one more than the logical time of the last write, i.e., 1 *warpts* of the logically latest

transaction to attempt a write. If a transaction $T$ attempts to access a cache line $L$ where $L.wts > T.warpts$, it means that $L$ was written by a transaction logically later than $T$, and $T$ must abort.

Every cache line also contains a read timestamp *rts*, equal to the logical time of the last read, i.e., *warpts* of the last transaction to read it. A transaction $T$ may read lines with any *rts*, but writing a cache line $L$ where $L.rts > T.warpts$ would overwrite a value which has already been observed by a later transaction, and is not permitted.

The *rts* and *wts* timestamps are maintained eagerly: that is, transactional loads update *rts* and transactional writes update *wts* at the time of the request, regardless of whether the transaction will eventually commit. The updated timestamps are not reverted if a transaction aborts; while this might unnecessarily abort some future transactions, those will be restarted, and consistency is not compromised.

**Encounter-time locks.** Unlike timestamps, transactional write data is not stored in the LLC until the transaction reaches its commit point. This creates an isolation problem if a transaction $T_1$ modifies a location and a logically later but physically concurrent transaction $T_2$ accesses this location: the value that should be seen by $T_2$ depends on whether $T_1$ will commit successfully, but $T_1$ is still in progress.

To avoid this issue, GETM uses locks to prevent $T_2$ from reading the location until $T_1$ has committed. Each cache line has two additional fields to support this: *#writes* and *owner* (see Table 4.1). When a transaction $T$ first encounters a previously untouched cache line $L$, it reserves $L$ by setting $L.\#writes$ to 1 and $L.owner$ to the transaction's global warp ID (because transactions are coalesced per warp, this uniquely identifies a running transaction; see Chapter 4.1).

Now when $T_2$ accesses $L$ (either for reading or writing), it must check whether $L$ has been reserved. If $L.\#writes \neq 0$ and $L.owner \neq T_2$, transaction $T_2$ proceeds with the *rts/wts* checks described above; if the checks fail then $T_2$ is aborted, otherwise $T_2$ stalls until $T_1$ commits. (We discuss the stall buffer where stalled transactions are queued in Chapter 4.5.)

The *owner/#writes* mechanism also allows a transaction to repeatedly write the same location. If $T$ is already the *owner* of a cache line, it bypasses the *rts* and *wts* timestamp checks, and writes the line. This is safe because $T$ must have previously satisfied the *rts* and *wts* timestamp constraints, and updated *wts*. As any other

transaction attempting to update the line since that time would have been stalled, neither *rts* and *wts* could have been altered since $T$'s reservation.

**Aborts and advancing logical time.** The logical time observed by each warp (*warpts*) advances when transactions are aborted. When a transaction aborts, it reports to the core the latest logical timestamp $t$ it attempted to read or write (the abort cause). Since the transaction will fail again unless it restarts at a time later than $t$, *warpts* is set to $t$ 1.

For example, if a transaction $T$ has aborted because of reading a cache line $L$, it must be because the cache line is logically newer than the transaction, i.e., $L.wts > warpts$. In this case, the SIMT core sets *warpts* to $L.wts$ 1, and $T$ is restarted. Similarly, if $T$ aborts because of a write, *warpts* is set to max$L.rts, L.wts$ 1, and the transaction restarts.

**Commit and cleanup.** When all threads in a warp reach the end of the transaction (commit or abort), the SIMT core serializes the write logs of all threads and sends them to the LLC. For all threads that have successfully reached the commit point, the core sends the address, write data, and write count (since multiple writes may have been coalesced).

Once this commit/abort log is received, each entry is written to the LLC and the relevant *#writes* entry is decremented. Once *#writes* in a cache line has reached 0, the cache line fully reflects the atomic transaction update, and can now be accessed by other transactions.

Aborted transactions instead send the address and write count for each modified cache block to facilitate cleanup. The *#writes* in each cache line is updated as above; after *#writes* has reached 0, the cache line reflects its pre-transaction state, and may be accessed by other transactions.

**The life of a transactional access.** Figure 4.6 shows how a transactional read or write is processed in GETM.

*Owner check* ❶. If *#writes* is non-zero but the owner field matches the current transaction, the line must be locked and the access succeeds ❷. Stores only increment *#writes* (since *wts* was already set by the previous write), while loads potentially update *rts* if it is less than *warpts*.

*Timestamp check* ❸. A transaction that attempts to load an address and finds its *wts* younger than the transaction's own *warpts* has detected a WAR conflict – i.e.,

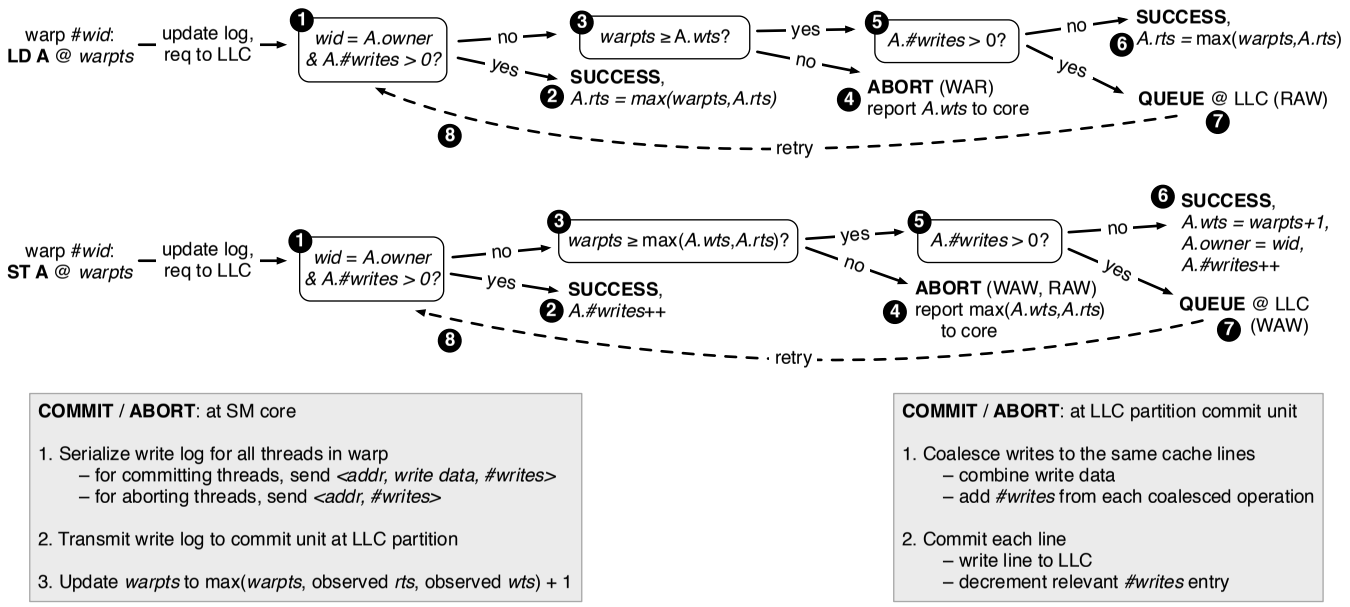



**Figure 4.6:** The flowchart for load, store, and commit/abort logic in GETM.

another transaction with a younger *warpts* has already written to the location – and must abort ❹. Similarly, a transaction that writes a location but finds either *wts* or *rts* to be younger than *warpts* must also abort, since a logically younger transaction has either written the location or observed its value ❹.

*Abort notification* ❹. If the version check discovers a conflict, the transaction must be aborted. To minimize the chances of the transaction aborting again, the SIMT core is sent the highest timestamp seen so far at the LLC; this will be used to update *warpts* and restart the transaction. Meanwhile, the core notes that the thread has aborted, and will clean up any reservations made when the entire warp reaches *txcommit* or when all threads have aborted.

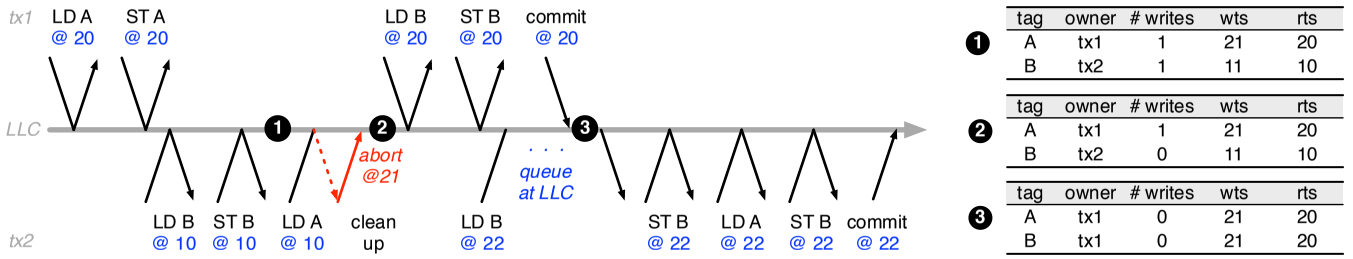
*Write lock check* ❺. Next, the transactional memory operation checks whether the accessed location has been reserved by another warp (i.e., whether *#writes* is non-zero). If not, the operation succeeds without conflict: a load will update *rts* (if $< warpts$) while a store will set *#writes* to 1 and update the location's *wts* with the transaction's *warpts* ❻.

*Queue* ❼ *and retry* ❽. Accesses that passed the timestamp check but do not own the active lock must be logically younger than the lock owner. To avoid unnecessary aborts, these requests are queued until the owner transaction commits. After the lock is released, the queued transactions will retry.

### 4.4.2 Walkthrough Example

Figure 4.7 illustrates how the GETM protocol operates on two conflicting transactions from the bank transfer example (Figure 4.1); in this benchmark, accounts are modelled as unique memory locations. The first transaction (*tx1*) transfers some amount from account A to account B, while the second (*tx2*) transfers another amount from B to A. Transaction *tx1* starts at *warpts* = 20, and transaction *tx2* starts at *warpts* = 10. The central grey line represents the LLC, and the thinner black arrows represent messages between the cores and the LLC. The interleaving of the accesses from each transaction has been chosen to illustrate how the eager conflict detection and queueing mechanisms work; in reality, any interleaving of the two transactions could occur.

First, *tx1* loads and stores location A: the load updates A's *rts* to match the




| | tag | owner | # writes | wts | rts |
|---|---|---|---|---|---|
| 1 | A | tx1 | 1 | 21 | 20 |
| | B | tx2 | 1 | 11 | 10 |

| | tag | owner | # writes | wts | rts |
|---|---|---|---|---|---|
| 2 | A | tx1 | 1 | 21 | 20 |
| | B | tx2 | 0 | 11 | 10 |

| | tag | owner | # writes | wts | rts |
|---|---|---|---|---|---|
| 3 | A | tx1 | 0 | 21 | 20 |
| | B | tx1 | 0 | 21 | 20 |

**Figure 4.7:** A walkthrough example of eager conflict resolution in GETM.

transaction's *warpts* (i.e., to 20), and the store updates the *wts* of A to exceed that of *tx1* (i.e., to 21). Then *tx2* does the same with B, updating its *wts* to 11 and *rts* to 10. At this point, *tx1* and *tx2* have accessed disjoint locations and so far do not conflict. The transaction metadata for addresses A and B at this point are shown in table ❶.

Next, *tx2* attempts to read location A, previously altered by *tx1*. Because $tx2.warpts < A.wts$, the load fails the version check and will abort *tx2* (cf. Figure 4.6). The LLC will notify the requesting core that the transaction been aborted, and that the next *warpts* should be later than 21. The core will then send the write/abort log for *tx2* to the LLC, which will release the reservation for B by setting the *# writes* field to 0. When *tx1* now sends load and store requests for B, both requests succeed since *tx2* had an older version and its write lock was cleared as *tx2* aborted. At this point, the metadata for A and B correspond to table ❷.

Transaction *tx2* now restarts at the core, with a higher *warpts* of 22. When its first load request (for B) arrives at the validation unit, it passes the version check but finds B reserved; the load is therefore queued in the VU's stall buffer and will be retried as the conflicting transaction commits.

Meanwhile, *tx1* gets to its commit instruction. Because all of its memory accesses have passed eager conflict detection, the transaction is guaranteed to succeed. The core therefore sends the write log to the LLC and moves on. As the write log is processed, write reservations (*#writes*) for both A and B are reset. Table ❸ shows the metadata at this point.

Once the commit of *tx1* has finished and released the reservations on A and B, any stalled transaction accesses are retried; in this case, this is the load of B from *tx2*, which now succeeds. Transaction *tx2* can then continue with its remaining memory accesses, and will succeed.

## 4.5 GETM Implementation Details

Adding transactional memory support requires modifications to both the SIMT core and the memory partition that houses the LLC slice and a memory controller: we need to modify the core to retry aborted transactions and record redo logs, and to add validation and commit hardware to each memory partition. Figure 4.5 shows the overall architecture components of a GPU core extended with GETM.

### 4.5.1 SIMT Core Extensions

*SIMT Stack.* Adding transactional memory support to a GPU's cores requires changing the SIMT stack to track which threads in the warp are executing transactions and which must be retried. To implement this, we leverage the modified SIMT stack proposed by Fung et al [77]. This mechanism is similar to branch divergence hardware [127]: for each warp, the top of the SIMT stack tracks the threads that are currently executing, while the stack entry immediately below tracks threads that have aborted and must be retried.

*Transaction management.* While individual threads can run separate transactions, commits occur at warp granularity when all threads in the warp have arrived at the commit point [76]. Nevertheless, transactions remain logically at thread granularity: when some of the warp's threads abort, they are automatically retried via the extended SIMT stack after the entire warp reaches the commit point [77].

*Transaction logs.* The GETM versioning mechanism is the same as in GPU transactional memory [77]. Logs are stored in each SIMT core's local address space, and cached by the L1/LLC caches. Although GETM only requires a write log, we also record a read log to permit intra-warp conflict detection [76]; in this technique, each transactional access is first checked against the local per-warp read and write logs and aborted if it conflicts with other threads in the same warp. At commit time, however, the read log is discarded and only the write log is sent to the commit units.

*Forward progress.* Aborted transactions ensure progress by restarting with a probabilistically increasing backoff [121].

### 4.5.2 Validation Unit

GETM protocol actions on the LLC side are carried out by Validation Units (VUs), one of which is colocated with each LLC bank. Each VU consists of (a) metadata storage structures to track the last-written and last-read versions for each address, and (b) a structure to buffer requests that found a location locked but were younger than the current owner.

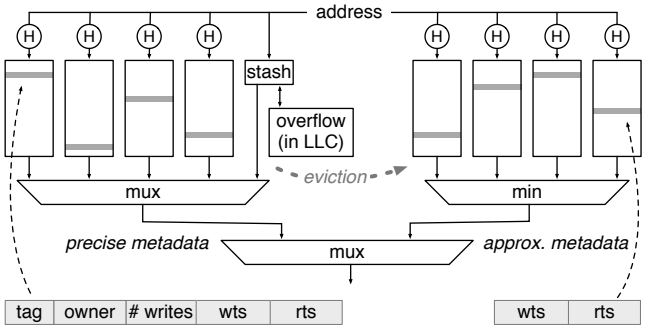



**Figure 4.8:** Transaction metadata table microarchitecture.

### Transaction metadata storage

Because GETM explicitly tracks versions to enable eager conflict detection, it must keep all metadata (*wts*, *rts*, *# writes*, and *owner*; see Table 4.1) for all locations that are part of any in-flight transaction, and some metadata (*wts* and *rts*) for all locations that have been (or could be) accessed transactionally.

These requirements pose some challenges: firstly, transactions could be very long (and, in general, unbounded), so fast access to a potentially large lookup structure is necessary; secondly, potentially all addresses could be accessed transactionally, and tracking metadata for them all is impractical.

Our solution relies on two observations. The first is that very long transactions are likely to be rare in well-tuned code; therefore the metadata table can be sized for the common case and provide a spillover mechanism (like in Unbounded TM [23]). The second is that metadata for addresses that are not being written by in-flight transactions can be maintained *approximately* provided that the only errors are overestimates: if the lookup mechanism reports a higher *rts* or *wts*, additional transactions may abort, but correctness will be preserved.

Figure 4.8 shows the microarchitecture of the metadata storage structure. Our implementation has one such structure at every LLC partition, responsible for the same address range. It consists of two tables, accessed simultaneously during lookups: the first tracks precise metadata for addresses accessed by in-flight transactions, while the second tracks approximate *rts* and *wts* for all other addresses.

*Precise metadata for in-flight accesses.* The precise metadata table is similar to

a cuckoo hash table [182], extended with a small stash [115] (conceptually similar to a victim cache); even a small stash allows the cuckoo table to maintain higher occupancy with limited resources [115]. When inserting a ⟨key, value⟩ pair causes too many swaps in the cuckoo table, the last ⟨key, value⟩ pair swapped out during the insertion process is placed in the stash, and during lookups the stash is searched in parallel with the cuckoo table itself. We use a four-way cuckoo table with four randomly generated $H_3$ hashes [203] and a 4-entry fully associative stash. To permit very long transactions, the precise table and stash can spill to an unbounded overflow space located in main memory and cached in the LLC. In our experiments the overflow space was never used, so we organized the overflow as a linked list; a commercial implementation would likely use an asymptotically faster design such as a balanced tree or another hashtable layer in main memory.

Unlike the original cuckoo table, our design allows the insertion process to terminate by evicting an entry that has not been reserved by any transaction (i.e., *# writes* is zero). Since the remaining metadata — *wts* and *rts* — can be safely approximated, the evicted entry is inserted into the approximate metadata structure described below.

*Approximate metadata for inactive locations.* The simplest design for approximate version tracking is a pair of registers tracking the maximum *wts* and *rts* that have been evicted from the precise table. When a lookup misses in the precise table, it is reinserted using the approximate *wts* and *rts* values from the two registers. When we conducted experiments with this configuration, however, we found that the version numbers increased very quickly and caused many aborts.

To combine efficient storage of large numbers of evicted addresses with the ability to discriminate among many of them, we use a recency Bloom filter [77]. This structure consists of several (in our case, four) ways indexed by different hashes of the lookup address (we again use $H_3$ hashes). Each address maps to one entry in each way, and each entry stores the maximum *wts* and *rts* of all inserted addresses that map to it. On insertion, the *wts* and *rts* in each way are only updated if they exceed the stored values (which may have come from a hash collision), and on lookup the minimum *wts* and *rts* among the four ways are returned.

*Timestamp rollover.* Unlike physical timestamps [76], logical timestamps advance very slowly. In our experiments, the increment rates ranged from one

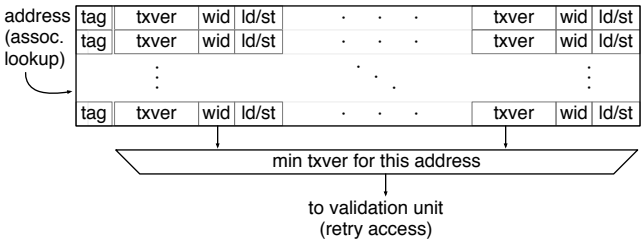



**Figure 4.9:** Stall buffer microarchitecture.

increment in 1,265 cycles to one in 15,836 cycles, depending on the benchmark. At this rate and with a 1 GHz clock, 32-bit timestamps will roll over less than once every 1.5 hours, and 48-bit timestamps will roll over less than once every 11 years.

When a validation unit detects a rollover, it must ensure that (a) all validation units roll over atomically, and (b) all SIMT cores have rolled over. The first task can be accomplished via two messages (containing the VU ID to break ties) sent via a single-wire ring connecting all validation units. The first message indicates that the recipient should stall and forward the message to its neighbour; all VUs will be known to have stalled when the message reaches back to the originating VU. The second message indicates that the recipient should roll over and continue execution. (Alternately, the existing interconnect can be used for this purpose with an ACK–REPLY protocol). Cores roll over on a request from the VUs sent over the interconnect. Once the cores have ACKed the request, the VU knows that no requests are in flight; it flushes the stall buffer and metadata tables and resumes.

### Stall buffer

Requests that passed the version check but found the address locked are queued in a *stall buffer* until the relevant transaction commits or aborts (see Chapter 4.4).

The organization of this structure, shown in Figure 4.9, is similar to a store buffer or an MSHR, but tracks several requests for each address (from different warps contending for the same location). When a committing transaction decrements the *#writes* count to 0, it checks whether any stall buffer entries are waiting on the relevant address; if so, the oldest request (i.e., with the minimum *warpts*) re-enters the validation unit. If the buffer is full, the transaction aborts.

| Baseline GPU | |
|---|---|
| SIMT core config | 15 cores, 48 × 32-wide warps / core, 2 × 16-wide SIMD |
| warp scheduler | greedy then oldest (GTO) |
| in-core storage | 32,768 registers / core, 16KB shared memory / core |
| L1 data cache | 48KB per core, 128-byte lines, 6-way assoc. |
| L2 cache (LLC) | 128KB / partition, 128-byte lines, 8-way assoc., |
| interconnect | 2 xbars (1 up, 1 down), 288GB/s each, 5-cycle latency |
| operating frequency | SIMT core: 1400 MHz, interconnect: 1400 MHz, memory: 924 × 4 (quad-pumped) |
| GDDR5 | 6 partitions, 32 queued requests each, FR-FCFS, Hynix H5GQ1H24AFR timing, total BW 177GB/s |
| memory scheduling latency | L1: 1 cycle; LLC: 330 cycles; DRAM: 200 cycles |
| **Transactional memory support** | |
| concurrency (tx warps/core) | 1, 2, 4, 8, 16, unlimited (optimal for each benchmark) |
| operating frequency | validation unit: 1400 MHz, commit unit: 700 MHz |
| metadata storage | precise: 4K entries (total) in 4-bank cuckoo HTs, 4-entry stashes<br>approx.: 1K entries (total) in 4-bank recency Bloom filters |
| stall buffer | 4 lines with 4 entries each, per partition |
| validation BW | 1 request/cycle per partition |
| commit BW | 32B/cycle per partition |
| intra-warp conflict detection | two-phase parallel, 4KB ownership table / tx warp |

**Table 4.2:** Simulated GPU and memory hierarchy for GETM.

### 4.5.3 Commit-Time Coalescing

The commit unit receives write logs from SIMT cores, coalesces multiple accesses to the same 32-byte regions, writes the data to the LLC, and decrements the relevant *#writes* entries. While coalescing is not needed for correctness, it efficiently uses the GPU's wide LLC port.

To coalesce writes, we use a simplified variant of the ring buffer used in KiloTM [77] and WarpTM [76]. In contrast to these proposals, in GETM the commit unit receives only the write log, so the buffer can be substantially reduced; we conservatively size it to half of that in prior work.

## 4.6 Methodology

**Simulation setup.** We follow the methodology established in previous GPU hardware transaction memory proposals [55, 76, 77]. GPGPU-Sim 3.x [31] is used

| name | abbreviation | description |
|---|---|---|
| Hash Table (CUDA) | HT-H | populate an 8000-entry hash table |
| | HT-M | populate an 80000-entry hash table |
| | HT-L | populate an 800000-entry hash table |
| Bank Account (CUDA) | ATM | parallel funds transfer (1M accounts) |
| Cloth Physics[40] (OpenCL) | CL | cloth physics (60K edges) |
| | CLto | tx-optimized version of CL |
| Barnes Hut [42] (CUDA) | BH | build an octree (30K bodies) |
| CudaCuts [235] (CUDA) | CC | image segmentation (200×150 pixels) |
| Data Mining [110] (CUDA) | AP | data mining (4000 records) |

**Table 4.3:** Benchmarks used for GETM evaluation.

to simulate the GPU and modified to implement GETM and prior proposals. We estimated area and power overheads of the structures required to implement TM by modelling them in CACTI 6.5 [154], conservatively assuming that all structures are accessed every cycle and accounting for the higher validation unit clock. We assumed a 32nm node (the smallest supported by CACTI 6.5).

Table 4.2 describes the simulation setup. For fair comparison of the eager conflict detection mechanism with the value-based detection from prior proposals, we keep the same baseline: a GPGPU similar to NVIDIA's GTX480 (Fermi [165]) with 15 cores, 6 memory partitions, and latencies derived from microbenchmark studies [241]. To investigate scalability to higher core counts, we also simulated a configuration with 56 cores in 28 clusters, and a 4MB L2 cache in eight 8-way banks; for WarpTM, we doubled the recency filter size, and for GETM we doubled only the precise metadata table.

**Baselines.** We compare GETM against WarpTM [76], and an idealized implementation of the EarlyAbort/Pause-n-Go (EAPG) proposal [55].[3] We use TM benchmarks from prior work [76, 77]; they are summarized in Table 4.3.

[3]Specifically, write signatures broadcast to cores were idealized as 64-bit messages, refcount table updates on the LLC side were idealized to one cycle for the entire tx log, and the early conflict check was made instant.

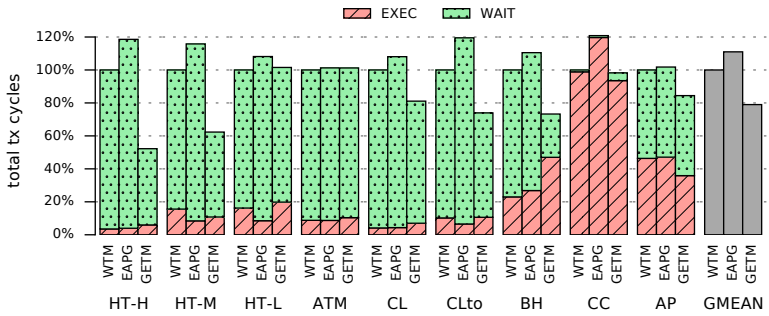



**Figure 4.10:** Transaction-only execution and wait time, normalized to WarpTM baseline (lower is better). Note that EAPG is idealized.

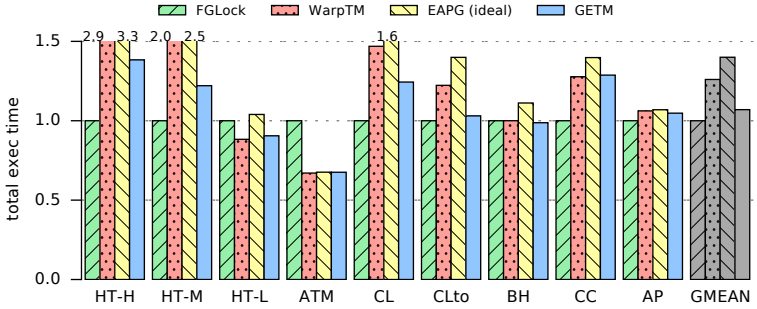



**Figure 4.11:** Program execution time normalized to the fine-grained lock baseline, including transactional and non-transactional parts (lower is better).

## 4.7 Evaluation Results

### 4.7.1 Performance Analysis

Figure 4.10 shows the total number of cycles spent executing transactions and waiting for other transactions to finish, normalized to the WarpTM baseline. For most workloads, GETM reduces both transaction execution time and wait time. CC and AP have contention over few memory locations, and GETM sees many aborts; because commits and aborts are cheap in GETM, however, this is still faster than WarpTM and EAPG. In CC and AP, transactions spend little time waiting because they account for a small portion of the total runtime. We find that, for these benchmarks, even idealized EAPG is ineffective, as only 5.2% aborts come from

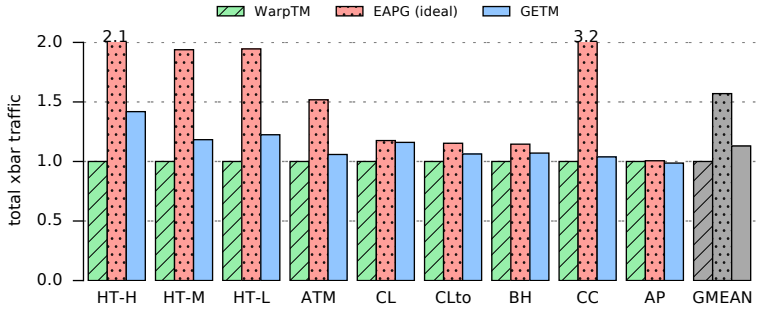



**Figure 4.12:** Crossbar traffic load normalized to WarpTM (lower is better).

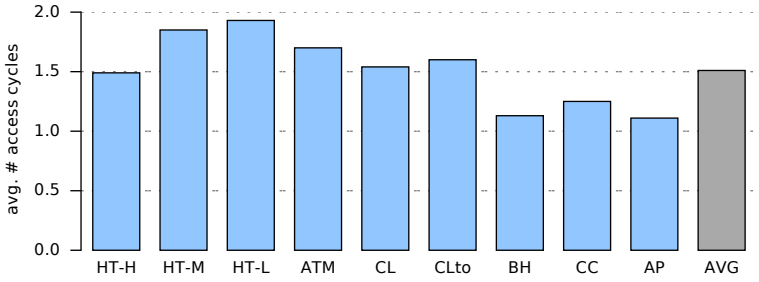
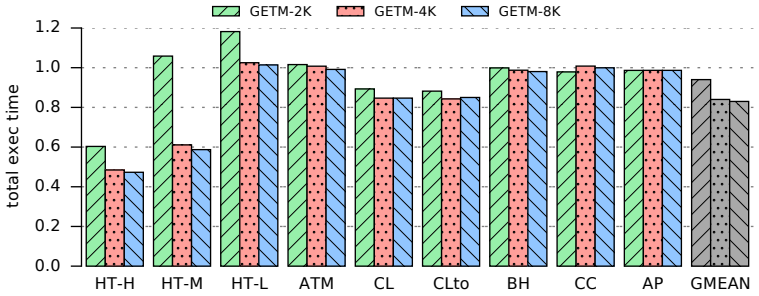


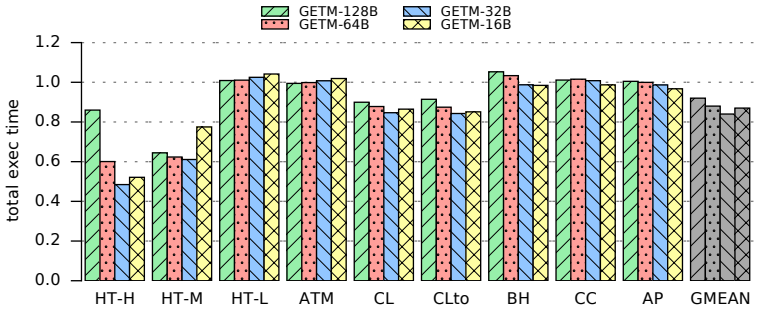

**Figure 4.13:** Mean latency of the cuckoo table in the metadata storage structure ($\geq 1.0$, lower is better).

the early-abort mechanism and 1.3% transactions are ever paused. Essentially, by the time a broadcast update reaches the cores, most conflicting transactions have already been sent for validation/commit. In fact, EAPG underperforms WarpTM because the additional early-abort broadcasts congest the core $\leftrightarrow$ LLC interconnect (even though these are idealized as single header-only flits). We expect that EAPG can be effective only with extremely long transactions.

Overall performance is shown in Figure 4.11: on average, GETM outperforms WarpTM by 1.2$\times$ (gmean) and is within 7% of the fine-grained lock baseline. The trend mirrors that of the transactional execution and wait time above. Benchmarks with high contention benefit more, because GETM aborts doomed transactions without the need to queue at the LLC for value-based validation, and show substantial improvements (up to 2.1$\times$ for HT-H). Low-contention workloads perform comparably to WarpTM.




**(a)** sensitivity to metadata table size




**(b)** sensitivity to metadata tracking granularity

**Figure 4.14:** Performance sensitivity of GETM to metadata table size and tracking granularity, normalized to a WarpTM baseline (lower is better).

The improved performance comes at a minor cost in interconnect traffic compared to WarpTM (Figure 4.12). Although GETM does not need to transmit the transaction read log at commit time, it needs to acquire locks for every write at encounter time, whereas WarpTM only contacts the TCD for loads. In addition, despite better performance, GETM has a higher abort rate, which adds to the interconnect traffic load.

### 4.7.2 Sensitivity Analysis

Because the validation unit contains a cuckoo-like structure where worst-case insertions can take many cycles, we measured the average number of validation unit cycles spent on accessing the metadata tables for each request (Figure 4.13). The combination of allowing evictions to the approximate table and the small stash makes insertions very efficient. Even under very high load factors ($> 99\%$), long

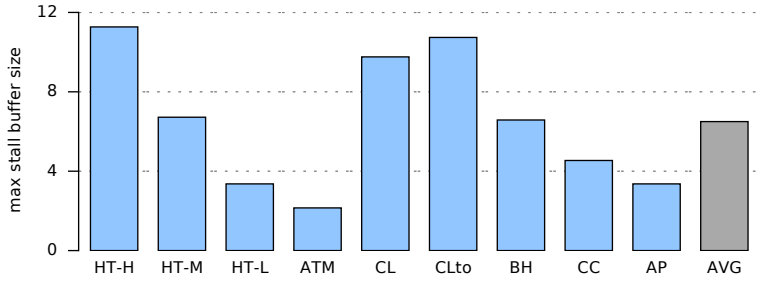



**Figure 4.15:** The maximum number of addresses queued at any given time (total of all stall buffers in the GPU).

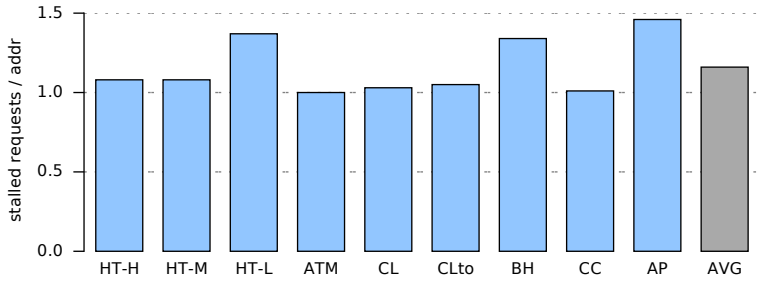



**Figure 4.16:** The average number of requests per address that concurrently reside in the stall buffer.

insert chains where all entries have $\#writes > 0$ are very unlikely; when they do occur, the stash is effective as predicted theoretically [115].

We also investigated the effect of changing metadata table sizes and granularity (Figure 4.14); we tested 2K, 4K, and 8K entries GPU-wide, and 16, 32, 64, and 128-byte granularity assuming 4K table entries GPU-wide. A 2K metadata footprint is too small (and, indeed, requires a larger stash), especially when parallelism is abundant (e.g., HT-H); because 8K entries do not significantly outperform 4K entries, we settled on 4K entries for other parts of the evaluation. Decreasing granularity generally improves performance because false sharing is reduced; however, it also reduces effective table size when parallelism is high and the total number of addresses accessed is higher. We chose 32-byte granularity for all other tests.

Since requests that pass the timestamp check but find their target location reserved are queued in the stall buffer, we measured stall buffer performance. Figure 4.15

| | best concurrency | | | | aborts / 1K commits | | | |
|---|---|---|---|---|---|---|---|---|
| | WTM | EAPG | WTM-EL | GETM | WTM | EAPG | WTM-EL | GETM |
| HT-H | 2 | 2 | 8 | 8 | 119 | 113 | 122 | 460 |
| HT-M | 8 | 4 | 8 | 8 | 98 | 84 | 83 | 172 |
| HT-L | 8 | 4 | 8 | 8 | 80 | 78 | 78 | 207 |
| ATM | 4 | 4 | 4 | 4 | 27 | 26 | 25 | 114 |
| CL | 2 | 2 | 4 | 4 | 93 | 91 | 119 | 205 |
| CLto | 4 | 2 | 4 | 4 | 110 | 61 | 72 | 176 |
| BH | 2 | 2 | 8 | ∞ | 93 | 86 | 145 | 865 |
| CC | ∞ | ∞ | ∞ | ∞ | 6 | 5 | 1 | 38 |
| AP | 1 | 1 | 1 | 1 | 231 | 237 | 204 | 9188 |

**Table 4.4:** Optimal concurrency (# warp transactions per core) settings and abort rates for different workloads. (WTM = WarpTM.)

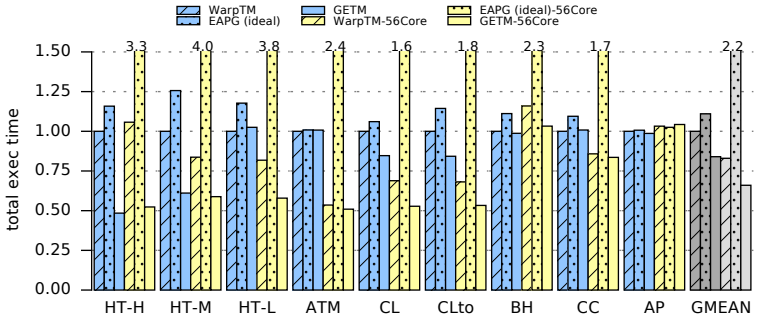



**Figure 4.17:** Program execution time in 15-core and 56-core GPUs, normalized to 15-core WarpTM (lower is better).

shows the maximum total occupancy of all stall buffers; this never rises above 12 requests across the entire GPU. Figure 4.16 shows that very few requests are queued up on average for any given address. In the rest of the evaluation, we conservatively sized the stall buffers to 4 addresses with space for 4 requests each.

### 4.7.3 Transaction Abort Rates

Both WarpTM and GETM limit transactional concurrency to optimize performance. Table 4.4 lists the best concurrency settings for each benchmarks — i.e., the number of warps in each core allowed to run transactions concurrently — and the resulting number of aborted transactions. With abundant parallelism (e.g., HT-H), GETM

| element | area [mm²] | power [mW] |
|---|---|---|
| WarpTM | | |
| CU: LWHR tables (3KB×6) | 0.108 | 21.84 |
| CU: LWHR filters (2KB×6) | 0.03 | 12.00 |
| CU: entry arrays (19KB×6) | 0.402 | 100.62 |
| CU: read-write buffers (32KB×6) | 1.734 | 132.48 |
| TCD: first-read tables (12KB×15) | 0.375 | 113.25 |
| TCD: last-write buffer (16KB total) | 0.031 | 9.86 |
| total WarpTM | 2.68 | 390.05 |
| EAPG (in addition to WarpTM) | | |
| CAT: Conflict Address Table (12KB×15) | 0.6 | 153.3 |
| RCT: Reference Count Table (15KB×6) | 0.294 | 75.6 |
| total EAPG | 3.574 | 618.95 |
| GETM (independent of WarpTM) | | |
| CU: write buffers (16KB×6) | 0.522 | 85.56 |
| VU: precise tables (64KB total) | 0.181 | 69.59 |
| VU: approximate tables (8KB total) | 0.018 | 8.51 |
| *warpts* tables (192B×15) | 0.015 | 10.65 |
| stall buffer (30B×4×6) | 0.0004 | 2.67 |
| total GETM | 0.736 | 176.98 |

**Table 4.5:** CACTI area and power (dynamic + static) estimates for WarpTM [76], EAPG [55], and GETM overheads (32nm node). CU: commit unit; TCD: temporal conflict detection; VU: validation unit.

is efficient at higher concurrency than WarpTM. The eager conflict detection in GETM also translates to dramatically faster commits and aborts than the value-based conflict detection in WarpTM, so GETM can handle higher abort rates and still perform substantially better.

### 4.7.4 Scalability

To investigate scalability at higher core counts, we also simulated WarpTM and GETM in a configuration with 56 SIMT cores and a 4MB LLC; Figure 4.17 shows the results. While performance differences vary slightly per benchmark, the overall trends match the 15-core setup.

### 4.7.5 Area and Power Cost

Table 4.5 shows the area and power overheads introduced by adding TM support. Because GETM removes most of the structures needed by WarpTM, it has 3.6× lower area overheads and 2.2× lower power overheads (4.9× and 3.6× lower than EAPG). Overall, GETM adds ∼0.2% area to a GTX 480 die scaled down to 32nm.

## 4.8 Summary

In this chapter, we present GETM, the first full GPU transactional memory mechanism with eager conflict resolution. By combining explicit version tracking with encounter-time write reservations, GETM enables efficient conflict detection and off-the-critical-path commits. GETM is up to 2.1× faster than the state-of-the-art GPU TM (1.2× gmean), while incurring 3.6× lower area overheads and 2.2× lower power overheads.

# Chapter 5

# Cache Coherence Protocol for Hierarchical Multi-GPU Systems

This chapter studies cache coherence protocol across multi-GPU systems for inter-GPU peer caching. We propose HMG, a hardware-managed cache coherence protocol designed to extend coherence guarantees across forward-looking **H**ierarchical **M**ulti-**G**PU systems with scoped memory consistency models. Unlike prior CPU and GPU protocols that enforce multi-copy-atomicity and/or track ownership within the protocol, HMG eliminates transient states and/or extra hardware structures that would otherwise be needed to cover the latency of acquiring write permissions [216, 221]. HMG also filters out unnecessary invalidation acknowledgment messages, since a write can be processed instantly if multi-copy-atomicity is not required. Similarly, unlike prior work that filters coherence traffic by tracking the read-only state of data regions [216, 253], HMG relies on precise but hierarchical tracking of sharers to mitigate the performance impact of bandwidth-limited inter-GPU links without adding unnecessary coherence traffic. In a 4-GPU system, HMG improves performance over a software-controlled, bulk invalidation-based coherence mechanism by 26% and over a non-hierarchical hardware cache coherence protocol by 18%, thereby achieving 97% of the performance of an idealized caching system.

As the demand for GPU compute continues to grow beyond what a single die can deliver [57, 91, 105, 215], GPU vendors are turning to new packaging technologies such as Multi-Chip Modules (MCMs) [29] and new networking technologies such

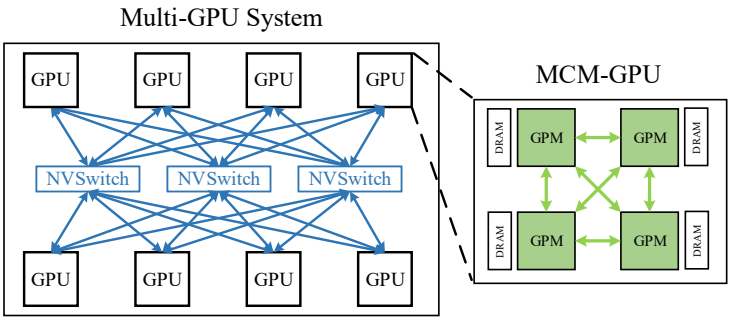



**Figure 5.1:** Forward-looking multi-GPU system. Each GPU has multiple GPU Modules (GPMs).

as NVIDIA's NVLink [160] and NVSwitch [162] and AMD's xGMI [1] in order to build ever-larger GPU systems [170, 173, 174]. Consequently, as Figure 5.1 depicts, modern GPU systems are becoming increasingly hierarchical. However, due to physical limitations, the large bandwidth discrepancy between existing inter-GPU links [1, 160] and on-package integration technologies [187] can contribute to Non-Uniform Memory Access (NUMA) behavior that often bottlenecks performance. Following established principles, GPUs use aggressive caching to recover some of the performance loss created by the NUMA effect [29, 143, 253], and these caches are kept coherent with lightweight coherence protocols that are implemented in software [29, 143], hardware [221, 253], or a mix of both [216].

GPU originally assumed that inter-thread synchronization would be coarse-grained and infrequent, and hence they adopted a bulk-synchronous programming model (BSP) for simplicity. This paradigm disallowed any data communication among Cooperative Thread Arrays (CTAs) of active kernels. However, in emerging applications, less-restrictive data sharing patterns and fine-grained synchronization are expected to be more frequent [43, 54, 113, 217]. BSP is too rigid and inefficient to support these new sharing patterns. To extend GPUs into more general-purpose domains, GPU vendors have released very precisely-defined scoped memory models [2, 98, 112, 135]. These models allow flexible communication and synchronization among threads in the same CTA, the same GPU, or anywhere in the system, usually by requiring programmers to provide *scope* annotations

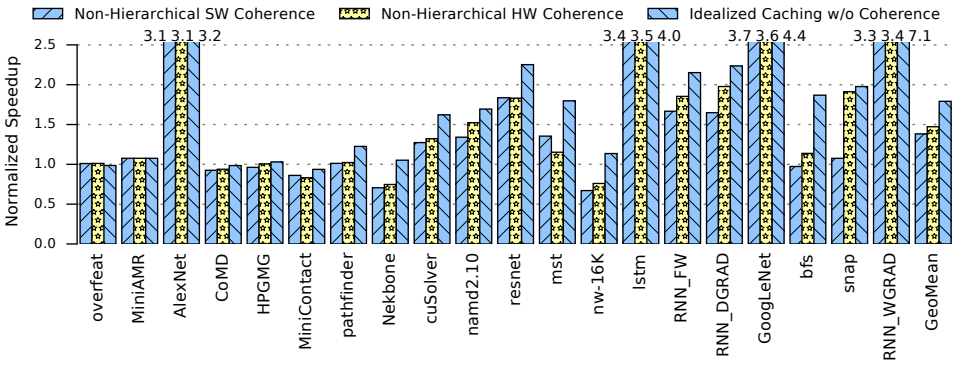



**Figure 5.2:** Benefits of caching remote GPU data under three different protocols on a 4-GPU system with 4 GPMs per GPU, all normalized to a baseline which has no such caching.

for synchronization operations. Scopes allow synchronization and coherence to be maintained entirely within a narrow subset of the full-system cache hierarchy, thereby delivering improved performance over system-wide synchronization enforcement. Furthermore, unlike most CPU memory models, these GPU models are now *non-multi-copy-atomic*: they do not require that memory accesses become visible to all observers at the same logical point in time. As a result, there is room for forward-looking GPU cache coherence protocols to be made even more relaxed, and hence capable of delivering even higher throughput, than protocols proposed in prior work (as outlined in Chapter 5.4).

Previously explored GPU coherence schemes [18, 29, 95, 143, 195, 216, 221, 253] were well tuned for GPUs and much simpler than CPU protocols, but few have studied how to scale these protocols to larger multi-GPU systems with deeper cache hierarchies. To test their efficiency, we simply apply the existing software and hardware coherence protocols GPU-VI [221] on a 4-GPU system, in which each GPU consists of 4 separate GPU Modules (GPMs). These protocols do not account for architectural hierarchy; we simply extend them as if the system were a flat platform of 16 GPMs. As Figure 5.2 shows, in hierarchical multi-GPU systems, existing non-hierarchical software and hardware VI coherence protocols indeed leave plenty room for improvement; see Chapter 5.4 for details. We therefore ask the following question: how do we extend existing GPU coherence protocols

across multiple GPUs while simultaneously providing high-performance support for flexible fine-grained synchronization within emerging GPU applications, and without a dramatic increase in protocol complexity? In this chapter, we answer this question with HMG, a hierarchical cache coherence protocol that is able to scale up to large multi-GPU systems, while nevertheless maintaining the simplicity and implementability that made prior GPU cache coherence protocols popular.

## 5.1 Emerging Programs Need Fine-Grained Communication

Nowadays, many applications contain fine-grained communication between CTAs of the same kernel and/or of dependent kernels [53, 58, 83, 118, 130, 184, 258]. For example, in RNNs, abundant inter-CTA communication exists in the neuron connections between continuous timesteps [64]. In the simulation of molecule or neutron dynamics [184, 258], inter-CTA communication is necessary for the movement dependency between different particles and different simulation timesteps. Graph algorithms usually dispatch vertices among multiple CTAs or kernels that need to exchange their individual update to the graph for the next round of computing until they reach convergence [91, 118]. We provide more details on the workloads we study in Chapter 5.7. All these applications can benefit from a hierarchical GPU system for higher performance and from the scoped memory model for efficient inter-CTA synchronization enforcement.

## 5.2 GPU Weak Memory Model

In the section, to avoid confusion around the term "shared memory", which is used to describe scratchpad memory on NVIDIA GPUs, we use "global memory" for the virtual address space shared by all CPUs and GPUs in a system.

Both CUDA and OpenCL originally supported a coarse-grained bulk-synchronous programming model. Under this paradigm, data sharing between threads of the same CTA could be performed locally in the shared memory scratchpad and synchronized using CTA execution barriers; but inter-CTA synchronization was permitted only between dependent kernel calls (i.e., where data produced by one kernel is consumed by the following kernels). They could not, with guaranteed correctness, perform

arbitrary communication using global memory. While many GPU applications work very well under a bulk-synchronous model with rare inter-CTA synchronizations, it quickly becomes a limiting factor for the types of emerging applications described in Chapter 5.1.

To support data sharing more flexibly and more efficiently, both CUDA and OpenCL have shifted from bulk-synchronous models to more general-purpose scoped memory models [2, 98, 112, 135, 168]. By incorporating the notion of *scope*, these new models allow each thread to communicate with any other threads in the same CTA (`.cta`), the same GPU (`.gpu`), and anywhere in the system (`.sys`)[1]. Scope indicates the set of threads with which a particular memory access wishes to synchronize. Synchronization of scope `.cta` is performed in the L1 cache of each GPU Streaming Multiprocessor (SM); synchronizations of scope `.gpu` and `.sys` are processed through the GPU L2 cache and via the memory hierarchy of the whole system, respectively.

## 5.3 Existing GPU Cache Coherence

Some GPU protocols advocate for strong memory models, and hence they propose sophisticated cache coherence protocols capable of delivering good performance [195]. Most other GPU protocols enforce variants of release consistency by invalidating possibly stale values in caches when performing *acquire* operations (implicitly including the start of a kernel), and by flushing dirty data during *release* operations (implicitly including the end of a kernel). Much of the research in the area today proposes optimizations on top of these basic principles. We broadly classify this work by whether reads or writes are responsible for invalidating stale data.

Among read-initiated protocols, hLRC [18] elided unnecessary cache invalidations and flushes by lazily performing coherence actions when synchronization variables change registration. Furthermore, the recent proposals of DeNovo [216] and VIPS [117] can protect read-only or private data from invalidation. However, they incur additional overheads and/or require software changes to convey

[1]We use NVIDIA terminology in this chapter. Equivalent scopes in HRF are work-group, device, and system [98].

region information for read-only data, ownership tracking in word granularity, or coarse-grained (memory page level)[2] private/shared data classification.

As for write-initiated cache invalidations, previous work has observed that MESI-like coherence protocols are a poor fit for GPUs [95, 221]. QuickRelease [95] reduced the overhead of cache flush by enforcing the partial order of writes with a FIFO. However, QuickRelease needs to partition the resources required by reads and writes; it also broadcasts invalidations to all remote caches. GPU-VI [221] is a simple yet effective hardware cache coherence protocol, but it predated scoped memory models and introduced extra overheads to enforce multi-copy-atomicity, which is no longer necessary. Also, GPU-VI was proposed for use within a single GPU, and did not consider the added complexity of having various bandwidth tiers.

## 5.4 The Novel Coherence Needs of Modern Multi-GPU Systems

To scale coherence across multiple GPUs, the design of HMG not only considers the architectural hierarchy of modern GPU systems (Figure 5.1), but also aggressively takes advantage of the latest scoped memory models (Chapter 5.2). Before diving into the details of HMG, we first describe our main insights below.

### 5.4.1 Extending Coherence to Multiple GPUs

As described in Chapter 5.3, prior GPU coherence protocols mainly focused on mechanisms that mitigate the impact of bulk cache invalidations. However, as Figure 5.2 shows, even fine-grained hardware VI cannot close the gap between what non-hierarchical protocols achieve and an idealized caching scenario. In future multi-GPUs, larger shared L2 caches will only amplify the cost of coarse-grained cache invalidations and of reloading invalidated data from remote GPUs via bandwidth-limited links. Indeed, Figure 5.3 shows that it is common for multiple GPMs on the same GPU to redundantly access a common range of addresses stored on remote GPUs. We therefore build HMG as a hierarchical protocol capable of being extended across multiple GPUs.

[2]GPUs need large pages (e.g., 2MB) to ensure high TLB coverage. Smaller pages can cause severe TLB bottlenecks [30].

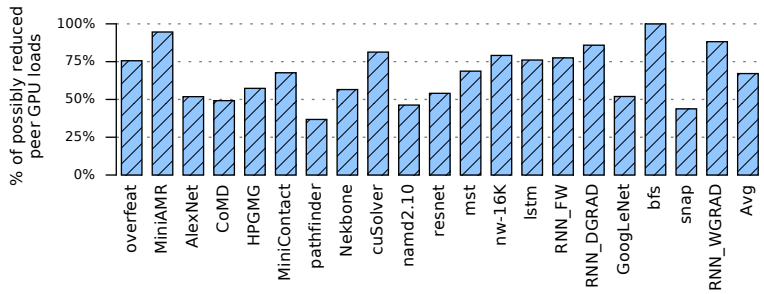



**Figure 5.3:** Percentage of inter-GPU loads destined to addresses accessed by another GPM in the same GPU.

There has been much research into hierarchical cache coherence for CPUs. However, unlike GPUs, CPUs usually enforce a stronger memory model (e.g., TSO) and have much stricter latency requirements. As such, CPU coherence protocols such as MESI track ownership to exploit write data locality [86, 126, 153]. Many transient states are added to reduce the coherence stalls, resulting in prohibitive verification complexity [155, 224]. Industrial products implemented more aggressive optimizations. For example, Sun's WildFire had special OS support for memory page replication and migration [88]. Intel's Skylake introduced IO directory cache and HitMe cache to reduce memory access latency [153]. These complexities are appropriate for latency-bound CPUs, but GPUs permit far more relaxed memory behavior, and hence HMG shows that the costs of such CPU-like protocols remain unnecessary for multi-GPUs.

### 5.4.2 Leveraging GPU Weak Memory Models

Besides the change of hardware architecture, scoped GPU memory models also inform the design of a good GPU coherence hierarchy. While non-scoped CPU memory models require all memory accesses to be kept coherent, GPU memory models that do explicitly expose scopes as part of the programming model require coherence to be enforced only at synchronization boundaries, and only with respect to other threads in the scope in question. The NVIDIA GPU memory model makes this relaxed nature of coherence very explicit [135]. A common pattern in multi-GPU applications will be for CTA or kernels running on a single GPU to

synchronize with each other first, and with kernels on other GPUs less frequently. Such patterns rely heavily on the comparative efficiency of `.gpu` scope over `.sys` scope; while some prior work has concluded that scopes are unnecessary within a single GPU [216], the latency/bandwidth gap between the broadest and narrowest scope is an order of magnitude larger in multi-GPU environments.

Furthermore, although some prior work has proposed multi-copy-atomic memory models for GPUs [16], recent GPU scoped memory models have since formalized the lack of such a requirement [98, 135]. Loosely speaking, multi-copy-atomicity requires memory to behave as if it were a single atomic unit, with only thread-private buffering allowed between cores and memory. As GPUs share an L1 cache across an SM, GPUs today are not multi-copy-atomic. Multi-copy-atomicity also can create apparent delays for subsequent memory accesses. Most CPUs enforce multi-copy-atomicity by using sophisticated coherence protocols with many transient states and by using out-of-order execution and speculation to hide the latency overheads. Some prior studies have found that single-GPU coherence protocols can also tolerate multi-copy-atomicity. For example, to reduce stalls, GPU-VI [221] added 3 and 12 transient states and 24 and 41 coherence state transitions in the L1 and L2 caches, respectively. In multi-GPU environments, however, the round trip time to remote GPUs is an order of magnitude larger and would put significantly increased pressure on the coherence protocol's ability to hide the latency. Instead, by leveraging non-multi-copy-atomicity, HMG eliminates transient states and invalidation acknowledgments altogether.

## 5.5 Baseline Non-Hierarchical Cache Coherence

We now describe how a non-hierarchical cache coherence (NHCC) protocol can be optimized for modern weak GPU memory models. Like most scoped protocols, NHCC propagates synchronization memory accesses to different caches according to the user-provided scope annotations. As compared to GPU-VI [221], NHCC eliminates all transient states and most invalidation acknowledgments. However, it does not consider the architectural hierarchy. As such, it will serve as our baseline during our later evaluations. In the next section, we will extend NHCC with a notion of hierarchy so that it scales better on larger multi-GPU systems like Figure 5.1.

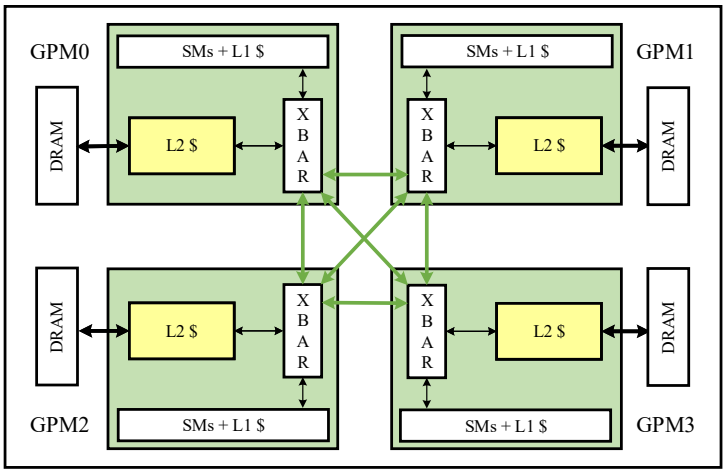



**Figure 5.4:** Future GPUs will consist of multiple GPU Modules (GPMs), and each GPM might be a chiplet in a single package.

### 5.5.1 Architectural Overview

A high-level diagram of our baseline single-GPU architecture for NHCC is shown in Figure 5.4. We assume L1 caches remain software-managed and write-through, as in GPUs today. Each GPU Module (GPM) has an L2 cache that holds both local and remote-GPM DRAM accesses contending for cache capacity with a typical replacement policy such as Least Recently Used (LRU). To support hardware inter-GPM coherence, one GPM in the system is chosen by some hash function as the *home node* for any given physical address. The home node always contains the most up-to-date value at each memory location.

Like many protocols, NHCC attaches an individual directory to every L2 cache within each GPM. The coherence directory is organized as a traditional set-associative structure. Each directory entry tracks the identity of all GPM sharers, along with coherence state. Like GPU-VI [221], each line can be tracked in one of two stable states: *Valid* and *Invalid*. However, unlike GPU-VI, NHCC does not have transient states, and it requires acknowledgments only for release operations. Non-synchronizing stores (i.e., the vast majority) do not require acknowledgments in NHCC.

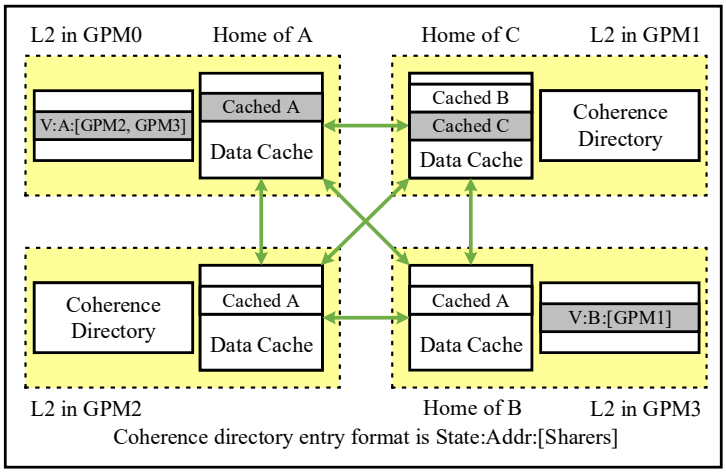



**Figure 5.5:** NHCC coherence architecture. The dotted yellow boxes are the L2 caches from Figure 5.4. The shaded gray cache lines and directory entries indicate lines for which the GPM in question is the *home node*.

We assume a non-inclusive inter-GPM L2 cache architecture to enable data to be cached freely across the different GPMs. Figure 5.5 shows an example in which GPM0 serves as the home node for address A. Other GPMs may cache the value at A locally, but GPM0 maintains the authoritative copy. In the same figure, address B is cached in GPM1, even though GPM3 (the home node for B) is not caching B. Similarly, data can be cached in the home node only, as with address C in our example in Figure 5.5.

In NHCC, explicit coherence maintenance messages (i.e., cache invalidations) are sent only in two cases: when there is read-write sharing between CTAs on different GPMs, and when there is a directory capacity eviction. The fact that most memory accesses incur no coherence overhead ensures that the GPU does not deviate from peak throughput in the GPU common case where data is either read-only or CTA-private. We measure the impact of coherence messages in Chapter 5.8.

To explain the basics of NHCC, we track the life of a memory reference as an example. First, a memory access from the SM queries the L1 cache. Upon a L1 miss or write, the request is routed to the local GPM L2 cache. If the request misses in the L2 (or writes to L2, again assuming a write-through policy), the address is

| State | Local Ld | Local St/Atom | Remote Ld | Remote St/Atom | Replace Dir Entry | Invalidation |
|---|---|---|---|---|---|---|
| **I** | - | - | add *s* to sharers, →V | add *s* to sharers, →V | N/A | - |
| **V** | - | inv all sharers, →I | add *s* to sharers | add *s* to sharers, inv other sharers | inv all sharers, →I | forward inv to all sharers (HMG only), →I |

**Table 5.1:** NHCC and HMG coherence directory transition table. *s* refers to the sender of the message.

checked to determine if the local GPM is the home node for this reference. If so, the request is routed to local DRAM. Otherwise, the request is routed to the L2 cache of the home node via the inter-GPM links. The request may then hit in the home node L2 cache, or it may get routed through to that particular GPM's off-chip memory. We provide full details below.

### 5.5.2 Coherence Protocol Flows in Detail

Table 5.1 details the full operation of NHCC. In this table, "local" refers to operations issued by the same GPM as the L2 cache which is handling the request. "Remote" requests are those originally issued by other GPMs. We walk through the entries in the table below.

**Local Loads:** When a local load request reaches the local L2 cache partition, if it hits, a reply is sent back to the requester directly. If the request misses, the next destination depends on where the data is mapped by the address hash function. If the local L2 cache partition happens to be the home node for the address being accessed, the request will be sent to DRAM. Otherwise, the load request will be forwarded to the home node. Loads with `.gpu` or `.sys` scope must always miss in the L1 cache and in the non-home L2 caches to guarantee forward progress.

**Local Stores:** Depending on L2 design, local stores may be stored as dirty data in the L2 cache, or alternatively they may be written through and stored as clean data in the L2 cache. All stores with scope greater than `.cta` (i.e., `.gpu` and `.sys`) must be written through in order to ensure forward progress. Data which is written back or written through the L2 is sent directly to DRAM if the local L2 cache is the home node for the address in question, or it is relayed to the home node otherwise.

If the local GPM is the home node and the coherence directory has recorded any sharers for the address in question, then these sharers must be notified that the data has been changed. As such, a local store triggers an invalidation message being sent to each sharer. These invalidations propagate in the background, off the critical path of any subsequent reads. There are no invalidation acknowledgments.

**Remote Loads:** When a remote load arrives at the local home L2 cache, it either hits in the cache and returns data to the requester, or it misses and forwards the request to DRAM. The coherence directory also records the ID of the requesting node. If the line is already being tracked, the requesting ID is simply added as an additional sharer. If the line is not being tracked yet, a new entry is allocated in the directory, possibly by evicting another valid entry (discussed further below).

**Remote Stores:** Remote stores that arrive at a home L2 are cached and written through or written back to DRAM, depending on the configuration of the L2. Since the requesting GPM may also be caching the stored data, the requester is recorded as a sharer. Since the data has been changed, all other sharers should be invalidated.

**Atomics and Reductions:** Atomic operations must always be performed at the home node L2. From a coherence transition perspective, these operations are treated as stores.

**Invalidations:** Upon receiving an invalidation request, any local clean copy of the address in question is invalidated. No acknowledgment needs to be sent.

**Directory Entry Eviction/Replacement:** Because the coherence directory is implemented as a set-associative cache, there may be entry evictions due to capacity and conflict misses. To ensure correctness, invalidation messages must be sent to all sharers of the entry that is being evicted. As with invalidations triggered by stores, these invalidations propagate in the background and do not require acknowledgments to be sent in return.

**Acquire:** Acquire operations greater than `.cta` scope (i.e., `.gpu` and `.sys`) invalidate the entire local L1 cache, following software coherence practice. However, they do not propagate past the L1 cache, as L2 coherence in GPMs is now maintained using NHCC.

**Release:** Release operations trigger a writeback of all dirty data to the respective home nodes, if writeback caches are being used. Releases also ensure completion of any write-through operations and invalidation messages that are still in flight. Release

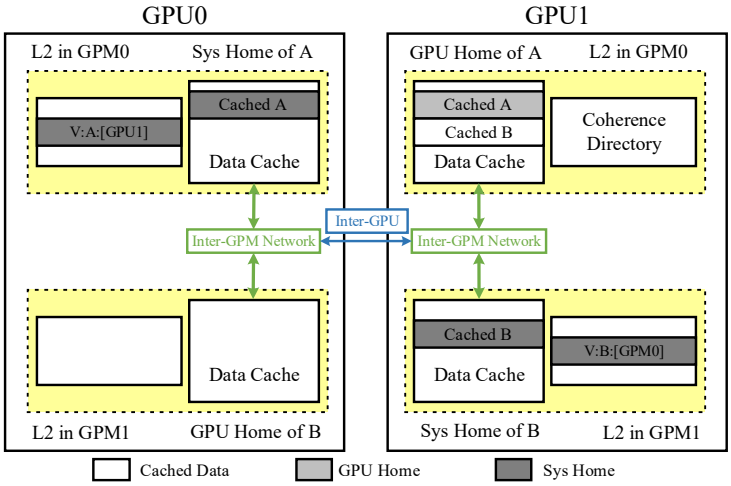
operations greater than `.cta` scope are propagated through the local L2 to all remote L2s to ensure that all invalidation messages have arrived at their destinations. Once this step is complete, each remote L2 sends back an acknowledgment for the release operation itself. The local L2 then collects these acknowledgments and returns a single response to the original requester.

**Cache Eviction:** Two design options are possible upon cache line eviction. First, a clean cache line being evicted from an L2 cache in a non-home GPM could send a downgrade message to the home node. This allows the home node to delete the remote node as a sharer and will potentially save an invalidation message from being sent later. However, this is not required for correctness. The second option is to have valid clean cache lines get silently evicted. This eliminates the overhead of the downgrade message, but it triggers an unneeded invalidation message upon eventual eviction of the coherence directory entry. Optionally, dirty cache lines being evicted and written back can use a new message type indicating that the data must be updated but that the issuing GPM need not be tracked as a sharer going forward. Again, this optimization is not strictly required for correctness, but may be useful in implementations using writeback caches.

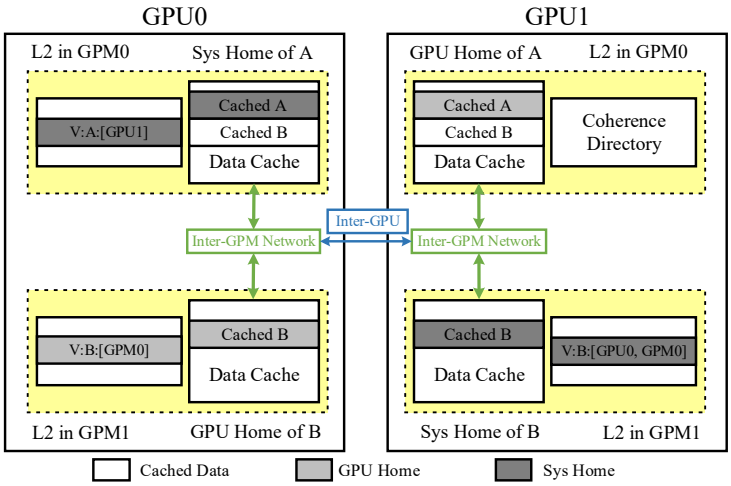
## 5.6 Hierarchical Multi-GPU Cache Coherence

Like most prior work, NHCC is designed for single-GPU scenarios and does not take the hierarchy between intra- and inter-GPU connections into account. This becomes a problem as we try to extend protocols like NHCC to multiple GPUs, as inter-GPU bandwidth limitations become a bottleneck.

To better exploit intra-GPU data locality, we propose a *hierarchical multi-GPU* (HMG) cache coherence protocol that extends NHCC to be able to take advantage of the type of locality that Figure 5.3 highlights. The HMG protocol fundamentally enables multiple cache requests from individual GPMs to be coalesced and/or cached within a single GPU *before* traversing the lower-bandwidth inter-GPU links, thereby saving bandwidth and energy.




**(a)** Before: GPU0:GPM0 is about to load address B




**(b)** After: B is cached in the L2 of the GPU0 home node for B as well as in the L2 of the original requester

**Figure 5.6:** Hierarchical coherence in multi-GPU systems. Loads are routed from the requesting GPM to the GPU home node, and then to the system home node, and responses are returned and cached accordingly.

### 5.6.1 Architectural Overview

HMG is composed of two layers. The first layer is designed for intra-GPU caching, while the second layer is targeted at optimizing memory request routing in inter-GPU settings. For the intra-GPU layer, we define a *GPU home node* for each given address within each individual MCM-GPU. An MCM-GPU home node manages inter-GPM coherence using NHCC described in Chapter 5.5. Using the intra-GPU coherence layer, data that is cached within a MCM-GPU can be consumed by multiple GPMs on that GPU without consulting a remote GPU.

We define one of the GPU home nodes for each address to be the *system home node*. The choice of system home node can be made using any NUMA page allocation policy, such as first touch page placement, NVIDIA Unified Memory [168], static distribution, or any other reasonable heuristic. Among multiple GPUs, sharers are tracked by the directory using a hierarchy-aware variant of the NHCC directory design. Specifically, each GPU home node will track any sharers among other GPMs in the same GPU. Each system home node will track any sharers among other GPUs, but not individual GPMs within these other GPUs. For an M-GPM, N-GPU system, each directory entry will therefore need to track as many as $M\ N-2$ sharers.

The hierarchical caching mechanism of an example two-GPU system is shown in Figure 5.6. Each GPU is shown with only two GPMs for brevity, but the protocol itself can extend to an arbitrary number of GPUs, with an arbitrary number of GPMs per GPU. In Figure 5.6(a), the system home node of address A is the L2 cache residing in GPU0:GPM0. This particular L2 cache also serves as the GPU home node for the same address within GPU0. The L2 cache in GPU0:GPM1 is kept coherent with the L2 cache in GPU0:GPM0 using the intra-GPU protocol layer. The L2 cache in GPU1:GPM0 serves as the GPU1 home node for address A, and it is kept coherent with the L2 cache in GPU1:GPM1 using the intra-GPU layer. Both GPU home nodes are kept coherent using the inter-GPU protocol layer.

Furthermore, suppose that from the state shown in Figure 5.6(a), GPU0:GPM0 wants to load address B, and the system home node for address B is mapped to GPU1:GPM1. GPU0:GPM1 is the GPU0 home node for B, so the load request propagates from GPU0:GPM0 to GPU0:GPM1 (the GPU home node), and then to GPU1:GPM1 (the system home node). When the response is sent back to the

requester, GPU0 (but *not* GPU0:GPM0 or GPU0:GPM1) is recorded as a sharer by the directory of the system home node GPU1:GPM1, and GPU0:GPM0 is recorded as a sharer by the directory of the GPU0 home node GPU0:GPM1, as shown in Figure 5.6(b).

### 5.6.2 Coherence Protocol Flows in Detail

HMG behaves similarly to Table 5.1 but adds the single extra transition shown in Table 5.1. No extra coherence states are added. We highlight the important differences between NHCC and HMG as follows.

**Loads:** Loads progress through the cache hierarchy from the local L2 cache, to the GPU home node, to the system home node. Specifically, loads that miss in the GPM-local L2 cache are routed to the GPU home node, unless the GPM-local L2 cache is already the GPU home node. From there, loads that miss in the GPU home node are routed to the system home node, unless the GPU home node is also the system home node. Loads that miss in the system home node are routed to DRAM.

Non-synchronizing loads (i.e., the vast majority) and loads with `.cta` scope can hit in all caches. However, loads with `.gpu` scope must miss in all caches prior to the GPU home node. Loads with `.sys` scope must also miss in the GPU home node; they may only hit in the system home node.

Loads propagating from the GPU home node to the system home node do not carry information about the GPM that originally requested the data. Because this information is already stored by the GPU home node, it would be redundant to store it again in the directory of the system home node. Instead, invalidations are propagated to sharers hierarchically as described below.

**Stores:** Stores are routed through a similar hierarchy as they write-through and/or write-back. Specifically, stores propagating past the GPM-local L2 cache are routed to the GPU home node (unless the GPM-local L2 is already the GPU home node), and stores propagating past the GPU home node are routed to the system home node (unless the GPU home node is already the system home node). Stores propagating past the system home node are written to DRAM. Similar to loads, stores or write-back/write-through operations propagating from the GPU home node to the system home node carry only the GPU identifier, not the identifier of the GPM

within that GPU.

Stores must be written through at least to the home node for the scope in question: the L1 cache for non-synchronizing and `.cta`-scoped stores, the GPU home node for `.gpu`-scoped stores, and the system home node for `.sys`-scoped stores. This ensures that synchronization operations will make forward progress.

**Atomics and Reductions:** Atomics are always performed in the home node for the scope in question and they continue to be treated as stores for the purposes of coherence protocol transitions, just as in NHCC. Once performed at the home node, responses are propagated back to the requester just as load responses are handled and the result is stored as a dirty line or written through to subsequent levels of the cache hierarchy, just as a store would be. For example, the result of a `.gpu`-scoped atomic read-modify-write operation performed in the GPU will be written through to the system home node, in systems which configure the GPU home node to be write-through for stores.

**Invalidations:** Because sharers are tracked hierarchically, invalidations sent due to stores and directory evictions must also propagate hierarchically. Invalidations sent from the system or GPU home node to other GPMs in the same GPU are processed and dropped without acknowledgment, just as in NHCC. However, in HMG any invalidations received by a GPU home node from the system home node must also be propagated to any and all GPM sharers within the same GPU. This is the special transition shown in Table 5.1 for HMG.

**Acquire:** As before, `.cta`-scoped acquire operations invalidate the local L1 cache, but nothing more, as all levels of L2 cache are being kept hardware-coherent.

**Release:** Release operations trigger writeback of all dirty data, at least to the home node for the scope being released. They also still ensure completion of any write-through operations and invalidation messages still in flight to the home node for the scope in question. A `.gpu`-scoped release operation, however, need not flush all write-back operations across the inter-GPU network before returning a completion acknowledgment to the original requester.

| Structure | Configuration |
|---|---|
| Number of GPUs | 4 |
| Number of SMs | 128 per GPU, 512 in total |
| Number of GPMs | 4 per GPU |
| GPU frequency | 1.3GHz |
| Max number of warps | 64 per SM |
| OS Page Size | 2MB |
| L1 data cache | 128KB per SM, 128B lines |
| L2 data cache | 12MB per GPU<br>128B lines, 16 ways |
| L2 coherence directory | 12K entries per GPU module<br>each entry covers 4 cache lines |
| Inter-GPM bandwidth | 2TB/s per GPU, bi-directional |
| Inter-GPU bandwidth | 200GB/s per link, bi-directional |
| Total DRAM bandwidth | 1TB/s per GPU |
| Total DRAM capacity | 32GB per GPU |

**Table 5.2:** Simulated GPU and memory hierarchy for HMG.

## 5.7 Methodology

To evaluate HMG, we use a proprietary industrial simulator to model a multi-GPU system described in Table 5.2. The simulator is driven by program traces that record instructions, registers, memory addresses, and CUDA events. All micro-architectural scheduling, and thus time for execution, is dynamic within the simulator and respects functional dependencies such as work scheduling, barrier synchronization, memory access latencies. However, it cannot accurately model spin-lock synchronizations in memory. While this type of communication is legal on current NVIDIA hardware, it is not yet widely adopted due to performance overheads and not present in our suite of workloads. Simulating the system-level effects of fine-grained synchronization, in reasonable time, without sacrificing fidelity [217, 229] remains an open problem for GPU researchers.

Figure 5.7 shows our simulator correlation versus a NVIDIA Quadro-GV100 GPU across a range of targeted microbenchmarks, public, and proprietary workloads. Figure 5.7 also shows the corresponding data for GPGPU-Sim, a widely-used academic GPU architecture simulator [31, 108, 111, 128, 190], with simulations capped at running for about one week. Our simulator has a correlation coefficient of 0.99 and average absolute error of 0.13. This compares favorably to GPGPU-Sim (at

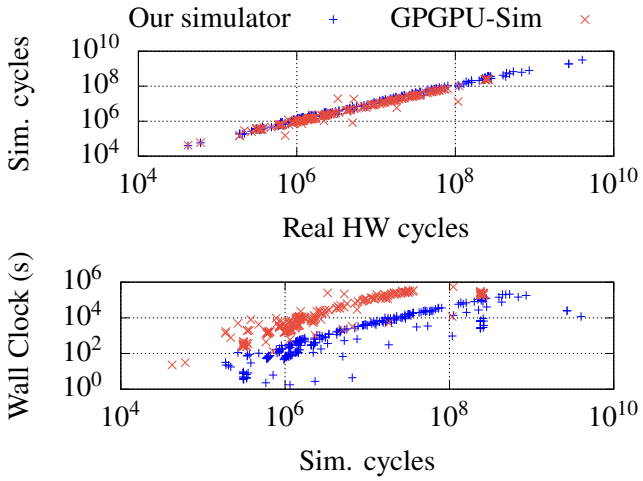



**Figure 5.7:** Simulator correlation vs. a NVIDIA Quadro GV100 and simulation runtime for our simulator and GPGPU-Sim.

0.99 and 0.045, respectively), as well as other recently reported simulator results[87] while being significantly faster, which allows us to run forward-looking GPU configurations more easily. Our simulator inherits the contiguous CTA scheduling and first-touch page placement polices from prior work [29, 143] to maximize data locality in memory.

To perform our evaluation, we choose a public subset of workloads (shown in Table 5.3) [53, 58, 83, 118, 130, 184, 258] that have sufficient parallelism to fill a 4-GPU system. These benchmarks utilize scoped and/or inter-kernel synchronization patterns. This ensures that performance does not regress on traditional workloads even as we accelerate workloads with more fine-grained sharing. Specifically, `cuSolver`, `namd2.10`, and `mst` use `.gpu`-scoped synchronization explicitly, others utilize inter-kernel communication by launching frequent dependent kernels, and a few are traditional bulk-synchronous providing a historical comparative baseline.

**Coherence Protocol Implementations:** This work implements and compares 4 coherence possibilities: a non-hierarchical software protocol (conventional software coherence with scopes and bulk-invalidation of caches), a non-hierarchical hardware protocol (NHCC), a hierarchical software protocol (conventional software coherence

| Benchmark | Abbrev. | Footprint |
|---|---|---|
| cuSolver | cuSolver | 1.60 GB |
| HPC_CoMD-xyz49 | CoMD | 313 MB |
| HPC_HPGMG | HPGMG | 1.32 GB |
| HPC_MiniAMR-test2 | MiniAMR | 1.80 GB |
| HPC_MiniContact | MiniContact | 246 MB |
| HPC_namd2.10 | namd2.10 | 72 MB |
| HPC_Nekbone-10 | Nekbone | 178 MB |
| HPC_snap | snap | 3.44 GB |
| Lonestar_bfs-road-fla | bfs | 26 MB |
| Lonestar_mst-road-fla | mst | 83 MB |
| ML_AlexNet_conv2 | AlexNet | 812 MB |
| ML_GoogLeNet_conv2 | GoogLeNet | 1.15 GB |
| ML_lstm_layer2 | lstm | 710 MB |
| ML_overfeat_layer1 | overfeat | 618 MB |
| ML_resnet | resnet | 3.20 GB |
| ML_RNN_layer4_DGRAD | RNN_DGRAD | 29 MB |
| ML_RNN_layer4_FW | RNN_FW | 40 MB |
| ML_RNN_layer4_WGRAD | RNN_WGRAD | 38 MB |
| Rodinia_nw-16K-10 | nw-16K | 2.00 GB |
| Rodinia_pathfinder | pathfinder | 1.49 GB |

**Table 5.3:** Benchmarks used for HMG evaluation.

with hierarchical extension to leverage scopes), and our proposed hierarchical hardware protocol (HMG). We also compare them to idealized caching that does not enforce coherence; this serves as a loose upper bound for performance that can be achieved via hardware caching. For non-hierarchical protocols, multi-GPU systems like Figure 5.1 behaves as a single flat GPU with more GPMs.

NHCC and HMG behave according to Chapter 5.5 and 5.6 respectively. Load-acquire operations in our software coherence protocols trigger bulk cache invalidations in any caches between the issuing SM and the home node for the scope in question. For example, `.gpu`-scoped loads will invalidate both the L1 cache of the issuing SM and the GPM-local L2 cache. In the hierarchical protocol, `.sys`-scoped loads invalidate the L1 cache of the issuing SM and all L2 caches of the issuing GPU. However, in the non-hierarchical protocol, `.sys`-scoped loads need not to invalidate L2 caches in other GPMs of the same GPU, as subsequent loads will not fetch stale data from those caches. Store-release operations stall subsequent operations until the home node for the scope in question clears all pending writes.

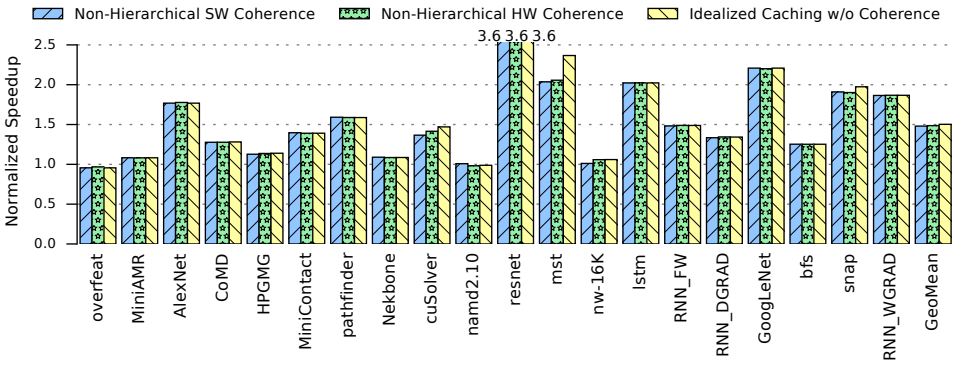



**Figure 5.8:** Performance of various inter-GPM coherence schemes in a single GPU with 4 GPMs. Performance is normalized to a scheme that does not perform inter-GPM caching.

In our evaluation, all caches are write-through. We do not implement the optional sharer downgrade messages. We model one directory optimization: each entry tracks the state of four cache lines together. This enables $12K \times 4 \times 128B = 6MB$ of data assigned to each GPM to be actively shared by other GPMs and/or GPUs. Chapter 5.8.2 later shows performance sensitivity to the choices of these parameters.

## 5.8 Evaluation Results

We first compare the performance of HMG to NHCC, software coherence protocols, and idealized caching without any coherence overhead. Then we conduct sensitivity analysis to explore the design space of HMG.

### 5.8.1 Performance Analysis

**Single-GPU System:** As Figure 5.8 shows, we observe that for most benchmarks, both software and hardware coherence generally perform similarly and close to an idealized non-coherent caching scheme. The relatively small L2 caches and relatively large inter-GPM bandwidths can minimize the performance penalty of cache invalidations in single-GPU systems, and hence we do not elaborate on them further here.

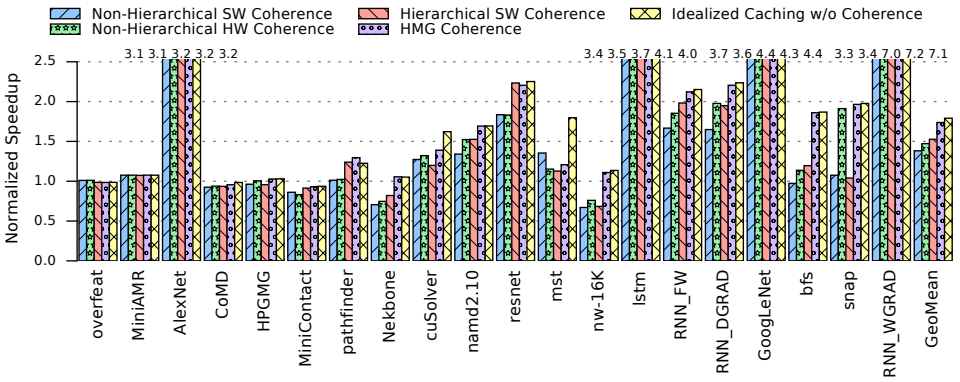



**Figure 5.9:** Performance of a 4-GPU system, where each GPU is composed of 4 GPMs. Performance is normalized to a 4-GPU system that disallows caching of remote GPU data. Five configurations are evaluated: software protocols with non-hierarchical and hierarchical implementations, NHCC, HMG, and ideal caching without coherence overhead.

**Multi-GPU System:** While software coherence may be sufficient within individual GPUs, even for benchmarks with fine-grained thread-to-thread communication, Figure 5.9 shows that the benefits of HMG are much more pronounced in deeply hierarchical multi-GPU systems, especially for the applications which have more fine-grained data sharing (i.e, the right half side). In a 4-GPU system, HMG generally outperforms both software coherence protocols and NHCC. Both software and hardware hierarchical protocols significantly benefit from the additional intra-GPU data locality. Meanwhile, the non-hierarchical protocols suffer from larger inter-GPU latency and bandwidth penalties.

Figure 5.10 and 5.11 show that cache line invalidations due to store instructions or coherence directory evictions do not have a significant impact on performance of HMG. This is because stores only trigger invalidations if there is a sharer for the same address and typically only a small percentage of the memory footprint of each workload contains read-write shared data. Even among stores or directory evictions that do trigger sharer invalidations, there are generally no more than two sharers in our workloads. These observations highlight the benefit of tracking sharers dynamically, rather than e.g., classifying data sharing type alone [253].

Graph workloads' fine-grained, often conflicting access patterns can lead to false

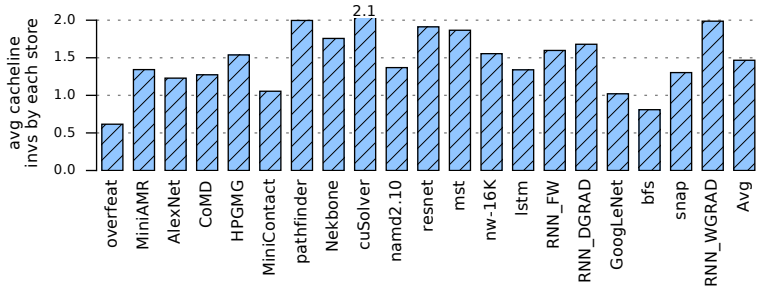



**Figure 5.10:** Average number of cache lines invalidated by each store request on shared data.

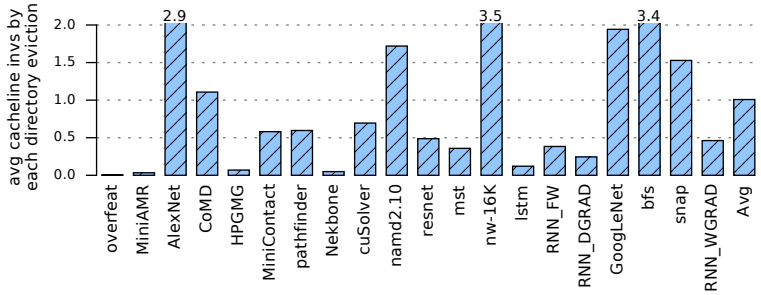



**Figure 5.11:** Average number of cache lines invalidated by each coherence directory eviction.

sharing. Store operations in software coherence protocols will simply write this data through, but HMG might trigger frequent invalidations (in these experiments, at the granularity of four cache lines per directory entry), depending on the input sets. In such cases, the hardware protocol HMG will have higher overhead. This explains the performance of `mst`, for example. For most other applications, the benefits of HMG outweigh the costs.

We also profile the bandwidth overhead of invalidation messages. Figure 5.12 shows that the total bandwidth cost of invalidation messages is generally as low as just a few gigabytes per second. This is consistent with prior data since there is little read-write sharing and a low number of sharers when invalidations must be sent out. The size of each invalidation message is also relatively small compared to a GPU cache line. Combined with the fact that GPU workloads are generally latency tolerant, it becomes clear that HMG for hierarchical multi-GPUs can deliver high

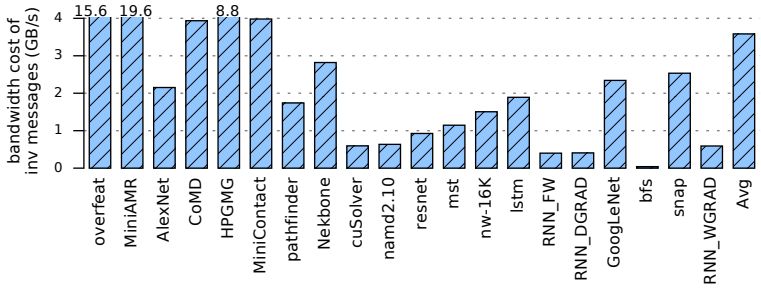



**Figure 5.12:** Total bandwidth cost of invalidation messages.

performance, at high efficiency, with relatively simple hardware implementation. Overall, our results confirm prior suggestions that complicated CPU-like coherence protocols are unnecessary, even in hierarchical multi-GPU contexts. By providing a lightweight coherence enforcement mechanism specifically tuned to the scoped memory model, HMG is able to deliver 97% of the ideal speedup that inter-GPU caching can possibly enable.

### 5.8.2 Sensitivity Analysis

To understand the relationship between our architectural parameters and the performance of HMG, we performed sensitivity studies across a range of design space parameters.

- Bandwidth-limited inter-GPU links are the main cause of NUMA effects that often bottleneck multi-GPU performance. Figure 5.13 shows that when sweeping across inter-GPU bandwidths, HMG is always the best performing coherence option, even when absolute performance begins to saturate due to sufficient inter-GPU bandwidth.

- The impact of L2 cache size on performance is shown in Figure 5.14. Because of the overhead of cache invalidation, the benefits of increased L2 capacity are restricted by software coherence protocols. Conversely, the performance of HMG increases as capacity grows, indicating the advantage of HMG will only become more favorable in systems with larger caches.

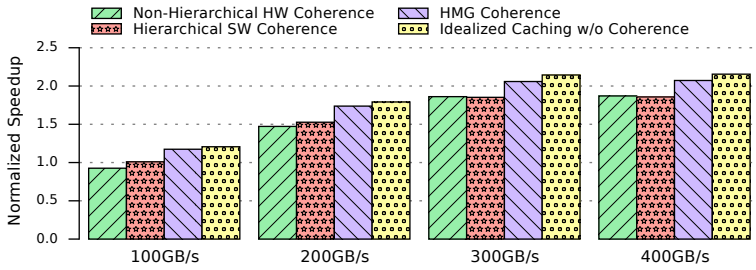



**Figure 5.13:** Performance sensitivity to inter-GPU bandwidth (baseline is no caching with configurations of Table 5.2).

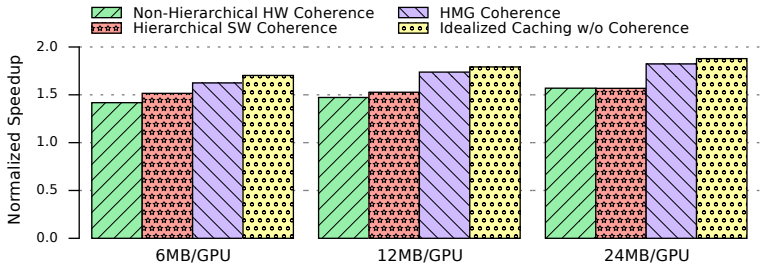



**Figure 5.14:** Performance sensitivity to L2 cache size (baseline is no caching with configurations of Table 5.2).

- Coherence directory sizing presents a trade-off between power/area and coverage/performance. As Figure 5.15 shows, the performance of our proposed HMG is somewhat sensitive to directory size. The benefit of hardware-managed coherence over software coherence shrinks if the directory is not able to track enough sharers and is forced to perform additional cache invalidations across GPUs. However, our modestly-sized directories are large enough to successfully capture the locality needed to deliver near-ideal caching performance.

- Coarse-grained directory entry tracking granularity (e.g., where each entry tracks four cache lines at a time) allows directories to be made smaller, but it also introduces a risk of false sharing. In order to quantify this impact, we varied the granularity tracked by each directory entry while simultaneously adjusting the total number of entries in order to keep the total coverage constant. The results (Figure 5.16) showed minimal sensitivity,

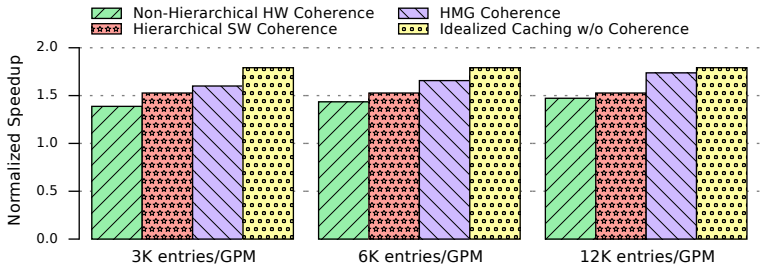



**Figure 5.15:** Performance sensitivity to the coherence directory size (baseline is no caching with configurations of Table 5.2).

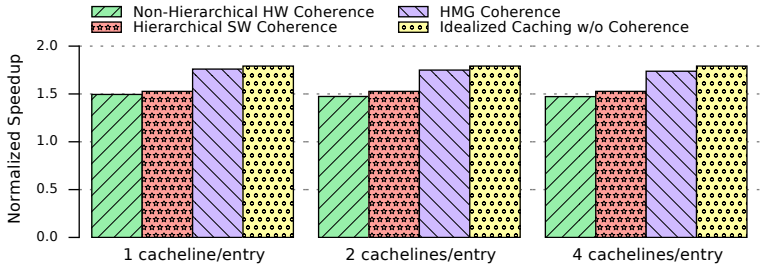



**Figure 5.16:** Performance sensitivity to the coherence directory tracking granularity (baseline is no caching with configurations of Table 5.2).

and we therefore conclude that coarse-grained directory tracking is a useful optimization for HMG.

### 5.8.3 Hardware Costs

In our HMG implementation, each directory entry needs to track as many as six sharers: three GPMs in the same GPU and three other GPUs. Therefore, a 6 bit vector is required for the sharer list. Because our protocol uses just two states, Valid and Invalid, only one bit is needed to track directory entry state. We assume 48 bits for tag addresses, so each entry in the coherence directory requires 55 bits of storage. Every GPM has 12K directory entries, so the total storage cost of the coherence directories is 84KB, which is only 2.7% of each GPM's L2 cache data capacity, a small price to pay for large performance improvements in future multi-GPUs.

### 5.8.4 Discussion

On-package integration [22, 29, 187] along with off-package integration technologies [1, 160, 162] enable more and more GPU modules to be integrated in a single systems. However, NUMA effects are exacerbated as the number of GPMs, GPUs, and non-uniform topologies increase within the system. In these situations, HMG's coherence directory would need to record more sharers and cover a larger footprint of shared data, but the system performance will likely be more sensitive to the link speeds and the actual network topology. As shown in Figure 5.15, HMG can perform very well even after we reduce the coherence directory size by 50%, showing that there is still room to scale HMG to larger systems. We envision our proposed coherence protocol being applicable for systems that can be comprised by a single NVSwitch-based network within a single operating system node. Systems significantly larger than this (e.g., 1024-GPU systems) may be decomposed into a hierarchy consisting of hardware-coherent GPU-clusters which are in turn share data using software mechanisms such as MPI or SHMEM [141, 180].

The rise of MCM-GPU-like architectures might seem to motivate adding scopes in between `.cta` and `.gpu`, to minimize the negative effects of coherence. However, our single-GPU performance results indicate that our workloads are minimally sensitive to the inter-GPM coherence mechanism due to high inter-GPM bandwidth. As a result, the performance benefits of introducing a new `.gpm` scope may not outweigh the added programmer burden of using numerous scopes. We expect further exploration of other software-hardware coherence interactions to remain an active area of research as GPU systems continue to grow in size.

## 5.9 Summary

In this chapter, we introduce HMG, a novel cache coherence protocol specifically tailored to scale well to hierarchical multi-GPU systems. HMG provides efficient support for fine-grained synchronization now permitted under recently-formalized scoped GPU memory models. We find that, without much complexity, simple hierarchical extensions and optimizations to existing coherence protocols can take advantage of relaxations now permitted in scoped memory models to achieve 97% performance of an ideal caching scheme that has no coherence overhead. Thanks to

its cheap hardware implementation and high performance, HMG demonstrates the most practical solution available for extending cache coherence to future hierarchical multi-GPU systems, and thereby for enabling continued performance scaling of applications onto larger and larger GPU-based systems.

# Chapter 6

# Scalable Multi-GPU Rendering via Parallel Image Composition

In this chapter, we propose CHOPIN, a Split Frame Rendering (SFR) technique that eliminates the performance overheads of prior solutions by leveraging parallel image composition. Unlike prior work, draw commands are distributed across different GPUs to remove redundant computation, and image composition is performed in parallel to obviate the need for sequential primitives exchanging. CHOPIN includes a novel draw command scheduler that predicts the proper GPU for each draw command to avoid the inter-GPU load imbalance, and a novel image composition scheduler to reduce network congestion that can easily result from naïve inter-GPU sub-image exchange. Through an in-depth analysis using cycle-level simulations on a range of real-world game traces, we demonstrate that CHOPIN outperforms the prior state-of-the-art SFR implementation by up to 1.56× (1.25× gmean) in an 8-GPU system.

GPUs were originally developed to accelerate graphics processing — the process of generating 2D-view images from 3D models [134]. Although much recent computer architecture research has focused on using GPUs for general-purpose computing, high-performance graphics processing has historically accounted for the lion's share of demand for GPUs. This continues to be the case, with graphics remaining the dominant source of revenue for GPU vendors: for example, NVIDIA's year 2019 revenues from the gaming (GeForce) and professional visualization (Quadro)

markets combined are 2.5× and 11.5× higher than that from the datacenter and automotive markets, respectively [176]. This is driven by many applications, including gaming, scientific data visualization, computer-aided design, Virtual Reality (VR), Augmented Reality (AR), and so on. Gaming itself continues to evolve: 4K/UHD high-resolution gaming requires 4× as many pixels to be rendered as 1080p HD gaming [4], while VR gaming is 7× more performance demanding than 1080p [169]. These requirements have imposed unprecedented challenges on vendors seeking to provide a high-quality experience to end consumers.

This need for substantial performance improvements has, however, been increasingly difficult to satisfy with conventional single-chip GPU systems. To continue scaling GPU performance, GPU vendors have recently built larger systems [170, 173, 174] that rely on distributed architectures such as Multi-Chip-Module GPU (MCM-GPU) [29] and multi-GPUs [143, 197, 253]. MCM-GPU and multi-GPU systems promise to push the frontiers of performance scaling much farther by connecting multiple GPU chip modules (GPMs) or GPUs with advanced packaging [187] and networking technologies, such as NVLink [160], NVSwitch [162], and XGMI [1]. In principle, these platforms can offer substantial opportunities for performance improvement; in practice, however, their performance tradeoffs for graphics processing are different from that of single-chip GPUs, and fully realizing the benefits requires the use of distributed rendering algorithms.

Distributed rendering is, of course, not new: GPU vendors have long combined two to four GPUs using techniques like SLI [166] and Crossfire [20]. They distribute the rendering workload using either Alternate Frame Rendering (AFR), where different GPUs process consecutive frames, or Split Frame Rendering (SFR), which assigns disjoint regions of a single frame to different GPUs. By processing alternate frames independently, AFR improves the overall rendering throughput, but does nothing to improve single-frame latencies. While the *average* frame rate improves, the *instantaneous* frame rate can be significantly lower than the *average* frame rate. This problem, called micro-stuttering, is inherent to AFR, and can result in a dramatically degraded gameplay experience [3, 5, 8]. In contrast, SFR can improve both the frame rate and the single-frame latencies [70, 103, 152]. Therefore, SFR is more widely used in practice, and we focus on SFR in this chapter. The tradeoff, however, is that SFR requires GPUs to exchange data for both inter- and intra-frame

data dependencies, which creates significant bandwidth and latency challenges for the inter-GPU interconnect.

While the recent introduction of high-performance interconnects like NVLink and XGMI promises to conquer the inter-GPU communication bandwidth constraints, key challenges still remain. SFR assigns split screen regions to separate GPUs, but the mapping of primitive (typically triangle) coordinates to screen regions is not known ahead of time, and must be computed before distributing work among GPUs. CPU pre-processing techniques [70, 102, 103] are limited by the low computing and data throughput. GPU methods rely on redundant computation — every GPU projects *all* 3D primitives to the 2D screen space and retain only the primitives within its assigned screen region — but this incurs significant overheads on modern high-triangle-count workloads in multi-GPU systems, and does not take advantage of the new high-speed interconnects. Recent work GPUpd [114] has attempted to reduce the redundant computation through additional interconnect traffic, but is bottlenecked by sequential inter-GPU primitive redistribution needed to protect the input order of primitives (see Chapter 6.2 for a detailed analysis of prior SFR solutions). Therefore, there is an urgent need for parallel rendering schemes that can leverage today's high-speed interconnects and reliably scale to multi-GPU systems.

## 6.1 Parallel Image Composition

Image composition is the reduction of several images into one, and is performed on pixel granularity. The reduction process is a sequence of operations, each of which has two inputs: the current pixel value $p_{old}$ and the incoming value $p_{new}$. The two are combined using an application-dependent function $f$ to produce the updated pixel $p = f\,p_{old}, p_{new}$. The exact definition of $f$ depends on the task: for example, $f$ can select the pixel which is closer to the camera, or blend the colour values of the two pixels. A common blending operation is the *over* operator [186] $p = p_{new}\ 1 - \alpha_{new} * p_{old}$, where $p$ represents the pixel colour and opacity components, and $\alpha$ is the pixel opacity only. Other blending operators include *addition*, *multiplication*, and so on.

For opaque pixels, $f$ is commonly defined to compare the depth value and keep the pixel which is closer to the camera. As we all know, picking the smallest depth

value from multiple pixels can be done out-of-order. However, composition of transparent or semi-transparent objects needs to blend multiple pixels, which in general must follow the depth order either front-to-back or back-to-font; for example, the visual effect of putting a drop of light-pink water above a piece of glass is different from the reversed order. For a series of pixels, therefore, the final value of $f$ is derived from an ordered reduction of individual operations, $f = f_1 \circ f_2 \circ \cdots \circ f_n$. The ordering of $f_1$ through $f_n$ matters, and in general the sequence cannot be permuted without altering the semantics of $f$. Fortunately, although blending operators are not commutative, they are associative: i.e., $f_1 \circ f_2 \circ f_3 \circ f_4 = f_1 \circ f_2 \circ f_3 \circ f_4$ [33]. As we detail in Chapter 6.3, CHOPIN leverages this associativity to compose transparent sub-images asynchronously.

Apart from the reduction function, how pixels are sent to the GPU where the reduction occurs also matters for performance. The simplest communication method is direct-send [99, 157]: once a GPU has finished processing its workload, it begins to distribute the image regions that belong to other GPUs, regardless of the readiness of the destination GPUs. With a large number of GPUs, this can easily congest the network with many simultaneous messages. To address this issue, binary-swap [136, 254] and Radix-k [183] first divide composition processes into multiple groups, and then compose sub-images with direct-send inside each group; to compose all sub-images, several rounds of this procedure are required. Alternately, Sepia [145] and Lightning-2 [227] designed special hardware to accelerate image composition, but this incurs expensive hardware cost.

In contrast, the approach we take in this chapter maintains the simplicity of direct-send, and mitigates network congestion issues via a novel image composition scheduler: within CHOPIN, any two GPUs start composition-related transfers only when they are ready and available.

## 6.2 Limits of Existing Solutions

SFR splits the workload of a single frame into multiple partitions and distributes them among different GPUs. However, individual GPUs must synchronize and exchange information *somewhere* along the rendering pipeline in order to produce the correct final image.

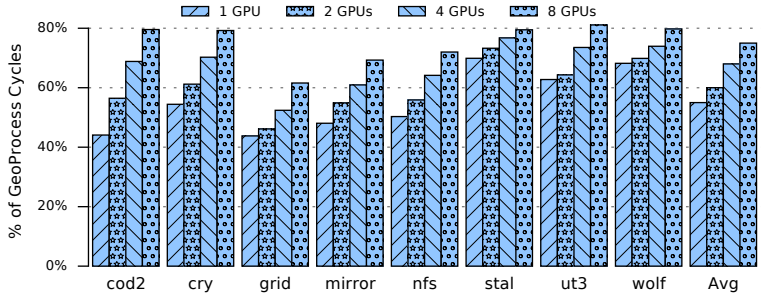



**Figure 6.1:** Percentage of geometry processing cycles in the graphics pipeline of conventional SFR implementation.

Based on where this synchronization happens, SFR implementations can be classified into three categories: sort-first, sort-middle, and sort-last [147]. Sort-first rendering identifies the destination GPUs of each primitive by conducting preliminary transformations at the very beginning of the graphics pipeline to compute the screen coordinates of all primitives, and distributes each primitive to the GPUs that correspond to the primitive's screen coordinates; after primitive distribution, each GPU can run the normal graphics pipeline independently. In contrast, both sort-middle and sort-last distribute primitives without knowing where they will fall in the screen space, and exchange partial information later: sort-middle rendering exchanges geometry processing results *before* the rasterization stage, while sort-last rendering exchanges fragments at the end of the pipeline for final image composition.

Among these three implementations, sort-middle is rarely adopted because the size of geometry processing outcome is very large (hundreds of kilobytes per primitive) [120, 208]. Both CPUs and GPUs have been used for the preliminary transformation in sort-first rendering [70, 102, 103, 148, 166]. Thanks to higher computing and data throughputs, traditional GPU-assisted implementations tend to perform better than CPUs, but they duplicate all primitives in every GPU to amortize the low bandwidth and long latency of traditional inter-GPU links [148, 166]. In these schemes, each GPU maps all primitives to screen coordinates, and eventually drops the primitives that fall outside of its assigned screen region. Unfortunately, this duplicated pre-processing stage is not scalable: as shown in Figure 6.1, redundant

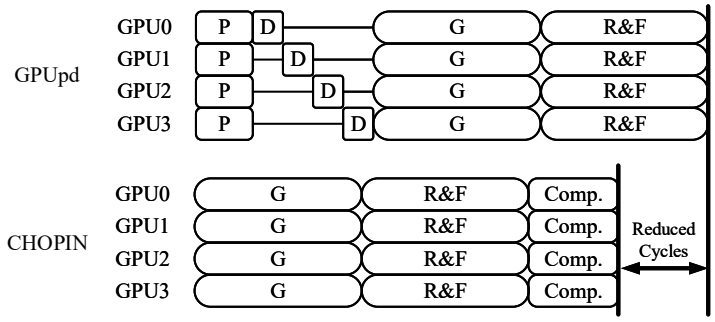



**Figure 6.2:** Graphics pipelines of GPUpd and CHOPIN. (P: Primitive Projection, D: Primitive Distribution, G: Geometry Processing, R: Rasterization, F: Fragment Processing, Comp: Parallel Image Composition.)

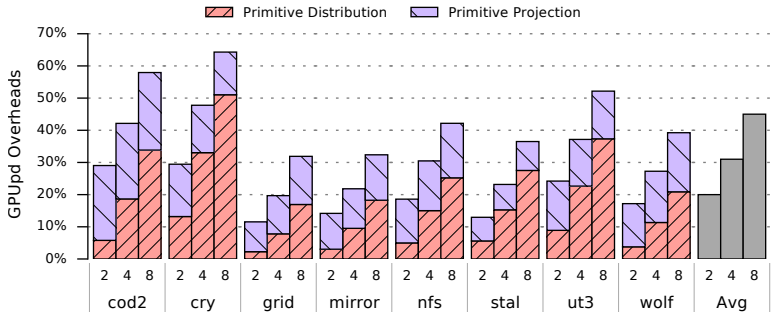



**Figure 6.3:** Percentage of execution cycles of the extra pipeline stages in GPUpd.

geometry processing will dominate the execution cycles of graphics pipeline and severely impact performance as the number of GPUs grows.

To address the problem of redundant computing and take advantage of recent high-performance interconnects, Kim et al. proposed GPUpd [114] pipeline, illustrated in Figure 6.2. GPUpd is a sort-first technique, which evenly distributes all primitives of each draw command across GPUs. All GPUs project the received primitives to screen space. The GPUs then exchange primitive IDs based on projection results through high-performance inter-GPU connections, so that each GPU owns only primitive IDs that fall into its assigned region of screen space. Finally, each GPU executes the full graphics pipeline on its received primitive IDs.

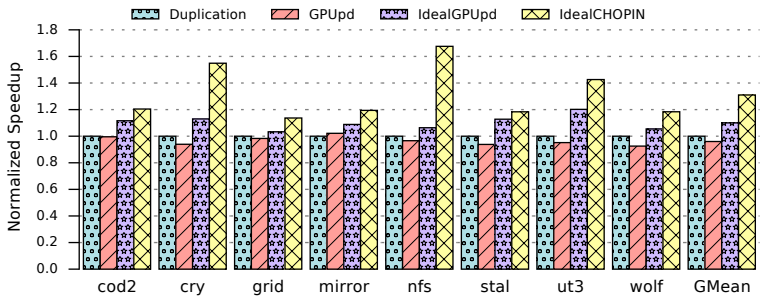



**Figure 6.4:** Potential performance improvement afforded by leveraging parallel image composition.

However, although GPUpd can reduce the overhead of redundant primitive projections, it requires GPUs to distribute primitive IDs *sequentially* to protect the input primitive order; otherwise, a GPU must have large memory to buffer exchanged primitive IDs and complex sorting structure to reorder them. During inter-GPU exchanging, GPU0 first distributes its primitive IDs to other GPUs, then GPU1 distributes its primitive IDs, and this procedure continues until all GPUs have completed primitive distribution. As shown in Figure 6.3, with more GPUs in the system (2–8 GPUs), the sequential inter-GPU primitive distribution becomes a critical performance bottleneck.

## 6.3 CHOPIN: Leveraging Parallel Image Composition

To eliminate the performance overhead of redundant computing and sequential inter-GPU synchronizations, in this chapter, we propose CHOPIN which is a *sort-last* rendering scheme, with a pipeline shown in Figure 6.2.

It first divides consecutive draw commands of each frame into multiple groups based on the draw command property. As for each group, draw commands are distributed across different GPUs. Since each draw command is only executed in a single GPU, CHOPIN is free of the redundant primitive projections that arise in traditional SFR implementations.

At group boundaries, sub-images generated by all GPUs are composed in parallel. For the group of opaque objects, sub-images can be composed out-of-order, because the pixels which are closer to the camera will always win. For the group of

transparent objects, we take advantage of the associativity of image composition described in Chapter 6.1: adjacent sub-images are composed asynchronously as soon as they are available.

However, naïve distribution of draw commands, such as using round-robin, can result in severe load imbalance among the GPUs. CHOPIN therefore relies on a novel draw command scheduler (Chapter 6.4.4) which can dispatch each draw command to a proper GPU based on the dynamic execution state. To mitigate network congestion and avoid unnecessary stalls, we also propose a scheduler for sub-image composition (Chapter 6.4.5), which ensures that any two GPUs can start composition only when their sub-images are ready and neither of them is composing with other GPUs.

Figure 6.4 illustrates the potential of CHOPIN in an ideal system where all intermediate results are buffered on-chip and the inter-GPU links have zero latency and unlimited bandwidth: parallel image composition offers up to 1.68× speedups (1.31× gmean) over the best prior SFR solution. We refer detailed evaluation methodology to Chapter 6.5.

## 6.4 The CHOPIN Architecture

The high-level system architecture of CHOPIN is shown in Figure 6.5, and consists of extensions in both the software and hardware layers.

In the software layer ❶, we divide draw commands into multiple groups. At the beginning and the end of each group, we insert two new Application Programming Interface (API) functions *CompGroupStart()* and *CompGroupEnd()* to start and finish the image composition. We also extend the driver by implementing a separate command list for each GPU.

In the hardware layer, we connect multiple GPUs with high-speed inter-GPU links ❷, similar to NVIDIA DGX system [170, 173], and present them to the OS as a single larger GPU. Draw commands issued by the driver are distributed among the different GPUs by a hardware scheduler ❸. After all draw commands of a single composition group have finished, *CompGroupEnd()* is called to compose the resulting sub-images. An image composition scheduler ❹ orchestrates which pairs of GPUs can communicate with one another at any given time.

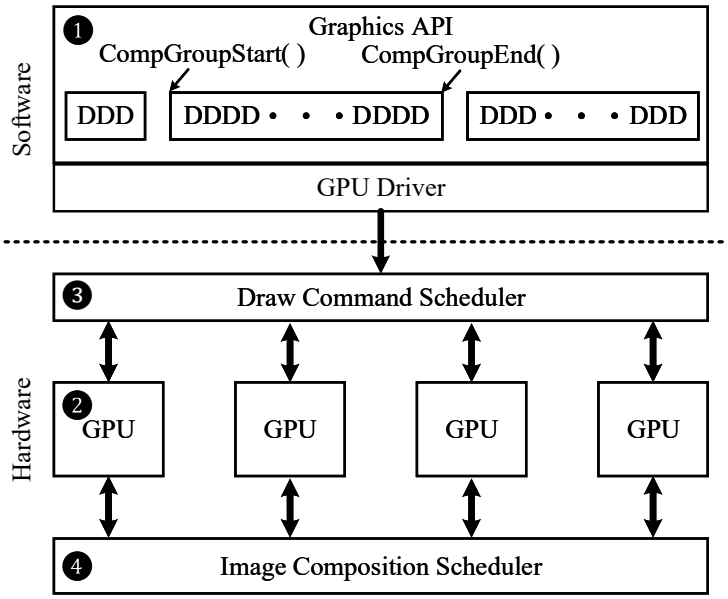



**Figure 6.5:** High-level system overview of CHOPIN (each "D" stands for a separate draw command).

### 6.4.1 Software Extensions

We first explain the semantics of extended graphics API functions. *CompGroupStart()* is called before each composition group starts. This function does the necessary preparations for image composition. It passes the number of primitives and the transparency information to the GPU driver; the GPU driver will then send these data to the GPU hardware. If there are transparent objects in composition group, the GPU driver allocates extra memory for sub-images in all GPUs, because transparent sub-images cannot be messed with the background before composition. When function *CompGroupEnd()* is called, the GPU driver sends a COMPOSE command to each GPU for image composition. The composition workflow is described in detail in Chapter 6.4.3.

The necessity of grouping draw commands is derived from the various properties of each draw command. CHOPIN assumes Immediate Mode Rendering (IMR), so we only group consecutive draw commands in a greedy fashion; however, more

sophisticated mechanisms could potentially reorder draw commands to create larger composition group at the cost of additional complexity. When processing a sequence of draw commands, a group boundary is inserted between two adjacent draw commands on any of the following events:

1. swapping to the next frame,
2. switching to a new render target or depth buffer,
3. enabling or disabling updates to the depth buffer,
4. changing the fragment occlusion test function, or
5. changing the pixel composition operator.

Event 1 is straightforward, because we have to finish the current frame before we move to the next one. Render Targets (RTs) are a feature that allow 3D objects to be rendered in an intermediate memory buffer, instead of the framebuffer (FB); they can be manipulated by pixel shaders in order to apply additional effects to the final image, such as light bloom, motion blur, etc. A depth buffer (or Z Buffer) is a memory structure that records the depth value of screen pixels, and is used to compute the occlusion status of newly incoming fragments. For both, Event 2 is necessary to maintain inter-RT and inter-depth-buffer dependencies, where the computing of future RTs and depth buffers depends on the content recorded in the current one.

In graphics applications, some draw commands check the depth buffer for occlusion verification without updating it. Not inserting a boundary here could allow some fragments to pass the depth test and update the frame buffer by mistake, leading to an incorrect final image. We use Event 3 to create a clean depth buffer before these draw commands begin.

Boundaries at Event 4 are needed because draw commands use depth comparison operators to retain or discard incoming fragments. Since CHOPIN distributes draw commands among multiple GPUs, having multiple comparison functions (e.g., *less-than* and *greater-than*) in a single group can scramble the depth comparison order and lead to incorrect depth verification. A group boundary at Event 4 will

guarantee every time a new comparison function is applied, depth test will start from a correct value.

As described in Chapter 6.1, pixel blending of consecutive draw commands is associative as long as a single blending operator (e.g., *over*) is used. However, the associativity is not transitive across different operators (e.g., mixed *over* and *additive* operators are not associative), and the composition of opaque and transparent objects also cannot be interleaved. Hence, whenever any draw command changes to a new operator (i.e., Event 5), we create a group boundary.

### 6.4.2 Hardware Extensions

Besides inter-GPU communications, the main operations of image composition are (a) reading local sub-image before sending it out and (b) composing pixels in destination GPUs. As both of these functions are carried out by the ROP, they do not require new functional components in CHOPIN.

However, since SFR (Split Frame Rendering) splits 2D screen space into multiple regions and assigns each region to a specific GPU, pixels must eventually be exchanged among GPUs after sub-images are generated, we need a hardware component that computes destination GPUs of individual pixels. We therefore slightly extend the ROPs with a simple structure that distributes pixels to different GPUs according to their screen positions.

We also require a draw command scheduler and an image composition scheduler to address the problems of load imbalance and network congestion, which are two main performance bottlenecks of a naïve implementation of CHOPIN. We describe them in Chapter 6.4.4 and 6.4.5, respectively.

### 6.4.3 Composition Workflow

Figure 6.6 shows the workflow of each composition group. When a composition group begins, we first check how many primitives (e.g., triangles) are included in this group ❶. If the number of primitives is smaller than a certain threshold, we revert to traditional SFR and duplicate all primitives in each GPU ❷. This is a tradeoff between redundant geometry processing and image composition overhead. For example, some draw commands are executed to set up the background before

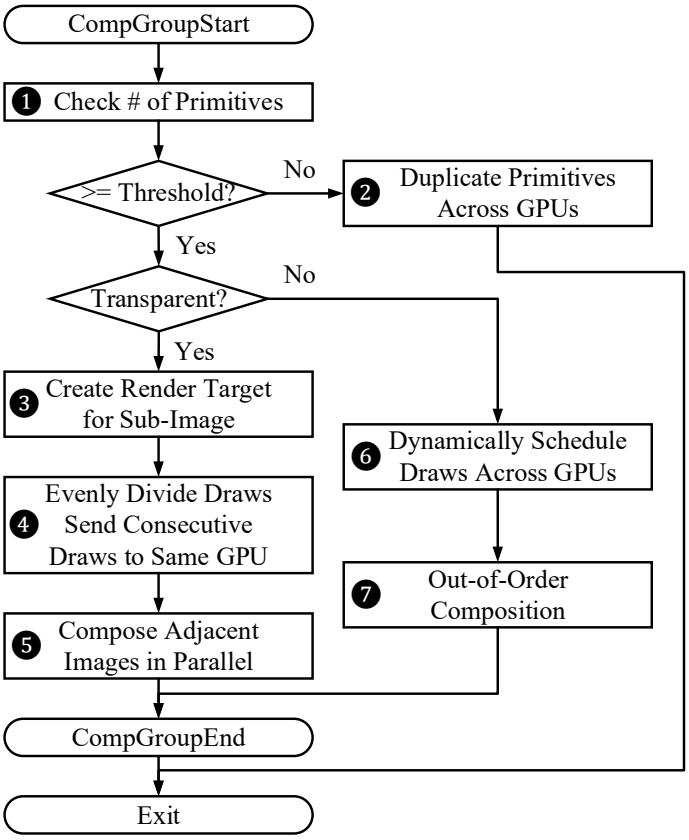



**Figure 6.6:** The workflow of each composition group.

real objects are rendered; because these draw commands simply cut a rectangle screen into two triangles, the geometry processing overhead is much smaller than other graphics pipeline stages, and the overhead of redundant geometry processing is also much smaller than the cost of image composition. Although this threshold is an additional parameter that must be set, our sensitivity analysis (see Figure 6.19) results show that the threshold value does not substantially impact the performance, so this is not a significant concern.

For each composition group that warrants parallel image composition, we first check if the group contains transparent objects. If so, the GPU driver needs to create extra render targets for sub-images in each GPU ❸. This is necessary because

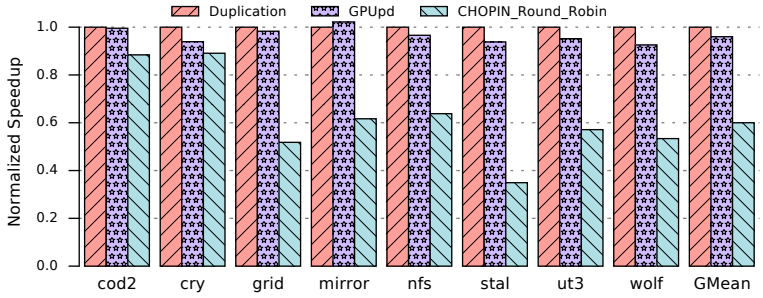



**Figure 6.7:** Performance overhead of round-robin draw command scheduling (normalized to the system which duplicates all primitive across GPUs).

transparent objects cannot be merged with the background until all sub-images have been composed — otherwise, the background pixels will be composed multiple times, creating an incorrect result. To protect the input order of transparent primitives and achieve reasonable load balance at the same time, we evenly divide draw commands and simply distribute the same amount of continuous primitives across GPUs ❹. This simple workload distribution is acceptable because, in current applications, only a small fraction of draw commands are transparent. After a GPU has finished its workload, we can begin to compose adjacent sub-images asynchronously by leveraging associativity ❺.

If the group has no transparent objects, CHOPIN dynamically distributes draw commands with our proposed scheduler ❻; in this case, it is not necessary to create extra render targets because generated sub-images will overwrite the background anyway. Finally, opaque sub-images are composed out-of-order ❼ by simply comparing their distances to the camera (depth value); sub-image pixels which are closer to the camera will be retained for final image composition.

### 6.4.4 Draw Command Scheduler

Although the parallel image composition technique in CHOPIN can avoid sequential inter-GPU synchronizations, the correct final image can only be generated after all sub-images have been composed; therefore, the slowest GPU will determine the overall system performance. As Figure 6.7 shows, simple draw command scheduling, such as round-robin, can lead to severe load imbalance and substantially impact performance.

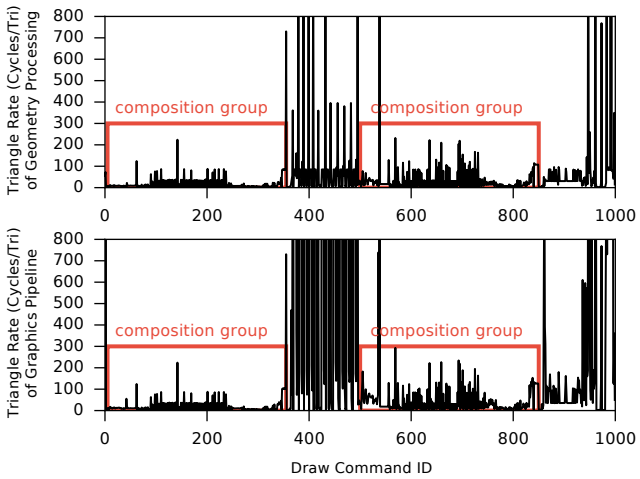



**Figure 6.8:** Triangle rate of geometry processing stage (top) and whole graphics pipeline (bottom). The data is from `cod2`, other applications have the same trend.

To achieve optimal load balance, we would ideally like to know the exact execution time of each draw command; however, this is unrealistic before the draw command is actually executed. Therefore, we need to approximately predict the draw command running time. A complete heuristic for rendering time estimation has been proposed in [240]: $t = c_1 \times \#tv\ c_2 * \#pix$, where $t$ is the estimated rendering time, $\#tv$ is the number of transformed vertices, $\#pix$ is the number of rendered pixels, and $c_1$ and $c_2$ are the vertex rate and pixel rate. Although this heuristic considered both geometry and fragment processing stages, the value of $c_1$ and $c_2$ can change dynamically across draw commands, and we cannot use this approach directly. OO-VR [246] samples these parameters on the first several draw commands and uses them for the remainder of the rendering computation; however, we have found that these parameters vary substantially, and such samples form a poor estimate for the dynamic execution state of the whole system. Other prior work [17] instead uses the triangle count of each draw command (which can be acquired from applications) as a heuristic to estimate rendering time. However, dynamically keeping tracking of all triangles throughout the graphics pipeline is complicated, especially after a triangle is rasterized into multiple fragments.

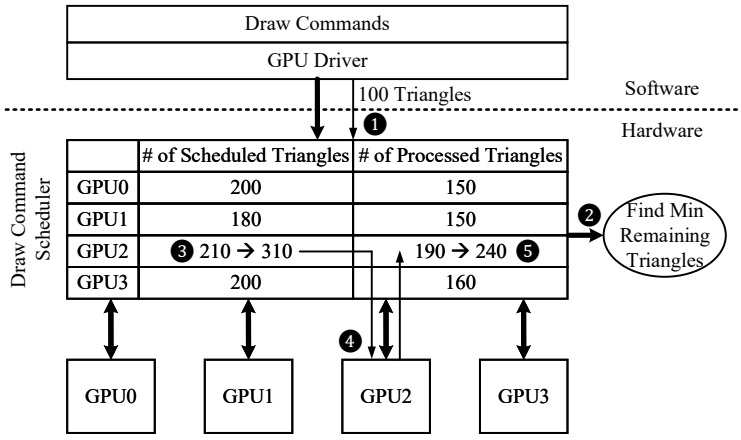
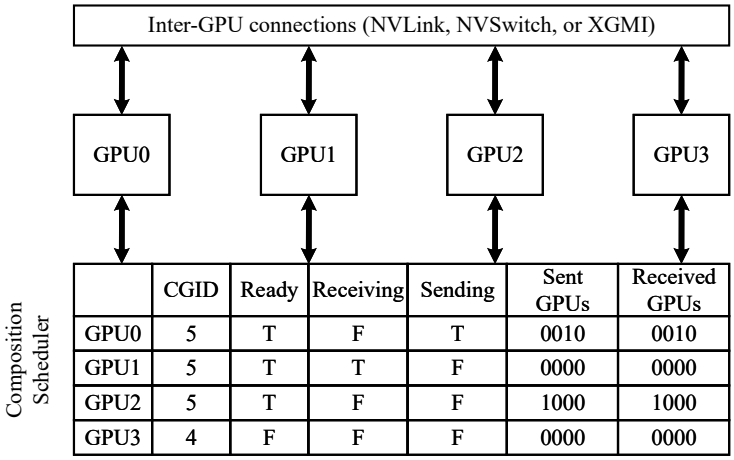



**Figure 6.9:** Draw command scheduler microarchitecture.

Fortunately, as Figure 6.8 shows, the *triangle rate* (i.e., cycles/triangle) of the geometry processing stage is similar to that of the whole graphics pipeline — this is similar to how the instruction processing rate in a CPU frontend limits the performance of the CPU backend. We therefore propose to use the number of remaining triangles in the geometry processing stage as an estimate of each GPU's remaining workload. Every time a draw command is issued by the GPU driver, we simply distribute it to the GPU which has the fewest remaining triangles in geometry processing stage.

The microarchitecture of our draw command scheduler is shown in Figure 6.9. The main structure is a table, in which each GPU has an entry to record the number of scheduled and processed triangles in that GPU; the remaining triangle count is the difference. The scheduled triangle count increments when a draw command is scheduled to a GPU, while the processed count increments as triangles finish geometry processing.

Figure 6.9 also shows a running example of how the scheduler operates: first, the GPU driver issues a draw command with 100 triangles ❶. Next, the draw command scheduler finds that GPU2 currently has the fewest remaining triangles ❷. The triangle count of this draw command is therefore added to the number of triangles scheduled to GPU2 ❸, while the scheduler distributes this draw command to GPU2 ❹.




| | CGID | Ready | Receiving | Sending | Sent GPUs | Received GPUs |
|---|---|---|---|---|---|---|
| GPU0 | 5 | T | F | T | 0010 | 0010 |
| GPU1 | 5 | T | T | F | 0000 | 0000 |
| GPU2 | 5 | T | F | F | 1000 | 1000 |
| GPU3 | 4 | F | F | F | 0000 | 0000 |

**Figure 6.10:** Image composition scheduler microarchitecture.

| Field | Meaning |
|---|---|
| CGID | Composition Group ID |
| Ready | Ready to compose with others? |
| Receiving | Receiving pixels from another GPU? |
| Sending | Sending pixels to another GPU? |
| SentGPUs | GPUs the sub-image has been sent to |
| ReceivedGPUs | GPUs we have composed with |

**Table 6.1:** Fields tracked by image composition scheduler.

Once the triangles pass through the geometry processing stage in graphics pipeline, the number of processed triangles for GPU2 is increased accordingly ❺.

### 6.4.5 Image Composition Scheduler

Once each GPU has finished its workload, it can begin to communicate with other GPUs for sub-images composition. However, blind inter-GPU communication can result in the congestion and under-utilization of interconnect resources (see Chapter 6.1). The most straightforward scheme, direct-send, sends the screen regions to any other GPUs without knowing if the destination GPU can accept it; if the target GPU is still computing, the waiting inter-GPU messages will blocked the interconnect. For example, assuming a situation where all GPUs except GPU0

have finished their draw commands, so GPUs begin to send their sub-images to GPU0. Because GPU0 is still running, inter-GPU messages will be blocked in the network. Even though GPUs could have communicated with another GPU rather than GPU0, now they have to wait until GPU0 is able to drain the network. Therefore, an intelligent scheduling mechanism for image composition is necessary.

Our proposed composition scheduler, shown in Figure 6.10, aims to avoid stalls due to the GPUs that are still running their scheduled draw commands or busy composing with other GPUs. It records the composition status (Table 6.1) of each GPU in a table: the composition group ID (CGID) is used to distinguish different groups, the Ready flag is set while a GPU generated its sub-image and became ready to compose with others, and the Receiving and Sending flags are used to indicate that a GPU is busy exchanging pixels with another GPU. Finally, SentGPUs and ReceivedGPUs record the GPUs with which a GPU has already communicated in a bit vector, vector size is same as the number of GPUs in the system.

Figure 6.11 shows the image composition scheduler workflow. Once a GPU has finished all draw commands and generated a sub-image, we set its Ready flag and increment the CGID by one to start a new composition phase ❶. We then check the status of other GPUs to see if any available GPUs can compose with each other. For groups of transparent objects ❷, only adjacent GPUs are checked because transparent sub-images cannot be composed entirely out-of-order (Chapter 6.1); for opaque groups, all GPUs are checked❸. Composition starts only if the remote GPU (1) is ready to compose and running in the same composition group (i.e., CGIDs are same), (2) has not yet been composed with (i.e., not set in ReceivedGPUs), and (3) is not sending pixels to another GPU.

As an example, consider the status of Figure 6.10. We can see that GPU0 and GPU2 have composed with each other, GPU3 is still running, and GPU1 just finished its workload and set its Ready flag. At this moment, GPU1 can compose with GPU0, so we set the Receiving flag of GPU1 and the Sending flag of GPU0 to indicate that these two GPUs are busy ❹. When image composition starts, GPU0 will read its sub-image and send out the region corresponding to GPU1. After these two GPUs have finished composition, we will reset the Receiving flag of GPU1 and the Sending flag of GPU0. Meanwhile, we will also add GPU0 into the ReceivedGPUs field of GPU1 and add GPU1 into the SentGPUs field of GPU0 ❺. This procedure

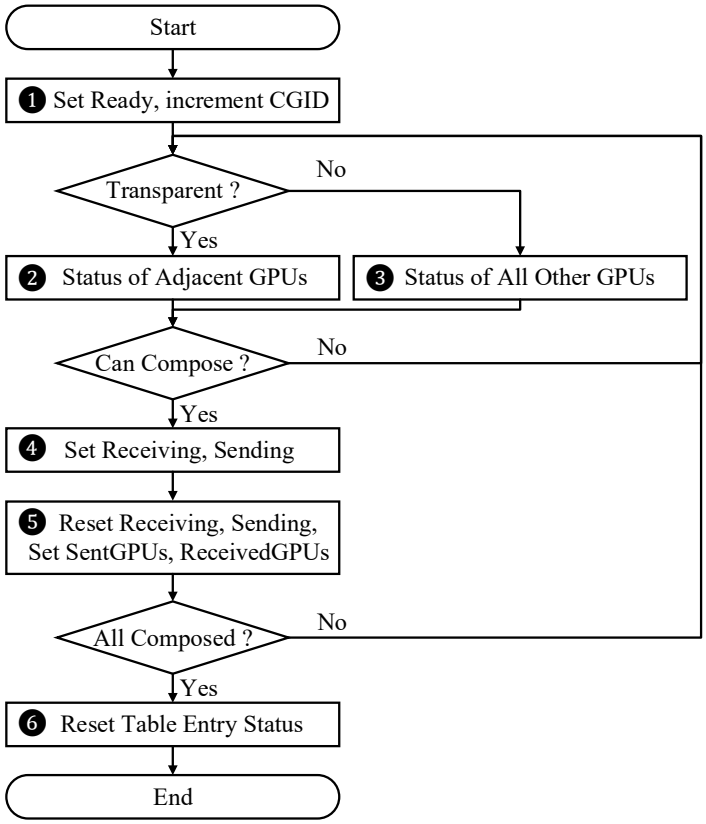



**Figure 6.11:** Image composition scheduler workflow.

is repeated until all sub-images are composed. Finally, we reset the table entry after a GPU has sent its sub-image to all other GPUs and the sub-images of all other GPUs has also been received ❻. The composition is finished once each GPU has composed with all other GPUs and, for transparent sub-images, the background.

## 6.5 Methodology

We evaluate CHOPIN by extending ATTILA [63, 151], a cycle-level GPU simulator which implements a wide spectrum of graphics features present in modern GPUs. Unfortunately, the latest ATTILA is designed to model an AMD TeraScale2 architecture [142], and is hard to configure as the latest NVIDIA Volta [172] or

| Structure | Configuration |
|---|---|
| GPU frequency | 1GHz |
| Number of GPUs | 8 |
| Number of SMs | 64 (8 per GPU) |
| Number of ROPs | 64 (8 per GPU) |
| SM configurations | 32 shader cores, 4 texture units |
| L2 Cache | 6MB in total |
| DRAM | 2TB/s, 8 channels<br>8 banks per channel |
| Composition group # primitives threshold | 4096 |
| Inter-GPU bandwidth | 64GB/s (uni-directional) |
| Inter-GPU latency | 200 cycles |

**Table 6.2:** Simulated GPU and memory hierarchy for CHOPIN.

| Benchmark | Abbr. | Resolution | # Draws | # Triangles |
|---|---|---|---|---|
| Call of Duty 2 | cod2 | $640 \times 480$ | 1005 | 219,950 |
| Crysis | cry | $800 \times 600$ | 1427 | 800,948 |
| GRID | grid | $1280 \times 1024$ | 2623 | 466,806 |
| Mirror's Edge | mirror | $1280 \times 1024$ | 1257 | 381,422 |
| Need for Speed: Undercover | nfs | $1280 \times 1024$ | 1858 | 534,121 |
| S.T.A.L.K.E.R.: Call of Pripyat | stal | $1280 \times 1024$ | 1086 | 546,733 |
| Unreal Tournament 3 | ut3 | $1280 \times 1024$ | 1944 | 630,302 |
| Wolfenstein | wolf | $640 \times 480$ | 1697 | 243,052 |

**Table 6.3:** Benchmarks used for CHOPIN evaluation.

Turning [164] systems; therefore, to fairly simulate the performance of different SFR implementations, we scale down system parameters, such as the number of SMs and ROPs, accordingly (Table 6.2). Similar simulation strategies have been widely used in related prior work [244, 245, 246, 247]. We extend the GPU driver for issuing draw commands and hardware register values to different GPUs. Similar to existing NVIDIA DGX system [170, 173], we model the inter-GPU links with point-to-point connections between GPU pairs, with a default bandwidth and latency of 64GB/s and 200 cycles.

As benchmarks, we use eight single-frame traces as shown in Table 6.3, which we manually annotate to insert the new API functions *CompGroupStart()* and

*CompGroupEnd()* at composition group boundaries. All benchmarks come from the real-world games; the number of draw commands and the number of primitives (triangles) and the target resolutions vary across the set, and are shown in Table 6.3.

Our SFR implementation splits each frame by interleaving $64 \times 64$ pixel tiles to different GPUs. Unlike AFR, SFR needs to handle the read-after-write dependencies on render targets and depth buffers. To ensure memory consistency, every time the application switch to a new render target or depth buffer, our simulation invokes an inter-GPU synchronization which requires each GPU to broadcast the latest content of their current render targets and depth buffers to other GPUs.

Apart from our CHOPIN system, we also implement primitive duplication, which we use as the baseline. We also implement the best prior work GPUpd [114], modelling both optimizations: batching and runahead execution.[1] To explore the upper bound on the performance of each technique, we also idealize GPUpd and CHOPIN in the same way: unlimited on-chip memory for buffering intermediate results, zero inter-GPU latency, and infinite inter-GPU bandwidth.

## 6.6 Evaluation Results

In this section, we first compare the performance of CHOPIN, primitive duplication, and GPUpd. Then we conduct sensitivity analysis to explore the design space, and finally evaluate the hardware costs.

### 6.6.1 Performance Analysis

The overall performance of multiple SFR implementations is shown in Figure 6.12. The performance of GPUpd is comparable to conventional primitive duplication. Idealization of GPUpd (i.e., our best implementation of GPUpd) can slightly improve the performance, but it's still substantially worse than CHOPIN. With the image composition scheduler enabled, CHOPIN works 25% (up to 56%) better than primitive duplication, and only 4.8% slower than IdealCHOPIN.

Figure 6.13 shows that the performance improvement of CHOPIN comes mainly from the reduced synchronization overheads: for GPUpd, this is the extra primitive

[1]We contacted the authors to request the GPUpd sources, but were denied because of IP issues; we therefore created a best-effort realistic implementation of GPUpd as well as an idealized variant.

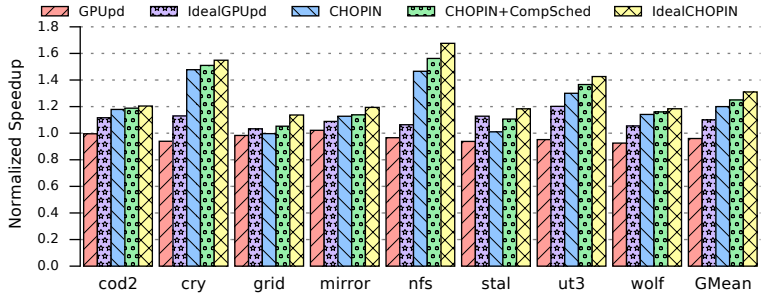



**Figure 6.12:** Performance of an 8-GPU system, baseline is primitive duplication with configurations of Table 6.2. (CompSched: composition scheduler)

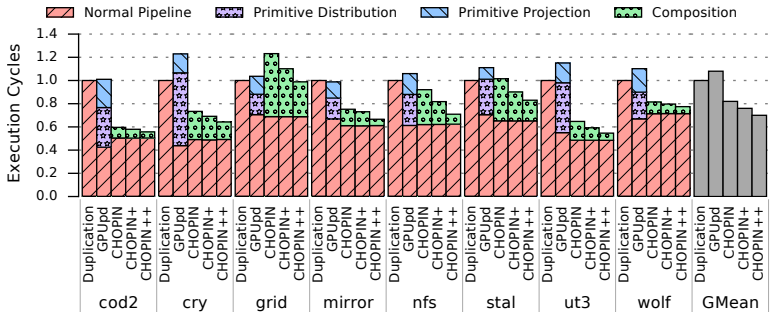



**Figure 6.13:** Execution cycle breakdown of graphics pipeline stages, normalize all results to the cycles of primitive duplication. (CHOPIN+: CHOPIN+composition scheduler, CHOPIN++: IdealCHOPIN)

projection and distribution stages, while for CHOPIN this is the image composition stage (e.g., the composition overhead of `grid` is large because it has much bigger inter-GPU traffic load, see Figure 6.14). Conventional primitive duplication suffers because of redundant geometry processing, which CHOPIN entirely avoids. Even though GPUpd still performs some redundant computation in the primitive projection stage, sequential inter-GPU primitive is its critical performance bottleneck.

CHOPIN avoids redundant geometry processing by distributing each draw command to a specific GPU, and substantially reduces the overhead of inter-GPU synchronization through parallel composition. With the image composition scheduler, the composition cost is reduced even more by avoiding unnecessary network congestion.

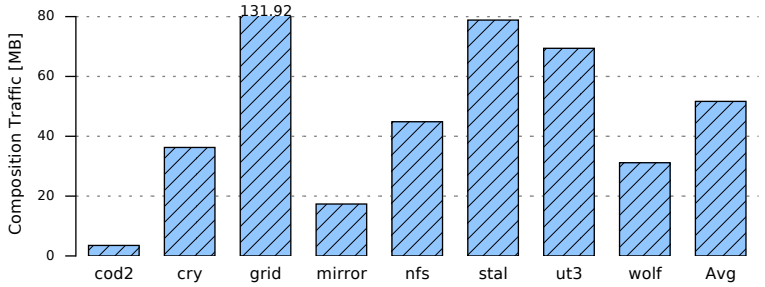



**Figure 6.14:** Traffic load of parallel image composition.

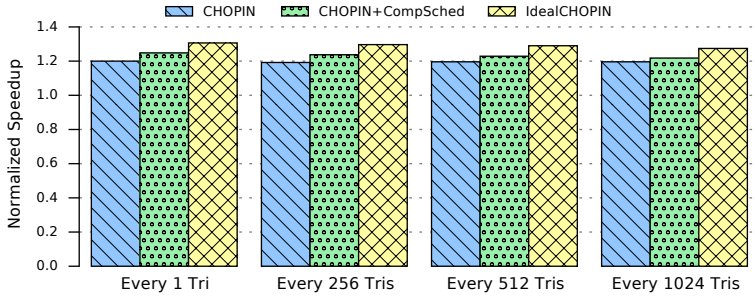



**Figure 6.15:** Performance sensitivity to the frequency of updates sent to draw command scheduler (baseline is primitive duplication with configurations from Table 6.2).

Distributing draw commands to different GPUs can potentially reduce the effectiveness of occlusion testing, because sub-images might not have the smallest depth value for each pixel before image composition. However, we find that the impact is minor: the number of processed fragments in ROPs only increases by 3.6%, 6.5%, and 8.4% in systems of 2, 4, and 8 GPUs, which still permits speedups up to 1.56×.

### 6.6.2 Composition Traffic Load

To reduce network traffic, CHOPIN only exchanges screen regions assigned to the GPUs that are communicating at any given moment. We also filter out the screen tiles that are not rendered by any draw commands, as they do not need to be composed. As Figure 6.14 shows, the average traffic load of image composition is only 51.66MB. Figure 6.13 shows that this does not create a substantial execution

overhead, especially with the image composition scheduler enabled. The large traffic load in `grid` is due to many large triangles that cover big screen regions; we leave optimizing this to future work.

In our experiments, we allow the GPUs to update the draw command scheduler statistics for every triangle processed, an average of 1.7MB traffic with 4B message size. To account for scaling to much larger systems and much larger triangle counts, however, we also investigated how a larger update interval would affect the performance of CHOPIN. Figure 6.15 sweeps this update frequency from every triangle to every 1024 triangles on an otherwise identical system; the average performance improvement of CHOPIN drops very slightly from 1.25× to 1.22×. With updates every 1024 triangles and 4B messages, the total update traffic load would be 4KB for 1 million triangles and 4MB for 1 billion triangles. The image composition scheduler receives notifications from GPUs at composition boundaries that they are ready to accept work, and sends notifications back to GPUs — 7 requests and 7 responses for each GPU in an 8-GPU system, plus an 8th pair to compose with the background — which results in $\left(8\ 8\right) \times 8 \times 4 = 512B$ with 4B messages. Both are negligible compared to sub-image frame content.

### 6.6.3 Sensitivity Analysis

To understand the relationship between our architectural parameters and the performance of CHOPIN, we performed sensitivity studies across a range of design space parameters.

**GPU count.** Even though integrating more GPUs in a system can provide abundant resources to meet the constantly growing computing requirements, it also can impose bigger challenge on inter-GPU synchronizations. As Figure 6.16 shows, GPUpd is constrained by the sequential primitive distribution, and performance does not scale with GPU count. In contrast, because CHOPIN parallelizes image composition, the inter-GPU communication is also accelerated with more GPUs. Therefore, the performance of CHOPIN is scalable and the improvement versus prior SFR solutions grows as the number of GPUs increases. Meanwhile, the image composition scheduler becomes more effective while GPU count is bigger: this is because naïve inter-GPU communication for image composition can congest the

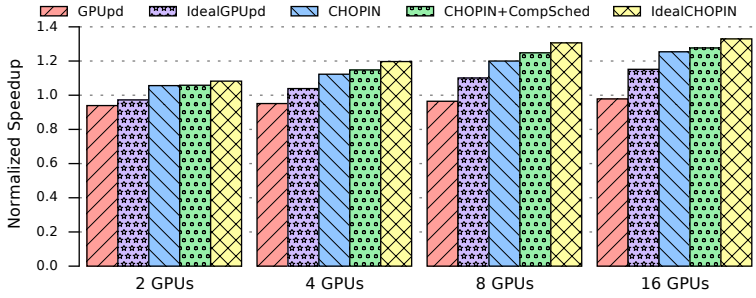



**Figure 6.16:** Performance sensitivity to the number of GPUs (for each GPU count configuration, baseline is primitive duplication with the same GPU count and other settings as in Table 6.2).

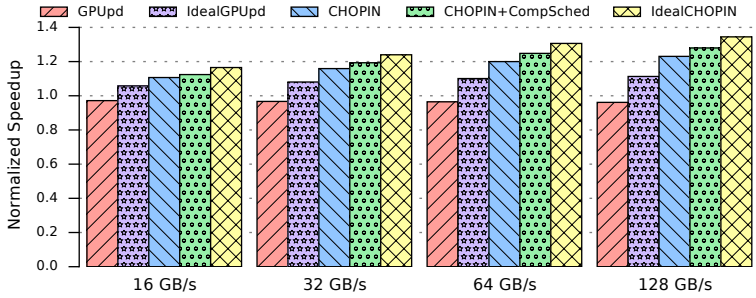



**Figure 6.17:** Performance sensitivity to inter-GPU link bandwidth (baseline is primitive duplication with configurations of Table 6.2).

network more frequently with more GPUs, which is a bigger bottleneck for a larger system.

**Inter-GPU link bandwidth and latency.** Since inter-GPU synchronization relies on the inter-GPU interconnect, we investigated sensitivity to link bandwidth and latency. CHOPIN performance scales with bandwidth (Figure 6.17), unlike GPUpd. Similarly, CHOPIN is not significantly affected by link latency (Figure 6.18), unlike GPUpd where latency quickly bottlenecks sequential primitive exchange.

**Composition group size threshold.** This parameter makes a tradeoff between the redundant geometry processing and the image composition overhead: if the number of primitives inside a composition group is smaller than a specific threshold, CHOPIN reverts to primitive duplication (see Figure 6.6). In theory, this threshold could be important: if set too small, it might not filter out most composition groups

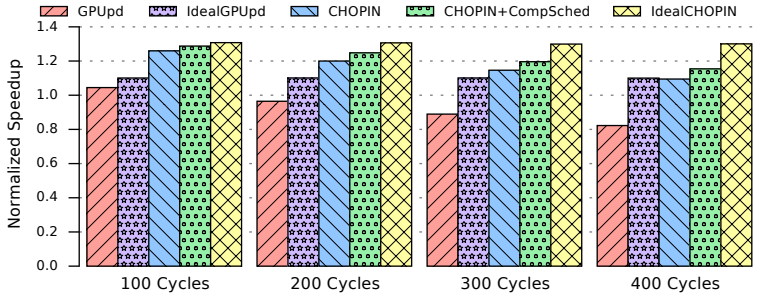



**Figure 6.18:** Performance sensitivity to inter-GPU link latency (baseline is primitive duplication with configurations of Table 6.2).

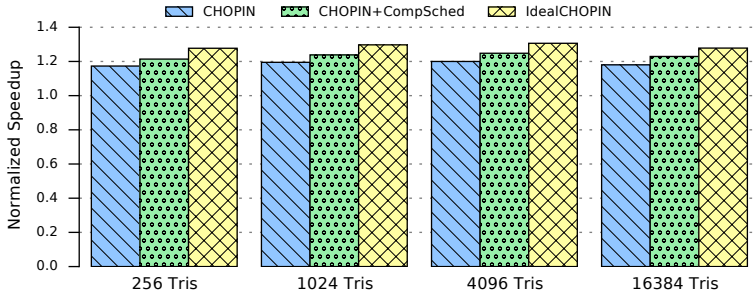



**Figure 6.19:** Performance sensitivity to the threshold of composition group size (baseline is primitive duplication with configurations of Table 6.2).

with few primitives; and if set too big, we can lose the potential performance improvement of parallel image composition. However, as Figure 6.19 shows, it turns out that the performance of CHOPIN is not very sensitive to the configuration of threshold value, and the threshold should be of little concern to programmers.

The main reason for the lack of sensitivity is that the statistic distribution of composition group sizes is bipolar: most composition groups have either a large number of triangles (e.g., consecutive draw commands for objects rendering) or very few triangles (e.g., background draw commands), and many threshold settings will separate them. For example, if the threshold is set as 4,096, CHOPIN will accelerate 6.5 composition groups on average, but those groups cover 92.44% of the triangles in the entire application. Enlarging the threshold to 16,384 will accelerate 5.25 composition groups, and cover 89.83% triangles on average.

### 6.6.4 Hardware Costs

The draw command scheduler and image composition scheduler are the main hardware cost of CHOPIN system. In an 8-GPU system, both schedulers have 8 entries. Each entry of draw command scheduler has two fields: the number of scheduled triangles and the number of processed triangles. To cover the requirements of most existing and future applications, we conservatively allocate 64 bits for each field. Therefore, the total size of draw command scheduler is 128 bytes.

As discussed in Chapter 6.6.3, with a group size threshold of 4,096, up to 13 (6.5 on average) draw command groups will trigger image composition, so we assume one byte is enough to represent a CGID for the image composition scheduler. The Ready, Receiving and Sending flags are all single bits. SentGPUs and ReceivedGPUs are bit vectors with as many bits as the number of GPUs in the system (for us, one byte). Therefore, the total size of the image composition scheduler in our implementation is 27 bytes.

### 6.6.5 Discussion

Rendering workloads scale in two ways: with resolution and with triangles per pixel. As resolution increases to 4K and beyond, triangle counts at iso-quality increase as well. As visual quality advances, however, triangle counts *per pixel* also increase — each frame of the latest game trends to have millions or billions of triangles which sizes are often the same as pixels [9]. This means that the performance overhead of redundant geometry processing and sequential inter-GPU primitive exchange in prior SFR solutions will increase, and the corresponding benefits of CHOPIN will grow.

As is, CHOPIN is applicable to NVIDIA DGX-scale systems. Systems which are significantly larger than this (e.g., 1024 GPUs [85]) may need more complicated rendering mechanisms, such as the combination of AFR and SFR.

## 6.7 Summary

In this chapter, we introduce CHOPIN, a novel architecture for split frame rendering in multi-GPU systems. CHOPIN is a sort-last rendering scheme which distributes each draw command to a specific GPU and avoids redundant geometry processing.

By taking leveraging the parallelism of image composition and modern high-speed inter-GPU links, CHOPIN also obviates the need for sequential inter-GPU communication.

CHOPIN includes two novel schedulers: a draw command scheduler to address load imbalance, and an image composition scheduler to reduce network congestion. All in all, CHOPIN outperforms the best prior work by up to 1.56× (1.25× gmean); and in contrast to existing solutions, scales as the number of GPUs grows.

# Chapter 7

# Related Work

This chapter discusses related work for this dissertation. We introduce the commonly enforced memory consistency models of CPUs and GPUs in Chapter 7.1. We survey different cache coherence protocols in Chapter 7.2. Chapter 7.3 describes the related work on transactional memories. In Chapter 7.4, we talk about previous proposals of graphics processing.

## 7.1 Work Related to Memory Consistency Enforcement

**Strong Consistency in GPUs.** Hechtman and Sorin first made the case that the performance impact of Sequential Consistency (SC) is likely small in GPUs [94]. Singh et al [220] observed that, while this was true for most workloads, some suffered severe penalties with SC because of read-only and private data; they proposed to classify these accesses at runtime and permit reordering while maintaining SC for read-write shared data. Relativistic Cache Coherence (RCC) approach is orthogonal: we focus on SC stall latency, and improve performance for both read-write and read-only data. Both [94] and [220] used a CPU-like setup with MESI and write-back L1 caches. In GPUs, however, write-through L1s perform better [221]: GPU L1 caches have very little space per thread, so a write-back policy brings infrequently written data into the L1 only to write it back soon afterwards. Commercial GPUs have write-through L1s [21, 165, 167]. RCC studies GPU-style write-through L1 caches, and compares against the best prior GPU implementation of weak consistency [221].

**Weak Consistency in GPUs.** Although above work argued to enforce SC in GPUs and they found that relaxed memory models could not significantly outperform SC, modern GPU products still enforce relaxed memory models, as weak models allow for more microarchitectural flexibility and arguably better performance/power/area tradeoffs. Many previous work has aimed to enforce relaxed consistency in GPUs with different coherence optimizations [18, 95, 117, 216]. In a pushing towards generality, GPU vendors have changed from conventional bulk-synchronous towards scoped memory models [2, 112, 171]. Both NVIDIA [135] and AMD [98] have published their formalized scoped memory models. We optimize HMG by leveraging the recent formal definitions of scoped memory model [98, 135] and provide efficient coherence support for multi-GPU system. Sinclair et al [216] adapted DeNovo [59] to GPUs with DRF-0 and HRF variants, and argued that the benefits of HRF over DRF-0 do not warrant the additional complexity of scopes. However, their evaluation was conducted within a single GPU, the latency gap between the broadest and narrowest scope is an order of magnitude larger in multi-GPU environments. Meanwhile, DeNovo requires software to expose additional details to the coherence hardware, while HMG requires no software changes.

**Strong Consistency in CPUs.** Many quills have been sacrificed to argue that sequential consistency is desirable in CPUs and propose how it could be efficiently implemented [12, 36, 50, 79, 81, 82, 84, 89, 131, 194, 219, 238]. Generally, speculation support or other hardware modifications are required to overcome the overheads of SC. Lin et al [131] and Gope et al [84] also used logical order to enforce SC in a CPU setting. RCC shares the conviction that sequential consistency is preferred, but focus on GPUs, which have different architectural constraints (e.g., no speculation support).

**Weak Consistency in CPUs.** Even though lots of academic work has been proposed to support SC in CPUs, almost all industrial vendors choose to relax memory order constraints for performance improvement. The widely employed memory consistency models in industry include Total Store Ordering (TSO) [181], Partial Store Ordering (PSO) [225], Release Consistency (RC) [78], ARMv8 [26], IBM Power [205], and so on; all of them made different tradeoffs about the performance, complexity, and programmability. We have more detailed discussions about memory consistency models in Chapter 2.3.

## 7.2 Work Related to Cache Coherence Protocol

**GPU Coherence.** We have discussed most of the existing GPU coherence protocols in Section 5.3. Besides them, Singh et al [221] proposed a GPU coherence protocol based on physical timestamps, and showed that MESI and write-back caches suffered NoC traffic and performance penalties in GPUs. While the consistency model is weak throughout, the base version (TC-STRONG) can support SC if the core does not permit multiple outstanding memory operations from one warp; we use this SC variant as a baseline of RCC. The improved version (TC-WEAK) cannot support SC, but offers 30% better performance; we use this as a comparison of RCC. RCC uses logical rather than physical timestamps, has lower complexity, and closes the performance gap between SC and relaxed memory model. However, all this work considered neither architecture hierarchy nor scoped memory model. In contrast, HMG explores the coherence support for future deeply hierarchical GPU systems with scoped memory model enforcement.

**Timestamp-Based Cache Coherence.** Nandy and Narayan [156] first observed that timestamps can reduce interconnect traffic due to invalidate messages in MSI-like protocols, but their protocol did not support SC. Shim et al [211] proposed LCC, a sequentially consistent library protocol, for multi-cores; LCC is equivalent to our TC-STRONG baseline. Singh et al [221] adapted LCC to GPUs and proposed a higher-performance weakly ordered variant with a novel fence completion mechanism; Kumar et al [119] used TC-WEAK for FPGA accelerators. Recently, Yao et al [250] adapted TC-WEAK to multi-cores by tracking writes with a Bloom filter. All of these protocols use physical timestamps, and SC variants must stall stores (and weak variants must stall fences) until completion; RCC uses logical time and stalls neither stores nor fences.

Lamport [122] first observed that consistency need only be maintained in logical time. This fact has been used to implement coherence on a logically ordered bus (e.g., [124, 223]) and to extend snooping coherence protocols to non-bus interconnects [14, 138]. Meixner and Sorin used logical timestamps to dynamically verify consistency models [139]. Yu et al [255] proposed using logical timestamps to directly implement coherence in CPU-style multi-cores, but maintains exclusive write states and recall/downgrade messages that RCC wishes to avoid to reduce

store latencies. At the same time, architectural features not present on GPUs (e.g., speculative execution) are required by [257] to support a timestamp speculation scheme. RCC shares the notion of keeping coherence with logical timestamps, but eschews exclusive states to focus on reducing store latencies. RCC is a simpler protocol that offers best-in-class performance in GPUs.

**Hierarchical Cache Coherence.** Coherence hierarchy has been commonly employed in CPUs [237, 239]. Most hierarchical CPU designs [80, 86, 88, 126, 153] have adopted MESI-like coherence, which has been proven to be a poor fit for GPUs [95, 221]. HMG shows that the complexity of extra states is also unnecessary for hierarchical GPUs. Both DASH [126] and WildFire [88] increased the complexity even more by employing the mixed coherence policy: intra-cluster snoopy coherence and inter-cluster directory-based coherence. To implement consistency model efficiently, Alpha GS320 [80] separated the commit events to allow time-critical replies to bypass inbound requests without violating memory order constraints. HMG can achieve almost optimal performance without such overheads.

**Heterogeneous Cache Coherence.** Shared data synchronization in the unified memory space of heterogeneous systems also requires efficient coherence protocols. Lowe-Power et al. proposed a heterogeneous system coherence for integrated CPU-GPU systems [188]. It replaced the standard directory with a region directory to reduce the bandwidth bottleneck of GPU memory requests. Projects such as Crossing Guard [179] and Spandex [19] proposed flexible coherence interfaces to integrate heterogeneous computing devices. We expect that HMG would be integrated nicely with such schemes due to its simple states and clear coherence hierarchy.

## 7.3 Work Related to Transactional Memory

**GPU Transactional Memory (TM).** To date, all hardware TM proposals for GPUs have been based on KiloTM [77]; this system combines lazy version management with lazy, value-based conflict detection. Follow-up work [76] extended KiloTM with an intra-warp conflict detection mechanism and a silent-commit filter for read-only transactions based on physical timestamps. A later proposal [55] added global broadcast updates about currently committing transactions, and leveraged this

to pause or abort doomed transactions; we use an idealized version of this as one of our baselines. GPU-LocalTM [234] is a limited form of transactional memory that guarantees atomicity only within a core's scratchpad; Bloom filters [35] are used for conflict detection. Software TM proposals for GPUs have used either per-object write locks [48] or combined value-based detection with TL2-like timestamp approach [248]. Given special DRAM subarrays [209], and at the cost of substantial memory overheads and extensive OS/software changes, GPU snapshot isolation [56] can reduce abort rates in long transactions by buffering many concurrent memory states; it retains two-round-trip lazy validation and must update snapshot versions in DRAM, resulting in even longer commit latencies.

**CPU Hardware TM.** Since hardware TM was first proposed [96, 228], many CPU implementations have been proposed. Many leverage the existing inter-core coherence mechanism to identify conflicts, either by modifying the coherence protocol [51, 60, 66, 233], adding extra bits to the coherence state [37, 149, 202], or leveraging coherence to update read/write signatures [46, 144, 252]. Existing GPU coherence proposals, however, cannot support eager TM: they either rely on special language-level properties [216], eschew write atomicity [221], or cannot support detecting conflict times [195]. Other TM proposals [49, 51, 89, 116, 189, 236] rely on signature or update broadcasts, or on software-assisted detection [212, 213, 214]. In contrast to TM, speculative lock elision can run parallel code in lock-free manner without requiring instruction set changes, coherence protocol extensions, or programmer support [191].

**Timestamp-Based TM.** Transactional memory schemes based on logical clocks share commonalities with timestamp-based approaches. These have been used mainly in software TMs to maintain consistency [74]; hardware TMs have leveraged them to maintain fairness and forward progress [23, 149, 192], snapshot isolation [133], and in prior GPU work to avoid validation of read-only transactions [76].

## 7.4 Work Related to Graphics Processing

**Graphics Processing in Multi-GPU Systems.** GPUpd [114] and OO-VR [246] are two multi-GPU proposals that attempt to leverage modern, high-speed inter-GPU connections. However, as discussed in Section 6.2, GPUpd is bottlenecked by

a sequential inter-GPU primitive exchange step, while CHOPIN composes sub-images in parallel. OO-VR is a rendering framework to improve data locality in VR applications, orthogonal to our problem of efficient image composition for Split Frame Rendering (SFR). Unlike OO-VR, the draw command distribution in CHOPIN does not rely on statically computed parameters; CHOPIN also includes an image composition scheduler to make full use of network resources.

NVIDIA's SLI [166] proposed attempts to balance the workload by dynamically adjusting how the screen is divided among GPUs. However, it still duplicates all primitives in every GPU, and incurs the attendant overheads. Both DirectX 12 [7] and Vulkan [6] expose multi-GPU hardware to programmers via Application Programming Interface (API), but relying only on this would require programmers to have exact static knowledge of the workload (e.g., workload distribution). CHOPIN can simplify programming and deliver reliable performance through dynamic scheduling in hardware.

**Parallel Rendering Frameworks.** Most SFR mechanisms were originally implemented for PC clusters. Among these implementations, WireGL [102], Chromium [103], and Equalizer [70] are high-level APIs which can allocate workload among machines based on different configurations. However, when the system is configured as sort-first, they use CPUs to compute the destinations of each primitive, and performance is limited by the poor computation throughput of CPUs. When the system is configured as sort-last, they assign one specific machine to collect all sub-images from others for composition, which again constitutes a bottleneck. In contrast, CHOPIN distributes draw commands to different GPUs based on dynamic execution state, and all GPUs in the system contribute to image composition in parallel.

To accelerate image composition, some implementations, like PixelFlow [146] and Pixel-Planes 5 [75] even implemented application specific hardware, with significant overheads. CHOPIN simply takes advantage of existing multi-GPU system and high-performance inter-GPU links, and incurs very small hardware costs. RealityEngine [15] and Pomegranate [71] aim to improve load balancing by frequently exchanging data before geometry processing, before rasterization, and after fragment processing; however, these complicated synchronization patterns are hard to coordinate, and huge traffic load can be imposed on inter-GPU links.

**Graphics Processing in Single-GPU and Mobile GPU Systems.** Besides parallel rendering, lots of work has also been done for graphics processing in single GPU or mobile GPU systems. By leveraging the similarity between consecutive frames, Arnau et al. proposed to use fragment memorization to filter out redundant fragment computing [27]. Rendering Elimination [25] shares the same observation of similarity, but it excludes redundant computing at a coarser granularity of screen tiles. To verify fragment occlusion as early as possible, Anglada et al. proposed early visibility resolution, a mechanism that leverages the visibility information obtained in a frame to predict the visibility of next frame [24]. Texture data is a dominant consumer of off-chip memory bandwidth, so Xie et al. explored the use of process-in-memory to reduce texture memory traffic [244]. In contrast to all these efforts, CHOPIN focuses on the efficient inter-GPU synchronization of parallel rendering in multi-GPU systems.

# Chapter 8

# Conclusions and Future Work

This chapter concludes this dissertation and provides directions for future work.

## 8.1 Conclusions

After numerous researchers have contributed their innovations to both hardware architecture and software Application Programming Interface (API), the GPU has successfully built its own ecosystem which can provide high-performance and cost-efficient computing service to a wide range of applications. True to its original usage of accelerating graphics processing, GPU is a highly parallel architecture that is designed to exploit find-grained Data Level Parallelism (DLP) and Thread Level Parallelism (TLP). It trades off single-thread latency for system-level throughput. Therefore, guaranteeing highly available parallelism during execution is critical to maximize GPU performance.

In parallel programming, individual threads are not totally independent – operations on shared data and hardware structures need to be synchronized under specific ordering constraints. Inefficient synchronization support can potentially serialize threads and reduce available parallelism, significantly hurting GPU performance. In this dissertation, we propose four enhancements to help GPU architectures provide efficient synchronization support to various applications.

First, we propose Relativistic Cache Coherence (RCC) in Chapter 3, a simple cache coherence protocol which can enforce Sequential Consistency (SC) efficiently

with distributed logical timestamps. Thanks to the timestamp independence of different SM cores, RCC can process store requests instantly by advancing the timestamp of writing core, rather than waiting for all sharers become invalid. Hence, RCC outperforms the best prior SC design by 29%, it also closes the performance gap between SC and weak memory model to only 7%. Additionally, RCC allows for switching between strong (RCC-SC) and weak (RCC-WO) consistency models at runtime with best-in-class performance and no hardware overhead.

Second, we propose GETM in Chapter 4, a novel logical-timestamp-based eager conflict detection mechanism for GPU Hardware Transactional Memory (HTM) to reduce the excessive latency of prior lazy solution. GETM eagerly detects conflicts by checking the timestamps of transactions and accessed data. Transactions are aborted immediately once a conflict is detected, so transactions that have reached commit point can be committed without additional validation. Benefiting from the dramatically faster conflict detection and transaction commits, GETM is up to 2.1× (1.2× gmean) faster than the best prior GPU TM proposal. Area overheads are 3.6× lower, and power overheads are 2.2× lower.

Third, we propose HMG in Chapter 5, a two-state hierarchical cache coherence protocol for efficient peer caching in multi-GPU systems with the enforcement of scoped memory model. Coherence hierarchy is implemented to fully exploit intra-GPU data locality and reduce the bandwidth cost of inter-GPU accesses. As scoped memory models [98, 135] have been formalized as non-multi-copy-atomic, HMG processes non-synchronization stores instantly without invalidation acknowledgments; only synchronizations stores are stalled to guarantee correct data visibility. Therefore, it is unnecessary to add transient coherence states and other hardware structures to reduce stalls, which has been very few. In a 4-GPU system, HMG can achieve 97% of the performance of an idealized caching system.

Finally, we propose CHOPIN in Chapter 6, a scalable Split Frame Rendering (SFR) scheme which fully takes advantage of the parallelism available in image composition. CHOPIN can eliminate the performance overheads of redundant computing and sequential primitive exchange that exist in prior solutions. CHOPIN composes opaque sub-images out-of-order; adjacent transparent sub-images are composed asynchronously by leveraging the associativity of pixel blending. We also design a draw command scheduler and an image composition scheduler to address

the problems of load imbalance and network congestion. In an 8-GPU system, CHOPIN outperforms the best prior work by up to 1.56× (1.25× gmean).

In summary, this dissertation shows that SC can be enforced efficiently in single-GPU systems with simple RCC. However, considering the huge bandwidth gap between inter- and intra-GPU links, future hierarchical multi-GPU architectures could change this insight. To alleviate the performance impact of bandwidth-limited inter-GPU links, we might need to relax the memory model to some extent and add some extra structures to optimize remote GPU accesses. Although the latest scoped memory model can potentially maximize application parallelism and simplify hardware implementation by relaxing store atomicity and adding scope annotations, it might impose big complexity on software programmers, thereby increasing the occurrence of synchronization bugs, such as data-races. Therefore, we believe – in future GPU systems – the tradeoff between memory model, performance, and programmability needs to be explored more deeply.

This dissertation also shows that, with eager conflict detection, GPU HTM and lock-based synchronizations can have comparable performance. However, to advocate the adoption of HTM in real GPU systems, we probably need to extend current GPU memory model with transaction-related rules for correctness guarantee. The combination of relaxed memory model and HTM could be a good tradeoff. In this way, we can reorder memory requests outside transactions, while memory requests inside transactions are executed sequentially. Additionally, it's also necessary to further reduce implementation cost of GPU HTM.

## 8.2 Directions of Future Work

### 8.2.1 Logical-Time Cache Coherence in Heterogeneous Systems

In Chapter 3, we proposed Relativistic Cache Coherence (RCC), a logical-timestamp-based cache coherence protocol for efficient Sequential Consistency (SC) in GPUs. Previously, a logical-timestamp coherence protocol, TARDIS, was proposed for CPUs [255, 257]. Considering the fact that industry vendors have exposed a Unified Memory (i.e, unified virtual address space) abstraction to programmers [168], a logical-timestamp-based cache coherence protocol for systems with integrated

CPUs and GPUs might be efficient to provide both high performance and easy programming. Meanwhile, the logical-timestamp coherence protocol also could be extended to hardware accelerators, even though only physical-timestamp cache coherence was exploited for FPGA accelerators [119]. Maintaining a consistent logical-timestamp cache coherence across heterogeneous systems also can reduce the notorious complexity of hardware verification, because it's unnecessary to verify multiple different cache coherence protocols and the interface between them.

In heterogeneous systems, applications running on different processors usually demonstrate super diverse execution characteristics, which will create different performance requirements for coherence protocol design. For example, at which granularity should the cache coherence be maintained between CPUs and GPUs need to be well explored. A granularity that is too small will result in frequent data communications and possibly under-utilize the connection bandwidth, but a coarse granularity might create lots of false sharing and potentially reduce data locality. Meanwhile, inefficient lease extension also could mess up data exchange between CPUs and GPUs, significantly hurting performance.

### 8.2.2 Reducing Transaction Abort Rates of GETM

In Chapter 4, we propose GETM, the first GPU Hardware Transactional Memory (HTM) that has eager conflict detection mechanism. Even though GETM works faster than all prior GPU TM proposals, it results in higher abort rates (Table 4.4). This depends on the available information that can be leveraged to judge which transaction should be aborted while conflicts happened. In lazy mechanisms, as transactions have finished, all information acquired during execution can be taken advantage. In contrast, eager mechanisms need to make instant decisions during transaction execution. A novel warp scheduling algorithm would be helpful if it can predict the transactions that are likely to conflict. One way to approach scheduler is to profile the aborts and classify them based on different abort reasons. Then, a new scheduler could be designed to avoid or reduce transactions aborts.

Compared to prior designs, GETM enables a higher concurrency to allow more transaction to execute in parallel (Table 4.4). However, GETM is not aware of dynamic execution by assuming a fixed concurrency level. This can be

optimized, because we observed that the level of contention between transactions changes dynamically at runtime. For example, `Barnes Hut` starts out as a high-contention application where every transaction tries to insert a leaf node near the root, and gradually relaxes into low-contention as octree grows. Therefore, a control mechanism could mitigate contention and reduce transaction abort rates, if it can dynamically adjust the concurrency level.

### 8.2.3 Scoped Memory Model vs. Easy Programming

In Chapter 5, we optimized HMG by leveraging the non-multi-copy-atomicity of the latest scoped GPU memory models [98, 135]. Although, in HMG, we successfully eliminated the complexity of invalidation acknowledgments and transient coherence states, the scope annotations which are inherent to the the latest memory models can actually complicate the programming. Programmers need to be aware of the scopes, at which the latest value of shared data is visible. Unreasonable scopes can result in incorrect synchronizations. If the correct synchronization scope cannot be determined according to static knowledge, programmers need to use a larger scope conservatively, although it is unnecessary actually. Therefore, an abstraction layer which can hide the complexity of scopes would simplify the programming and advocate more users.

Considering that GPU architecture is latency-tolerant and GPU applications do not have as much data sharing as CPUs, the GPU scoped memory models might have some space to be enforced more strongly to reduce the effort on correct synchronizations. Although prior work has concluded that SC can achieve similar performance to weak memory models [195] and the scope complexity of HRF [98] is not necessary for high performance [216], these researches were conduced in conventional single-GPU systems. In modern or forward-looking multi-GPU systems, the latency/bandwidth gap between the broadest and narrowest scope is an order of magnitude larger. Therefore, SC or Data-Race-Free (DRF) might be too strong or insufficient to guarantee high performance. The tradeoff between memory model, performance, and programmability needs to explored more deeply.

### 8.2.4 Scaling CHOPIN to Larger Systems

CHOPIN (Chapter 6) as-is is applicable for NVIDIA DGX-scale system [170, 173]. Systems which are significantly larger (e.g., 1024 GPUs) may need more innovations. Meanwhile, insatiable appetite for better visual quality has led to higher and higher resolutions, which can potentially increase the inter-GPU traffic load of image composition. These problems may lead to future research directions as follows.

First, adopting pure Split Frame Rendering (SFR) mode in a larger system will make each GPU receive a small number of draw commands that is hard to be load-balanced. At the meantime, it will also be much more challenging for the scheduling of image composition. Hence, to make full use of the available GPUs, the combination of Alternate Frame Rendering (AFR) and SFR probably will be a better choice. For example, in the combined mode, GPUs are divided into multiple groups, AFR and SFR are adopted for the inter- and intra-group rendering respectively. The schedulers in Chapters 6.4.4 and 6.4.5 are proposed for SFR, so they are still applicable for the intra-group rendering. Considering the workload variance between frames, a mechanism which can dynamically adjust the GPU group size could be helpful to achieve a better and smoother user experience.

Second, we found that larger traffic load of image composition can impact the performance benefit of CHOPIN (e.g., `grid` in Figures 6.12 and 6.14). Intuitively, we imagine an effective frame content compression mechanism would be helpful for this issue. Moreover, reducing composition frequency could be an alternative choice, but this might need the change in programming model to help programmers reorganize draw commands and create fewer but larger composition groups.

Finally, the proposed schedulers in Chapter 6 have fixed functions, which are not adaptive to various programs. However, modern graphics applications are becoming more and more diverse, so designing configurable schedulers and exposing configurations to software would be desirable for programmers. This might need the extension of graphics APIs, such as DirectX [7] and Vulkan [10].

# Bibliography


[1] AMD's answer to Nvidia's NVLink is xGMI, and it's coming to the new 7nm Vega GPU. https://www.pcgamesn.com/amd-xgmi-vega-20-gpu-nvidia-nvlink. Accessed on 2020-06-25. → pages 4, 12, 75, 100, 103

[2] HSA Platform System Architecture Specification Version 1.2. http://www.hsafoundation.com/?ddownload=5702. Accessed on 2020-06-25. → pages 75, 78, 130

[3] How to fix micro stutter in videos+games? https://www.reddit.com/r/nvidia/comments/3su8qq/how_to_fix_micro_stutter_in_videosgames/, 2016. Accessed on 2020-06-25. → page 103

[4] Nvidia GeForce GTX 1080i review: The best 4K graphics card right now. https://www.rockpapershotgun.com/2018/01/30/nvidia-geforce-gtx-1080-review-best-4k-graphics-card/, 2018. Accessed on 2020-06-25. → page 103

[5] What is microstutter and how do I fix it? https://www.pcgamer.com/what-is-microstutter-and-how-do-i-fix-it/, 2018. Accessed on 2020-06-25. → page 103

[6] Vulkan 1.1 out today with multi-GPU support, better DirectX compatibility. https://arstechnica.com/gadgets/2018/03/vulkan-1-1-adds-multi-gpu-directx-compatibility-as-khronos-looks-to-the-future/, 2018. Accessed on 2020-06-25. → page 134

[7] Direct3D 12 Programming Guide. https://docs.microsoft.com/en-us/windows/win32/direct3d12/directx-12-programming-guide, 2019. Accessed on 2020-07-08. → pages 2, 134, 141

[8] How to fix stuttering of CrossFire? https://www.reddit.com/r/crossfire/comments/ddlcl9/how_to_fix_stuttering_of_crossfire/, 2019. Accessed on 2020-06-25. → page 103

[9] A first look at Unreal Engine 5. https://www.unrealengine.com/en-US/blog/a-first-look-at-unreal-engine-5, 2020. Accessed on 2020-06-25. → page 127

[10] Vulkan® 1.2.146 - A Specification (with all registered Vulkan extensions). https://www.khronos.org/registry/vulkan/specs/1.2-extensions/pdf/vkspec.pdf, 2020. Accessed on 2020-07-08. → pages 2, 141

[11] S. V. Adve and M. D. Hill. Weak Ordering-A New Definition. In *Proceedings of the 17th International Symposium on Computer Architecture (ISCA)*, pages 2–14. IEEE, 1990. → page 15

[12] S. Aga, A. Singh, and S. Narayanasamy. zFENCE: Data-less Coherence for Efficient Fences. In *Proceedings of the 29th International Conference on Supercomputing (ICS)*, pages 295–305. ACM, 2015. → page 130

[13] N. Agarwal, T. Krishna, L. S. Peh, and N. K. Jha. GARNET: A Eetailed On-Chip Network Model inside a Full-System Simulator. In *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 33–42. IEEE, 2009. → page 38

[14] N. Agarwal, L.-S. Peh, and N. K. Jha. In-Network Snoop Ordering: Snoopy Coherence on Unordered Interconnects. In *Proceedings of the 15th International Symposium on High Performance Computer Architecture (HPCA)*, pages 67–78. IEEE, 2009. → page 131

[15] K. Akeley. Reality Engine Graphics. In *Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH)*, pages 109–116. ACM, 1993. → page 134

[16] J. Alglave, M. Batty, A. F. Donaldson, G. Gopalakrishnan, J. Ketema, D. Poetzl, T. Sorensen, and J. Wickerson. GPU Concurrency: Weak Behaviours and Programming Assumptions. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 577–591. ACM, 2015. → pages 4, 14, 20, 21, 38, 81

[17] D. G. Aliaga and A. Lastra. Automatic Image Placement to Provide A Guaranteed Frame Rate. In *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH)*, pages 307–316, 1999. → page 115

[18] J. Alsop, M. S. Orr, B. M. Beckmann, and D. A. Wood. Lazy Release Consistency for GPUs. In *Proceedings of the 49th International Symposium on Microarchitecture (MICRO)*, page 26. IEEE, 2016. → pages 76, 78, 130

[19] J. Alsop, M. D. Sinclair, and S. V. Adve. Spandex: A Flexible Interface for Efficient Heterogeneous Coherence. In *Proceedings of the 45th International Symposium on Computer Architecture (ISCA)*, pages 261–274. IEEE, 2018. → page 132

[20] AMD. AMD CrossFire guide for Direct3D® 11 applications. https://gpuopen-librariesandsdks.github.io/doc/AMD-CrossFire-guide-for-Direct3D11-applications.pdf. Accessed on 2020-06-25. → pages 5, 7, 103

[21] AMD. AMD Graphics Cores Next (GCN) Architecture. https://www.techpowerup.com/gpu-specs/docs/amd-gcn1-architecture.pdf, 2012. Accessed on 2020-06-25. → pages 20, 21, 22, 32, 129

[22] AMD. Multi-Chip Module Architecture: The Right Approach for Evolving Workloads. http://developer.amd.com/wordpress/media/2017/11/LE-62006-SB-Latency-170824-Final-1.pdf, August 2017. Accessed on 2020-06-25. → page 100

[23] C. S. Ananian, K. Asanović, B. C. Kuszmaul, C. E. Leiserson, and S. Lie. Unbounded Transactional Memory. In *Proceedings of the 11th International Symposium on High Performance Computer Architecture (HPCA)*, pages 316–327. IEEE, 2005. → pages 62, 133

[24] M. Anglada, E. de Lucas, J.-M. Parcerisa, J. L. Aragón, and A. González. Early Visibility Resolution for Removing Ineffectual Computations in the Graphics Pipeline. In *Proceedings of the 25th International Symposium on High Performance Computer Architecture (HPCA)*, pages 635–646. IEEE, 2019. → page 135

[25] M. Anglada, E. de Lucas, J.-M. Parcerisa, J. L. Aragón, P. Marcuello, and A. González. Rendering Elimination: Early Discard of Redundant Tiles in the Graphics Pipeline. In *Proceedings of the 25th International Symposium on High Performance Computer Architecture (HPCA)*, pages 623–634. IEEE, 2019. → page 135

[26] ARM Ltd. ARM Architecture Reference Manual: ARMv8, for ARMv8-A architecture profile. https://static.docs.arm.com/ddi0487/fb/DDI0487F_b_armv8_arm.pdf, 2013. Accessed on 2020-06-25. → pages 19, 130

[27] J.-M. Arnau, J.-M. Parcerisa, and P. Xekalakis. Eliminating Redundant Fragment Shader Executions on a Mobile GPU via Hardware Memoization. In *Proceeedings of the 41th International Symposium on Computer Architecture (ISCA)*, pages 529–540. ACM, 2014. → page 135

[28] D. C. Arnold, D. H. Ahn, B. R. De Supinski, G. L. Lee, B. P. Miller, and M. Schulz. Stack Trace Analysis for Large Scale Debugging. In *International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1–10. IEEE, 2007. → page 46

[29] A. Arunkumar, E. Bolotin, B. Cho, U. Milic, E. Ebrahimi, O. Villa, A. Jaleel, C.-J. Wu, and D. Nellans. MCM-GPU: Multi-Chip-Module GPUs for Continued Performance Scalability. In *Proceedings of the 44th International Symposium on Computer Architecture (ISCA)*, pages 320–332. ACM, 2017. → pages 4, 11, 74, 75, 76, 92, 100, 103

[30] R. Ausavarungnirun, J. Landgraf, V. Miller, S. Ghose, J. Gandhi, C. J. Rossbach, and O. Mutlu. Mosaic: A GPU Memory Manager with Application-Transparent Support for Multiple Page Sizes. In *Proceedings of the 50th International Symposium on Microarchitecture (MICRO)*, pages 136–150. ACM, 2017. → page 79

[31] A. Bakhoda, G. L. Yuan, W. W. Fung, H. Wong, and T. M. Aamodt. Analyzing CUDA Workloads Using a Detailed GPU Simulator. In *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 163–174. IEEE, 2009. → pages xiv, 10, 37, 38, 65, 91

[32] B. Bentley. Validating the Intel® Pentium® 4 Microprocessor. In *Proceedings of the 38th Annual Design Automation Conference (DAC)*, pages 244–248, 2001. → page 42

[33] E. W. Bethel, H. Childs, and C. Hansen. *High Performance Visualization: Enabling Extreme-Scale Scientific Insight (Chapter 5)*. CRC Press, 2012. → pages 7, 105

[34] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. The Gem5 Simulator. *SIGARCH Computer Architecture News*, 39:1–7, 2011. → page 37

[35] B. H. Bloom. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Communications of the ACM*, 13(7):422–426, 1970. → page 133

[36] C. Blundell, M. M. Martin, and T. F. Wenisch. InvisiFence: Performance-Transparent Memory Ordering in Conventional Multiprocessors. In *Proceedings of the 36th International Symposium on Computer Architecture (ISCA)*, pages 233–244. ACM, 2009. → page 130

[37] J. Bobba, N. Goyal, M. D. Hill, M. M. Swift, and D. A. Wood. TokenTM: Efficient Execution of Large Transactions with Hardware Transactional Memory. In *Proceedings of the 35th International Symposium on Computer Architecture (ISCA)*, pages 127–138. IEEE, 2008. → page 133

[38] H.-J. Boehm and S. V. Adve. Foundations of the C++ Concurrency Memory Model. In *Proceedings of the 29th International Conference on Programming Language Design and Implementation (PLDI)*, pages 68–78. ACM, 2008. → page 15

[39] C. Boyd. DirectX 11 Compute Shader. https://docplayer.net/36909319-Directx-11-compute-shader-chas-boyd-architect-windows-desktop-and-graphics-technology-microsoft.html, 2008. Accessed on 2020-06-25. → page 17

[40] A. Brownsword. Cloth in OpenCL. In *GDC*, 2009. → pages 38, 66

[41] S. Burckhardt, R. Alur, and M. M. K. Martin. Verifying Safety of a Token Coherence Implementation by Parametric Compositional Refinement. In *International Workshop on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pages 130–145, 2005. → page 42

[42] M. Burtscher and K. Pingali. An Efficient CUDA Implementation of the Tree-Based Barnes Hut n-Body Algorithm. In *GPU Computing Gems Emerald Edition*, pages 75–92. Elsevier, 2011. → pages 38, 66

[43] M. Burtscher, R. Nasre, and K. Pingali. A Quantitative Study of Irregular Programs on GPUs. In *International Symposium on Workload Characterization (IISWC)*, pages 141–151. IEEE, 2012. → page 75

[44] H. W. Cain, M. M. Michael, B. Frey, C. May, D. Williams, and H. Le. Robust Architectural Support for Transactional Memory in the Power Architecture. In *Proceedings of the 40th International Symposium on Computer Architecture (ISCA)*, pages 225–236. ACM, 2013. → page 47

[45] J. F. Cantin, M. H. Lipasti, and J. E. Smith. The Complexity of Verifying Memory Coherence. In *Proceedings of the 15th Annual Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 254–255. ACM, 2003. → page 15

[46] J. Casper, T. Oguntebi, S. Hong, N. G. Bronson, C. Kozyrakis, and K. Olukotun. Hardware Acceleration of Transactional Memory on Commodity Systems. In *Proceedings of the 16th International Conference on Architectural Support*

*for Programming Language and Operating Systems (ASPLOS)*, pages 27–38. ACM, 2011. → page 133

[47] D. Cederman and P. Tsigas. On Dynamic Load Balancing on Graphics Processors. In *Proceedings of the 23rd SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware (GH)*, pages 57–64. ACM, 2008. → page 38

[48] D. Cederman, P. Tsigas, and M. T. Chaudhry. Towards a Software Transactional Memory for Graphics Processors. In *Eurographics Symposium on Parallel Graphics and Visualization (EGPGV)*, pages 121–129, 2010. → page 133

[49] L. Ceze, J. Tuck, J. Torrellas, and C. Cascaval. Bulk Disambiguation of Speculative Threads in Multiprocessors. In *Proceedings of the 33rd International Symposium on Computer Architecture (ISCA)*, pages 227–238. ACM, 2006. → page 133

[50] L. Ceze, J. Tuck, P. Montesinos, and J. Torrellas. BulkSC: Bulk Enforcement of Sequential Consistency. In *Proceedings of the 34th International Symposium on Computer Architecture (ISCA)*, pages 278–289. ACM, 2007. → page 130

[51] H. Chafi, J. Casper, B. D. Carlstrom, A. McDonald, C. C. Minh, W. Baek, C. Kozyrakis, and K. Olukotun. A Scalable, Non-blocking Approach to Transactional Memory. In *Proceedings of the 13th International Symposium on High Performance Computer Architecture (HPCA)*, pages 97–108. IEEE, 2007. → pages 50, 133

[52] S. Chaudhry, R. Cypher, M. Ekman, M. Karlsson, A. Landin, S. Yip, H. Zeffer, and M. Tremblay. Rock: A High-Performance Sparc CMT Processor. *IEEE Micro*, 29(2):6–16, 2009. → page 47

[53] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A Benchmark Suite For Heterogeneous Computing. In *International Symposium on Workload Characterization (IISWC)*, pages 44–54. IEEE, 2009. → pages 38, 77, 92

[54] S. Che, B. M. Beckmann, S. K. Reinhardt, and K. Skadron. Pannotia: Understanding Irregular GPGPU Graph Applications. In *International Symposium on Workload Characterization (IISWC)*, pages 185–195. IEEE, 2013. → page 75

[55] S. Chen and L. Peng. Efficient GPU Hardware Transactional Memory through Early Conflict Resolution. In *Proceedings of the 22nd International Symposium on High Performance Computer Architecture (HPCA)*, pages 274–284. IEEE, 2016. → pages 47, 48, 65, 66, 72, 132

[56] S. Chen, L. Peng, and S. Irving. Accelerating GPU Hardware Transactional Memory with Snapshot Isolation. In *Proceedings of the 44th International Symposium on Computer Architecture (ISCA)*, pages 282–294. IEEE, 2017. → pages 48, 133

[57] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer. cuDNN: Efficient Primitives for Deep Learning. *CoRR*, abs/1410.0759, October 2014. URL http://arxiv.org/abs/1410.0759. → page 74

[58] L. Chien. How to Avoid Global Synchronization by Domino Scheme. *NVIDIA GPU Technology Conference (GTC)*, 2014. → pages 77, 92

[59] B. Choi, R. Komuravelli, H. Sung, R. Smolinski, N. Honarmand, S. V. Adve, V. S. Adve, N. P. Carter, and C.-T. Chou. DeNovo: Rethinking the Memory Hierarchy for Disciplined Parallelism. In *Proceedings of the 20th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 155–166. IEEE, 2011. → page 130

[60] J. Chung, L. Yen, S. Diestelhorst, M. Pohlack, M. Hohmuth, D. Christie, and D. Grossman. ASF: AMD64 Extension for Lock-Free Data Structures and Transactional Memory. In *Proceedings of the 43rd International Symposium on Microarchitecture (MICRO)*, pages 39–50. IEEE, 2010. → pages 47, 133

[61] E. M. Clarke, O. Grumberg, H. Hiraishi, S. Jha, D. E. Long, K. L. McMillan, and L. A. Ness. Verification of the Futurebus+ cache coherence protocol. *Formal Methods in System Design*, 6(2):217–232, 1995. → page 42

[62] W. J. Dally, C. T. Gray, J. Poulton, B. Khailany, J. Wilson, and L. Dennison. Hardware-Enabled Artificial Intelligence. In *Symposium on VLSI Circuits*, pages 3–6. IEEE, 2018. → page 11

[63] V. M. Del Barrio, C. González, J. Roca, A. Fernández, and E. Espasa. ATTILA: A Cycle-Level Execution-Driven Simulator for Modern GPU Architectures. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 231–241. IEEE, 2006. → page 119

[64] G. Diamos, S. Sengupta, B. Catanzaro, M. Chrzanowski, A. Coates, E. Elsen, J. Engel, A. Hannun, and S. Satheesh. Persistent RNNs: Stashing Recurrent Weights On-Chip. In *International Conference on Machine Learning (ICML)*, pages 2024–2033, 2016. → page 77

[65] D. Dice, O. Shalev, and N. Shavit. Transactional Locking II. In *International Symposium on Distributed Computing*, pages 194–208. Springer, 2006. → page 50

[66] D. Dice, Y. Lev, M. Moir, and D. Nussbaum. Early Experience with a Commercial Hardware Transactional Memory Implementation. In *Proceedings of the 14th International Conference on Architectural Support for Programming Language and Operating Systems (ASPLOS)*, pages 157–168. ACM, 2009. → page 133

[67] D. L. Dill, A. J. Drexler, A. J. Hu, and C. H. Yang. Protocol Verification as a Hardware Design Aid. In *ICCD*, volume 92, pages 522–525. Citeseer, 1992. → page 42

[68] M. Dubois, C. Scheurich, and F. Briggs. Memory Access Buffering in Multiprocessors. In *Proceedings of the 13rd International Symposium on Computer Architecture (ISCA)*, pages 434–442. ACM, 1986. → pages 20, 37

[69] L. Durant, O. Giroux, M. Harris, and N. Stam. Inside Volta: The World's Most Advanced Data Center GPU. https://devblogs.nvidia.com/inside-volta/, 2017. Accessed on 2020-06-25. → pages 3, 45

[70] S. Eilemann, M. Makhinya, and R. Pajarola. Equalizer: A Scalable Parallel Rendering Framework. *IEEE transactions on visualization and computer graphics (TVCG)*, 15(3):436–452, 2009. → pages 103, 104, 106, 134

[71] M. Eldridge, H. Igehy, and P. Hanrahan. Pomegranate: A Fully Scalable Graphics Architecture. In *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRPAH)*, pages 443–454. ACM, 2000. → page 134

[72] A. ElTantawy and T. M. Aamodt. MIMD Synchronization on SIMT Architectures. In *Proceedings of the 49th International Symposium on Microarchitecture (MICRO)*, pages 1–14. IEEE, 2016. → pages 4, 46

[73] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger. Dark Silicon and the End of Multicore Scaling. In *Proceedings of the 38th International Symposium on Computer Architecture (ISCA)*, pages 365–376. IEEE, 2011. → page 1

[74] P. Felber, C. Fetzer, P. Marlier, and T. Riegel. Time-Based Software Transactional Memory. *Transactions on Parallel and Distributed Systems*, 21(12): 1793–1807, 2010. → page 133

[75] H. Fuchs, J. Poulton, J. Eyles, T. Greer, J. Goldfeather, D. Ellsworth, S. Molnar, G. Turk, B. Tebbs, and L. Israel. Pixel-Planes 5: A Heterogeneous Multiprocessor Graphics System Using Processor-Enhanced Memories. *Siggraph Computer Graphics*, 23(3):79–88, 1989. → page 134

[76] W. W. L. Fung and T. M. Aamodt. Energy Efficient GPU Transactional Memory via Space-time Optimizations. In *Proceedings of the 46th International Symposium on Microarchitecture (MICRO)*, pages 408–420. IEEE, 2013. → pages 4, 6, 47, 48, 50, 53, 54, 61, 63, 65, 66, 72, 132, 133

[77] W. W. L. Fung, I. Singh, A. Brownsword, and T. M. Aamodt. Hardware Transactional Memory for GPU Architectures. In *Proceedings of the 44th International Symposium on Microarchitecture (MICRO)*, pages 296–307. IEEE, 2011. → pages 4, 6, 47, 48, 54, 61, 63, 65, 66, 132

[78] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In *Proceedings of the 17th International Symposium on Computer Architecture (ISCA)*, pages 15–26. IEEE, 1990. → pages 20, 37, 130

[79] K. Gharachorloo, A. Gupta, and J. L. Hennessy. Two Techniques to Enhance the Performance of Memory Consistency Models. In *Proceedings of the 20th International Conference on Parallel Processing (ICPP)*, 1991. → page 130

[80] K. Gharachorloo, M. Sharma, S. Steely, and S. Van Doren. Architecture and Design of AlphaServer GS320. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 13–24. ACM, 2000. → page 132

[81] C. Gniady and B. Falsafi. Speculative Sequential Consistency with Little Custom Storage. In *Proceedings of the 11th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 179–188. IEEE, 2002. → page 130

[82] C. Gniady, B. Falsafi, and T. N. Vijaykumar. Is SC + ILP = RC? In *Proceedings of the 26th International Symposium on Computer Architecture (ISCA)*, pages 162–171. ACM, 1999. → page 130

[83] J. Gong, S. Markidis, E. Laure, M. Otten, P. Fischer, and M. Min. Nekbone Performance on GPUs with OpenACC and CUDA Fortran Implementations. *The Journal of Supercomputing*, 72(11):4160–4180, 2016. → pages 77, 92

[84] D. Gope and M. H. Lipasti. Atomic SC for Simple In-order Processors. In *Proceedings of 20th the International Symposium on High Performance Computer Architecture (HPCA)*, pages 404–415. IEEE, 2014. → pages 24, 130

[85] A. P. Grosset, M. Prasad, C. Christensen, A. Knoll, and C. Hansen. TOD-Tree: Task-Overlapped Direct Send Tree Image Compositing for Hybrid MPI Parallelism and GPUs. *IEEE transactions on visualization and computer graphics (TVCG)*, 23(6):1677–1690, 2016. → page 127

[86] S.-L. Guo, H.-X. Wang, Y.-B. Xue, C.-M. Li, and D.-S. Wang. Hierarchical Cache Directory for CMP. *Journal of Computer Science and Technology*, 25 (2):246–256, 2010. → pages 80, 132

[87] A. Gutierrez, B. Beckmann, A. Dutu, J. Gross, J. Kalamatianos, O. Kayiran, M. Lebeane, M. Poremba, B. Potter, S. Puthoor, M. D. Sinclair, M. Wyse, J. Yin, X. Zhang, A. Jain, and T. G. Rogers. Lost in Abstraction: Pitfalls of Analyzing GPUs at the Intermediate Language Level. In *Proceedings of the 24th International Symposium on High Performance Computer Architecture (HPCA)*, pages 141–155. IEEE, 2018. → page 92

[88] E. Hagersten and M. Koster. WildFire: A Scalable Path for SMPs. In *Proceedings of the 5th International Symposium on High Performance Computer Architecture (HPCA)*, pages 172–181. IEEE, 1999. → pages 80, 132

[89] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional Memory Coherence and Consistency. In *Proceedings of the 31st International Symposium on Computer Architecture (ISCA)*. ACM, 2004. → pages 13, 14, 130, 133

[90] R. Haring, M. Ohmacht, T. Fox, M. Gschwind, D. Satterfield, K. Sugavanam, P. Coteus, P. Heidelberger, M. Blumrich, R. Wisniewski, a. gara, G. Chiu, P. Boyle, N. Chist, and C. Kim. The IBM Blue Gene/Q Compute Chip. *IEEE Micro*, 32(2):48–60, 2012. → page 47

[91] P. Harish and P. Narayanan. Accelerating Large Graph Algorithms on the GPU Using CUDA. In *International conference on high-performance computing (HiPC)*, pages 197–208. Springer, 2007. → pages 2, 6, 74, 77

[92] M. Harris and L. Nyland. Inside Pascal: NVIDIA's Newest Computing Platform. *NVIDIA GPU Technology Conference (GTC)*, 2016. → pages 21, 22

[93] T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy. Composable Memory Transactions. In *Proceedings of the 10th Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 48–60. ACM, 2005. → page 47

[94] B. A. Hechtman and D. J. Sorin. Exploring Memory Consistency for Massively-Threaded Throughput-Oriented Processors. In *Proceedings of the 40th International Symposium on Computer Architecture (ISCA)*, pages 201–212. ACM, 2013. → pages 4, 20, 21, 22, 129

[95] B. A. Hechtman, S. Che, D. R. Hower, Y. Tian, B. M. Beckmann, M. D. Hill, S. K. Reinhardt, and D. A. Wood. QuickRelease: A Throughput-oriented Approach to Release Consistency on GPUs. In *Proceedings of the 20th International Symposium on High Performance Computer Architecture (HPCA)*, pages 189–200. IEEE, 2014. → pages 76, 79, 130, 132

[96] M. Herlihy and J. E. B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *Proceedings of the 20th International Symposium on Computer Architecture (ISCA)*, pages 289–300. ACM, 1993. → pages 4, 46, 133

[97] T. H. Hetherington, M. O'Connor, and T. M. Aamodt. MemcachedGPU: Scaling-up Scale-out Key-value Stores. In *Proceedings of the 6th ACM Symposium on Cloud Computing (SoCC)*, pages 43–57. ACM, 2015. → page 45

[98] D. R. Hower, B. A. Hechtman, B. M. Beckmann, B. R. Gaster, M. D. Hill, S. K. Reinhardt, and D. A. Wood. Heterogeneous-Race-Free Memory Models. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 427–440. ACM, 2014. → pages 4, 5, 15, 75, 78, 81, 130, 137, 140

[99] W. M. Hsu. Segmented Ray Casting for Data Parallel Volume Rendering. In *Proceedings of the 1993 symposium on Parallel Rendering*, pages 7–14. IEEE, 1993. → page 105

[100] J. Huang. GTC Keynote Speech. https://www.youtube.com/watch?v=Z2XlNfCtxwI, 2019. Accessed on 2020-07-08. → page 3

[101] S. Huang, S. Xiao, and W.-c. Feng. On the Energy Efficiency of Graphics Processing Units for Scientific Computing. In *International Symposium on Parallel & Distributed Processing (IPDPS)*, pages 1–8. IEEE, 2009. → pages 2, 3

[102] G. Humphreys, M. Eldridge, I. Buck, G. Stoll, M. Everett, and P. Hanrahan. WireGL: A Scalable Graphics System for Clusters. In *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH)*, pages 129–140. ACM, 2001. → pages 104, 106, 134

[103] G. Humphreys, M. Houston, R. Ng, R. Frank, S. Ahern, P. D. Kirchner, and J. T. Klosowski. Chromium: A Stream-Processing Framework for Interactive Rendering on Clusters. *ACM transactions on graphics (TOG)*, 21(3):693–702, 2002. → pages 103, 104, 106, 134

[104] IBM. Power ISA, Version 2.07B. http://kib.kiev.ua/x86docs/POWER/PowerISA_V2.07B.pdf, 2015. Accessed on 2020-06-25. → pages 19, 21

[105] Inside HPC. TOP500 Shows Growing Momentum for Accelerators. https://insidehpc.com/2015/11/top500-shows-growing-momentum-for-accelerators/, 2015. Accessed on 2020-06-25. → page 74

[106] Intel. Intel Architecture Instruction Set Extensions Programming Reference: Chapter 8: Intel Transactional Synchronization Extensions. Technical report, 2012. → page 47

[107] C. Jacobi, T. Slegel, and D. Greiner. Transactional Memory Architecture and Implementation for IBM System Z. In *Proceedings of the 45th International Symposium on Microarchitecture (MICRO)*, pages 25–36. IEEE, 2012. → page 47

[108] A. Jain, M. Khairy, and T. G. Rogers. A Quantitative Evaluation of Contemporary GPU Simulation Methodology. *Proceedings of the ACM on Measurement and Analysis of Computing Systems (SIGMETRICS)*, page 35, 2018. → page 91

[109] A. B. Kahng, B. Li, L.-S. Peh, and K. Samadi. ORION 2.0: A Fast and Accurate NoC Power and Area Model for Early-stage Design Space Exploration. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 423–428. IEEE, 2009. → page 38

[110] G. Kestor, V. Karakostas, O. S. Unsal, A. Cristal, I. Hur, and M. Valero. RMS-TM: A Comprehensive Benchmark Suite for Transactional Memory Systems. In *Proceedings of the 2nd International Conference on Performance engineering (ICPE)*, pages 335–346. ACM, 2011. → page 66

[111] M. Khairy, A. Jain, T. M. Aamodt, and T. G. Rogers. Exploring Modern GPU Memory System Design Challenges through Accurate Modeling. *CoRR*, abs/1810.07269, October 2018. URL http://arxiv.org/abs/1810.07269. → page 91

[112] Khronos. The OpenCL Specification Version 2.2. https://www.khronos.org/registry/OpenCL/specs/2.2/pdf/OpenCL_API.pdf, 2019. Accessed on 2020-07-03. → pages 2, 3, 9, 75, 78, 130

[113] J. Kim and C. Batten. Accelerating Irregular Algorithms on GPGPUs Using Fine-Grain Hardware Worklists. In *Proceedings of the 47th International Symposium on Microarchitecture (MICRO)*, pages 75–87. IEEE, 2014. → page 75

[114] Y. Kim, J.-E. Jo, H. Jang, M. Rhu, H. Kim, and J. Kim. GPUpd: A Fast and Scalable Multi-GPU Architecture Using Cooperative Projection and Distribution. In *Proceedings of the 50th International Symposium on Microarchitecture (MICRO)*, pages 574–586. ACM, 2017. → pages 5, 7, 12, 104, 107, 121, 133

[115] A. Kirsch, M. Mitzenmacher, and U. Wieder. More Robust Hashing: Cuckoo Hashing with a Stash. *SIAM Journal on Computing*, 39(4):1543–1561, 2009. → pages 63, 70

[116] T. Knight. An Architecture for Mostly Functional Languages. In *Proceedings of the Conference on LISP and Functional Programming*, pages 105–112. ACM, 1986. → page 133

[117] K. Koukos, A. Ros, E. Hagersten, and S. Kaxiras. Building Heterogeneous Unified Virtual Memories (UVMs) without the Overhead. *ACM Transactions on Architecture and Code Optimization (TACO)*, 13(1):1, 2016. → pages 78, 130

[118] M. Kulkarni, M. Burtscher, C. Casçaval, and K. Pingali. LoneStar: A Suite of Parallel Irregular Programs. In *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 65–76. IEEE, 2009. → pages 6, 77, 92

[119] S. Kumar, A. Shriraman, and N. Vedula. Fusion: Design Tradeoffs in Coherent Cache Hierarchies for Accelerators. In *Proceedings of the 42nd International Symposium on Computer Architecture (ISCA)*, pages 733–745. ACM, 2015. → pages 25, 131, 139

[120] S. Laine and T. Karras. High-Performance Software Rasterization on GPUs. In *Proceedings of the SIGGRAPH Symposium on High Performance Graphics (HPG)*, pages 79–88. ACM, 2011. → page 106

[121] S. Lam and L. Kleinrock. Packet Switching in a Multiaccess Broadcast Channel: Dynamic Control Procedures. *Transactions on Communications*, 23(9):891–904, 1975. → page 61

[122] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM*, 21:558, 1978. → pages 24, 25, 131

[123] L. Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *Transactions on Computers*, (9):690–691, 1979. → pages 14, 19

[124] H. Q. Le, W. J. Starke, J. S. Fields, F. P. O'Connell, D. Q. Nguyen, B. J. Ronchetti, W. M. Sauer, E. M. Schwarz, and M. T. Vaden. IBM POWER6 Microarchitecture. *IBM Journal of Research and Development*, 51(6):639–662, 2007. → page 131

[125] E. A. Lee. The Problem with Threads. *IEEE Computer*, 39(5):33–42, 2006. → page 46

[126] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy. The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor. In *Proceedings of the 17th International Symposium on Computer Architecture (ISCA)*, pages 148–159. IEEE, 1990. → pages 80, 132

[127] A. Levinthal and T. Porter. Chap – A SIMD Graphics Processor. In *Proceedings of the 11th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH)*, pages 77–82. ACM, 1984. → pages 10, 61

[128] J. Lew, D. A. Shah, S. Pati, S. Cattell, M. Zhang, A. Sandhupatla, C. Ng, N. Goli, M. D. Sinclair, T. G. Rogers, et al. Analyzing Machine Learning Workloads Using a Detailed GPU Simulator. In *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 151–152. IEEE, 2019. → page 91

[129] A. Li, G.-J. van den Braak, H. Corporaal, and A. Kumar. Fine-Grained Synchronizations and Dataflow Programming on GPUs. In *Proceedings of the 29th International Conference on Supercomputing (ICS)*, pages 109–118. ACM, 2015. → page 45

[130] D. Li and M. Becchi. Multiple Pairwise Sequence Alignments with the Needleman-Wunsch Algorithm on GPU. In *SC Companion: High Performance Computing, Networking, Storage and Analysis (SCC)*, pages 1471–1472. IEEE, 2012. → pages 77, 92

[131] C. Lin, V. Nagarajan, R. Gupta, and B. Rajaram. Efficient Sequential Consistency via Conflict Ordering. In *Proceedings of the 17th International Conference on Architectural Support for Programming Language and Operating Systems (ASPLOS)*, pages 273–286. ACM, 2012. → pages 24, 130

[132] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. NVIDIA Tesla: A Unified Graphics and Computing Architecture. *IEEE Micro*, 28(2):39–55, 2008. → page 2

[133] H. Litz, D. Cheriton, A. Firoozshahian, O. Azizi, and J. P. Stevenson. SI-TM: Reducing Transactional Memory Abort Rates Through Snapshot Isolation. In *Proceedings of the 19th International Conference on Architectural Support for Programming Language and Operating Systems (ASPLOS)*, pages 383–398. ACM, 2014. → page 133

[134] D. Luebke and G. Humphreys. How GPUs Work. *Computer*, 40(2):96–100, 2007. → page 102

[135] D. Lustig, S. Sahasrabuddhe, and O. Giroux. A Formal Analysis of the NVIDIA PTX Memory Consistency Model. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 257–270. ACM, 2019. → pages 4, 5, 15, 75, 78, 80, 81, 130, 137, 140

[136] K.-L. Ma, J. S. Painter, C. D. Hansen, and M. F. Krogh. A Data Distributed, Parallel Algorithm for Ray-Traced Volume Rendering. In *Proceedings of Parallel Rendering Symposium*, pages 15–22. IEEE, 1993. → page 105

[137] J. Manson, W. Pugh, and S. V. Adve. The Java Memory Model. In *Proceedings of the Annual Symposium on Principles of Programming Languages (POPL)*, pages 378–391. ACM, 2005. → page 15

[138] M. M. K. Martin, D. J. Sorin, A. Ailamaki, A. R. Alameldeen, R. M. Dickson, C. J. Mauer, K. E. Moore, M. Plakal, M. D. Hill, and D. A. Wood. Timestamp Snooping: An Approach for Extending SMPs. In *Proceedings of the 9th International Conference on Architectural Support for Programming Language and Operating Systems (ASPLOS)*, pages 25–36. ACM, 2000. → page 131

[139] A. Meixner and D. J. Sorin. Dynamic Verification of Memory Consistency in Cache-Coherent Multithreaded Computer Architectures. volume 6, pages 18–31. IEEE, 2008. → page 131

[140] M. Méndez-Lojo, M. Burtscher, and K. Pingali. A GPU Implementation of Inclusion-based Points-to Analysis. In *Proceedings of the 17th Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 107–116. ACM, 2012. → page 45

[141] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard, Version 3.1. https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf, June 2015. Accessed on 2020-06-25. → page 100

[142] Mike Houston. Anatomy of AMD's TeraScale Graphics Engine. http://attila.ac.upc.edu/wiki/images/3/34/Houston-amd-terascale.pdf, 2008. Accessed on 2020-06-25. → pages 2, 119

[143] U. Milic, O. Villa, E. Bolotin, A. Arunkumar, E. Ebrahimi, A. Jaleel, A. Ramirez, and D. Nellans. Beyond the Socket: NUMA-aware GPUs. In *Proceedings of the 50th International Symposium on Microarchitecture (MICRO)*, pages 123–135. ACM, 2017. → pages 4, 11, 75, 76, 92, 103

[144] C. C. Minh, M. Trautmann, J. Chung, A. McDonald, N. Bronson, J. Casper, C. Kozyrakis, and K. Olukotun. An Effective Hybrid Transactional Memory System with Strong Isolation Guarantees. In *Proceedings of the 34th International Symposium on Computer Architecture (ISCA)*, pages 69–80. ACM, 2007. → page 133

[145] L. Moll, A. Heirich, and M. Shand. Sepia: Scalable 3D Compositing Using PCI Pamette. In *Proceedings of the 7th International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 146–155. IEEE, 1999. → page 105

[146] S. Molnar, J. Eyles, and J. Poulton. PixelFlow: High-Speed Rendering Using Image Composition. In *Proceedings of the 19th Annual Conference on*

*Computer Graphics and Interactive Techniques (SIGGRAPH)*, pages 231–240. ACM, 1992. → page 134

[147] S. Molnar, M. Cox, D. Ellsworth, and H. Fuchs. A Sorting Classification of Parallel Rendering. *IEEE Computer Graphics and Applicationsi (CG&A)*, 14 (4):23–32, 1994. → page 106

[148] J. R. Monfort and M. Grossman. Scaling of 3D Game Engine Workloads on Modern Multi-GPU Systems. In *Proceedings of the Conference on High Performance Graphics (HPG)*, pages 37–46. ACM, 2009. → page 106

[149] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood. LogTM: log-based transactional memory. In *Proceedings of the 12th International Symposium on High Performance Computer Architecture (HPCA)*, pages 254–265. IEEE, 2006. → pages 13, 14, 49, 50, 133

[150] N. Moscovici, N. Cohen, and E. Petrank. POSTER: A GPU-Friendly Skiplist Algorithm. In *Proceedings of the 22nd Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 449–450. ACM, 2017. → page 45

[151] V. Moya, C. Gonzalez, J. Roca, A. Fernandez, and R. Espasa. Shader Performance Analysis on A Modern GPU Architecture. In *Proceedings of the 38th International Symposium on Microarchitecture (MICRO)*, pages 355–364. IEEE, 2005. → page 119

[152] C. Mueller. The Sort-First Rendering Architecture for High-Performance Graphics. In *Proceedings of the 1995 symposium on Interactive 3D graphics (I3D)*, pages 75–84, 1995. → page 103

[153] D. Mulnix. Intel Xeon Processor Scalable Family Technical Overview. https://software.intel.com/en-us/articles/intel-xeon-processor-scalable-family-technical-overview, 2017. Accessed on 2020-06-25. → pages 80, 132

[154] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi. CACTI 6.0: A Tool to Model Large Caches. Technical report, HP Laboratories, 2009. → page 66

[155] V. Nagarajan, D. J. Sorin, M. D. Hill, and D. A. Wood. A Primer on Memory Consistency and Cache Coherence, Second Edition. *Synthesis Lectures on Computer Architecture*, 15(1):1–294, 2020. → pages 14, 15, 80

[156] S. K. Nandy and R. Narayan. An Incessantly Coherent Cache Scheme for Shared Memory Multithreaded Systems. MIT LCS CSG Memo 356. → page 131

[157] U. Neumann. Communication Costs for Parallel Volume-Rendering Algorithms. *IEEE Computer Graphics and Applicationsi (CG&A)*, 14(4):49–58, 1994. → page 105

[158] NVIDIA. GeForce Now: The Power to Play. https://www.nvidia.com/en-us/geforce-now/, . Accessed on 2020-07-08. → page 3

[159] NVIDIA. NVIDIA Clara Parabricks. https://developer.nvidia.com/clara-parabricks, . Accessed on 2020-07-08. → page 3

[160] NVIDIA. NVIDIA NVLink: High Speed GPU Interconnect. https://www.nvidia.com/en-us/design-visualization/nvlink-bridges/. Accessed on 2020-06-25. → pages 3, 4, 12, 75, 100, 103

[161] NVIDIA. NVIDIA RTX™ platform. https://developer.nvidia.com/rtx, . Accessed on 2020-07-08. → page 3

[162] NVIDIA. NVIDIA NVSwitch: The World's Highest-Bandwidth On-Node Switch. https://images.nvidia.com/content/pdf/nvswitch-technical-overview.pdf, . Accessed on 2020-06-25. → pages 3, 4, 75, 100, 103

[163] NVIDIA. NVIDIA TENSOR CORES: Unprecedented Acceleration for HPC and AI. https://www.nvidia.com/en-us/data-center/tensor-cores/, . Accessed on 2020-07-08. → page 3

[164] NVIDIA. NVIDIA Turing GPU Architecture Whitepaper. https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf, . Accessed on 2020-06-25. → pages 17, 120

[165] NVIDIA. NVIDIA's Next Generation CUDA Compute Architecture: Fermi. https://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf, 2009. Accessed on 2020-06-25. → pages 20, 21, 22, 32, 38, 66, 129

[166] NVIDIA. SLI Best Practices. http://developer.download.nvidia.com/whitepapers/2011/SLI_Best_Practices_2011_Feb.pdf, 2011. Accessed on 2020-06-25. → pages 5, 7, 103, 106, 134

[167] NVIDIA. NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/documents/NV-DS-Tesla-KCompute-Arch-May-2012-LR.pdf, 2012. Accessed on 2020-06-25. → pages 20, 21, 22, 32, 129

[168] NVIDIA. Unified Memory in CUDA 6. https://devblogs.nvidia.com/unified-memory-in-cuda-6/, Nov 2013. Accessed on 2020-06-25. → pages 3, 78, 88, 138

[169] NVIDIA. NVIDIA Geforce GTX 1080. http://international.download.nvidia.com/geforce-com/international/pdfs/GeForce_GTX_1080_Whitepaper_FINAL.pdf, 2016. Accessed on 2020-06-25. → page 103

[170] NVIDIA. NVIDIA DGX-1: Essential Instrument for AI Research. https://www.nvidia.com/en-us/data-center/dgx-1/, 2017. Accessed on 2020-06-25. → pages 3, 75, 103, 109, 120, 141

[171] NVIDIA. NVIDIA Tesla V100 Architecture. http://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf, August 2017. Accessed on 2020-06-25. → page 130

[172] NVIDIA. NVIDIA Tesla V100 GPU Architecture Whitepaper. https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf, 2017. Accessed on 2020-06-25. → page 119

[173] NVIDIA. NVIDIA DGX-2: The world's most powerful AI system for the most complex AI challenges. https://www.nvidia.com/en-us/data-center/dgx-2/, 2018. Accessed on 2020-06-25. → pages 3, 75, 103, 109, 120, 141

[174] NVIDIA. NVIDIA HGX-2: Powered by NVIDIA Tesla V100 GPUs and NVSwitch. https://www.nvidia.com/en-us/data-center/hgx/, 2018. Accessed on 2020-06-25. → pages 3, 75, 103

[175] NVIDIA. New GPU-Accelerated Weather Forecasting System Dramatically Improves Accuracy. https://news.developer.nvidia.com/new-gpu-accelerated-weather-forecasting-system-dramatically-improves-accuracy/, 2019. Accessed on 2020-07-08. → page 3

[176] NVIDIA. NVIDIA 2019 Annual Review. https://s22.q4cdn.com/364334381/files/doc_financials/annual/2019/NVIDIA-2019-Annual-Report.pdf, 2019. Accessed on 2020-06-25. → page 103

[177] NVIDIA. CUDA C++ Programming Guide Version 11.0. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html, 2020. Accessed on 2020-07-03. → pages 2, 9

[178] NVIDIA. NVIDIA DLSS 2.0: A Big Leap In AI Rendering. https://www.nvidia.com/en-us/geforce/news/nvidia-dlss-2-0-a-big-leap-in-ai-rendering/, 2020. Accessed on 2020-07-08. → page 3

[179] L. E. Olson, M. D. Hill, and D. A. Wood. Crossing Guard: Mediating Host-Accelerator Coherence Interactions. In *Proceedings of the 22th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 163–176. ACM, 2017. → page 132

[180] OpenSHMEM Project. OpenSHMEM Application Programming Interface. http://www.openshmem.org/site/sites/default/site_files/OpenSHMEM-1.4.pdf, December 2017. Accessed on 2020-06-25. → page 100

[181] S. Owens, S. Sarkar, and P. Sewell. A Better x86 Memory Model: x86-TSO. In *International Conference on Theorem Proving in Higher Order Logics (TPHOL)*, pages 391–407. Springer, 2009. → pages 19, 21, 130

[182] R. Pagh and F. F. Rodler. Cuckoo hashing. In *European Symposium on Algorithms*, pages 121–133. Springer, 2001. → page 63

[183] T. Peterka, D. Goodell, R. Ross, H.-W. Shen, and R. Thakur. A Configurable Algorithm for Parallel Image-Compositing Applications. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–10. IEEE, 2009. → page 105

[184] J. C. Phillips, R. Braun, W. Wang, J. Gumbart, E. Tajkhorshid, E. Villa, C. Chipot, R. D. Skeel, L. Kale, and K. Schulten. Scalable Molecular Dynamics with NAMD. *Journal of Computational Chemistry*, 26(16): 1781–1802, 2005. → pages 77, 92

[185] F. Pong, A. Nowatzyk, G. Aybay, and M. Dubois. Verifying Distributed Directory-Based Cache Coherence Protocols: S3.mp, a Case Study. In *European Conference on Parallel Processing*, pages 287–300. Springer, 1995. → page 42

[186] T. Porter and T. Duff. Compositing Digital Images. In *Proceedings of the 11th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH)*, pages 253–259. ACM, 1984. → page 104

[187] J. W. Poulton, W. J. Dally, X. Chen, J. G. Eyles, T. H. Greer, S. G. Tell, J. M. Wilson, and C. T. Gray. A 0.54 pJ/b 20 Gb/s Ground-Referenced Single-Ended Short-Reach Serial Link in 28 nm CMOS for Advanced Packaging Applications. *IEEE Journal of Solid-State Circuits (JSSC)*, 48(12):3206–3218, 2013. → pages 4, 75, 100, 103

[188] J. Power, A. Basu, J. Gu, S. Puthoor, B. M. Beckmann, M. D. Hill, S. K. Reinhardt, and D. A. Wood. Heterogeneous System Coherence for Integrated CPU-GPU Systems. In *Proceedings of the 46th International Symposium on Microarchitecture (MICRO)*, pages 457–467. ACM, 2013. → pages 20, 132

[189] S. H. Pugsley, M. Awasthi, N. Madan, N. Muralimanohar, and R. Balasubramonian. Scalable and Reliable Communication for Hardware Transactional Memory. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 144–154. IEEE, 2008. → page 133

[190] M. A. Raihan, N. Goli, and T. M. Aamodt. Modeling Deep Learning Accelerator Enabled GPUs. In *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 79–92. IEEE, 2019. → page 91

[191] R. Rajwar and J. R. Goodman. Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution. In *Proceedings of the 34th International Symposium on Microarchitecture (MICRO)*, pages 294–305. IEEE, 2001. → page 133

[192] R. Rajwar and J. R. Goodman. Transactional Lock-free Execution of Lock-Based Programs. In *Proceedings of the 10th International Conference on Architectural Support for Programming Language and Operating Systems (ASPLOS)*, pages 5–17. ACM, 2002. → page 133

[193] G. Ramalingam. Context-Sensitive Synchronization-Sensitive Analysis is Undecidable. *Transactions on Programming languages and Systems (TOPLAS)*, 22(2):416–430, 2000. → page 46

[194] P. Ranganathan, V. S. Pai, and S. V. Adve. Using Speculative Retirement and Larger Instruction Windows to Narrow the Performance Gap Between Memory Consistency Models. In *Proceedings of the 9th Annual Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 199–210. ACM, 1997. → page 130

[195] X. Ren and M. Lis. Efficient Sequential Consistency in GPUs via Relativistic Cache Coherence. In *Proceedings of the 23rd International Symposium on High Performance Computer Architecture (HPCA)*, pages 625–636. IEEE, 2017. → pages 76, 78, 133, 140

[196] X. Ren and M. Lis. High-Performance GPU Transactional Memory via Eager Conflict Detection. In *Proceedings of the 24th International Symposium on High Performance Computer Architecture (HPCA)*, pages 235–246. IEEE, 2018. → page 14

[197] X. Ren, D. Lustig, E. Bolotin, A. Jaleel, O. Villa, and D. Nellans. HMG: Extending Cache Coherence Protocols Across Modern Hierarchical Multi-GPU Systems. In *Proceedings of the 26th International Symposium on High Performance Computer Architecture (HPCA)*, pages 582–595. IEEE, 2020. → page 103

[198] I. Rickards and E. Sørgård. Integrating CPU & GPU: the ARM methodology. In *Game Developers Conference (GDC)*, 2013. → page 21

[199] J. Rose, J. Luu, C. W. Yu, O. Densmore, J. Goeders, A. Somerville, K. B. Kent, P. Jamieson, and J. Anderson. The VTR Project: Architecture and CAD for FPGAs from Verilog to Routing. In *Proceedings of the International Symposium on Field Programmable Gate Arrays (FPGA)*, pages 77–86. ACM, 2012. → page 38

[200] C. J. Rossbach, O. S. Hofmann, D. E. Porter, H. E. Ramadan, B. Aditya, and E. Witchel. TxLinux: Using and Managing Hardware Transactional Memory in an Operating System. In *Proceedings of 21st Symposium on Operating Systems Principles (SOSP)*, pages 87–102. ACM, 2007. → page 47

[201] C. J. Rossbach, O. S. Hofmann, and E. Witchel. Is Transactional Programming Actually Easier? In *Proceedings of the 15th Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 47–56. ACM, 2010. → page 47

[202] B. Saha, A.-R. Adl-Tabatabai, and Q. Jacobson. Architectural Support for Software Transactional Memory. In *Proceedings of the 39th International Symposium on Microarchitecture (MICRO)*, pages 185–196. IEEE, 2006. → page 133

[203] D. Sanchez, L. Yen, M. D. Hill, and K. Sankaralingam. Implementing Signatures for Transactional Memory. In *Proceedings of the 40th International*

*Symposium on Microarchitecture (MICRO)*, pages 123–133. IEEE, 2007. → page 63

[204] J. Sanders and E. Kandrot. *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley Professional, 2010. → pages 4, 20

[205] S. Sarkar, P. Sewell, J. Alglave, L. Maranget, and D. Williams. Understanding POWER Multiprocessors. In *Proceedings of the 32nd International Conference on Programming Language Design and Implementation (PLDI)*, pages 175–186. ACM, 2011. → pages 19, 21, 130

[206] J.-P. Schoellkopf. SRAM Memory Device with Flash Clear and Corresponding Flash Clear Method, Feb. 19 2008. US Patent 7,333,380. → page 33

[207] M. Segal and K. Akeley. The OpenGL® Graphics System: A Specification (Version 4.6 (Core Profile)). https://www.khronos.org/registry/OpenGL/specs/gl/glspec46.core.pdf, 2019. Accessed on 2020-06-25. → page 17

[208] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, et al. Larrabee: A Many-Core x86 Architecture for Visual Computing. *Transactions on Graphics (TOG)*, 27 (3):1–15, 2008. → page 106

[209] V. Seshadri, Y. Kim, C. Fallin, D. Lee, R. Ausavarungnirun, G. Pekhimenko, Y. Luo, O. Mutlu, P. B. Gibbons, M. A. Kozuch, et al. RowClone: Fast and Energy-Efficient in-DRAM Bulk Data Copy and Initialization. In *Proceedings of the 46th International Symposium on Microarchitecture (MICRO)*, pages 185–197. ACM, 2013. → page 133

[210] Shara Tibken. CES 2019: Moore's Law is dead, says Nvidia's CEO. https://www.cnet.com/news/moores-law-is-dead-nvidias-ceo-jensen-huang-says-at-ces-2019/, 2019. Accessed on 2020-07-08. → page 1

[211] K. S. Shim, M. H. Cho, M. Lis, O. Khan, and S. Devadas. Library Cache Coherence. Technical Report MIT-CSAIL-TR-2011-027, MIT, 2011. → pages 25, 131

[212] A. Shriraman and S. Dwarkadas. Refereeing Conflicts in Hardware Transactional Memory. In *Proceedings of the 23rd International Conference on Supercomputing (ICS)*, pages 136–146. ACM, 2009. → page 133

[213] A. Shriraman, M. F. Spear, H. Hossain, V. J. Marathe, S. Dwarkadas, and M. L. Scott. An Integrated Hardware-Software Approach to Flexible Transactional Memory. In *Proceedings of the 34th International Symposium on Computer Architecture (ISCA)*, pages 104–115. ACM, 2007. → page 133

[214] A. Shriraman, S. Dwarkadas, and M. L. Scott. Flexible Decoupled Transactional Memory Support. In *Proceedings of the 35th International Symposium on Computer Architecture (ISCA)*, pages 139–150. IEEE, 2008. → page 133

[215] K. Simonyan and A. Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition. *CoRR*, abs/1409.1556, September 2014. URL http://arxiv.org/abs/1409.1556. → page 74

[216] M. D. Sinclair, J. Alsop, and S. V. Adve. Efficient GPU Synchronization Without Scopes: Saying No to Complex Consistency Models. In *Proceedings of the 48th International Symposium on Microarchitecture (MICRO)*, pages 647–659. ACM, 2015. → pages 74, 75, 76, 78, 81, 130, 133, 140

[217] M. D. Sinclair, J. Alsop, and S. V. Adve. HeteroSync: A Benchmark Suite for Fine-Grained Synchronization on Tightly Coupled GPUs. In *International Symposium on Workload Characterization (IISWC)*, pages 239–249. IEEE, 2017. → pages 75, 91

[218] P. S. Sindhu, J.-M. Frailong, and M. Cekleov. *Scalable Shared Memory Multiprocessors*, chapter Formal Specification of Memory Models, page 25. Springer, 1992. → pages 19, 21

[219] A. Singh, S. Narayanasamy, D. Marino, T. Millstein, and M. Musuvathi. End-to-end Sequential Consistency. In *Proceedings of the 39th International Symposium on Computer Architecture (ISCA)*, pages 524–535. IEEE, 2012. → page 130

[220] A. Singh, S. Aga, and S. Narayanasamy. Efficiently Enforcing Strong Memory Ordering in GPUs. In *Proceedings of the 48th International Symposium on Microarchitecture (MICRO)*, pages 699–712. ACM, 2015. → pages 19, 20, 21, 22, 36, 37, 38, 129

[221] I. Singh, A. Shriraman, W. W. Fung, M. O'Connor, and T. M. Aamodt. Cache Coherence for GPU Architectures. In *Proceedings of the 19th International Symposium on High Performance Computer Architecture (HPCA)*, pages 578–590. IEEE, 2013. → pages 4, 6, 19, 20, 22, 23, 25, 32, 37, 38, 74, 75, 76, 79, 81, 82, 129, 131, 132, 133

[222] P. Singh, C.-R. M, P. Raghavendra, A. Nandi, D. Das, and T. Tye. AMD Platform Coherency and SoC Verification Challenges. In *Proceedings of the 4th ACM Symposium on Cloud Computing (SoCC)*. ACM, 2013. → page 21

[223] B. Sinharoy, R. N. Kalla, J. M. Tendler, R. J. Eickemeyer, and J. B. Joyner. POWER5 System Microarchitecture. *IBM journal of research and development*, 49(4.5):505–521, 2005. → page 131

[224] D. J. Sorin, M. D. Hill, and D. A. Wood. A Primer on Memory Consistency and Cache Coherence. *Synthesis Lectures on Computer Architecture*, 6(3): 1–212, 2011. → pages 14, 15, 80

[225] SPARC International. The SPARC Architecture Manual, Version 9. https://cr.yp.to/2005-590/sparcv9.pdf, 1994. Accessed on 2020-06-25. → pages 41, 130

[226] D. Steinkraus, I. Buck, and P. Simard. Using GPUs for Machine Learning Algorithms. In *8th International Conference on Document Analysis and Recognition (ICDAR)*, pages 1115–1120. IEEE, 2005. → page 2

[227] G. Stoll, M. Eldridge, D. Patterson, A. Webb, S. Berman, R. Levy, C. Caywood, M. Taveira, S. Hunt, and P. Hanrahan. Lightning-2: A High-Performance Display Subsystem for PC Clusters. In *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH)*, pages 141–148. ACM, 2001. → page 105

[228] J. M. Stone, H. S. Stone, P. Heidelberger, and J. Turek. Multiple reservations and the Oklahoma update. *Parallel & Distributed Technology: Systems & Applications*, 1(4):58–71, 1993. → pages 46, 133

[229] Y. Sun, T. Baruah, S. A. Mojumder, S. Dong, X. Gong, S. Treadway, Y. Bao, S. Hance, C. McCardwell, V. Zhao, H. Barclay, A. K. Ziabari, Z. Chen, R. Ubal, J. L. Abellán, J. Kim, A. Joshi, and D. Kaeli. MGPUSim: Enabling Multi-GPU Performance Modeling and Optimization. In *Proceedings of the 46th International Symposium on Computer Architecture (ISCA)*, pages 197–209. ACM, 2019. → page 91

[230] SUN Microsystems. SPARC Architecture Manual V8. https://gaisler.com/doc/sparcv8.pdf, 1990. Accessed on 2020-06-25. → pages 19, 21

[231] R. N. Taylor. Complexity of Analyzing the Synchronization Structure of Concurrent Programs. *Acta Informatica*, 19(1):57–84, 1983. → page 46

[232] T. N. Theis and H.-S. P. Wong. The End of Moore's Law: A New Beginning for Information Technology. *Computing in Science & Engineering*, 19(2): 41–50, 2017. → page 1

[233] S. Tomić, C. Perfumo, C. Kulkarni, A. Armejach, A. Cristal, O. Unsal, T. Harris, and M. Valero. EazyHTM: EAger-LaZY hardware Transactional Memory. In *Proceedings of the 42nd International Symposium on Microarchitecture (MICRO)*, pages 145–155. ACM, 2009. → pages 50, 133

[234] A. Villegas, Á. Navarro, R. Asenjo Plaza, O. Plata, R. Ubal, and D. Kaeli. Hardware Support for Local Memory Transactions on GPU Architectures. In *TRANSACT*, 2015. → pages 48, 133

[235] V. Vineet and P. Narayanan. CUDA Cuts: Fast Graph Cuts on the GPU. In *Proceedings of the Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, pages 1–8. IEEE, 2008. → page 66

[236] M. M. Waliullah and P. Stenstrom. Starvation-Free Transactional Memory-System Protocols. In *European Conference on Parallel Processing (ECPP)*, pages 280–291. Springer, 2007. → page 133

[237] D. A. Wallach. *PHD: A Hierarchical Cache Coherent Protocol*. PhD thesis, Massachusetts Institute of Technology, 1992. → page 132

[238] T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos. Mechanisms for Store-Wait-Free Multiprocessors. In *Proceedings of the 34th International Symposium on Computer Architecture (ISCA)*, pages 266–277. ACM, 2007. → page 130

[239] A. W. Wilson Jr. Hierarchical Cache/Bus Architecture for Shared Memory Multiprocessors. In *Proceedings of the 14th International Symposium on Computer Architecture (ISCA)*, pages 244–252. ACM, 1987. → page 132

[240] M. Wimmer and P. Wonka. Rendering Time Estimation for Real-Time Rendering. In *Eurographics Symposium on Rendering*, pages 118–129, 2003. → page 115

[241] H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos. Demystifying GPU Microarchitecture through Microbenchmarking. In *Proceedings of the International Symposium on Performance Analysis of Systems & Software (ISPASS)*, pages 235–246. IEEE, 2010. → pages 30, 37, 38, 66

[242] D. A. Wood, G. A. Gibson, and R. H. Katz. Verifying a Multiprocessor Cache Controller Using Random Test Generation. *IEEE Design & Test of Computers*, 7(4):13–25, 1990. → page 42

[243] S. Xiao and W.-c. Feng. Inter-Block GPU Communication via Fast Barrier Synchronization. In *International Symposium on Parallel & Distributed Processing (IPDPS)*, pages 1–12. IEEE, 2010. → pages 36, 38

[244] C. Xie, S. L. Song, J. Wang, W. Zhang, and X. Fu. Processing-in-Memory Enabled Graphics Processors for 3D Rendering. In *Proceedings of the 23rd International Symposium on High Performance Computer Architecture (HPCA)*, pages 637–648. IEEE, 2017. → pages 120, 135

[245] C. Xie, X. Fu, and S. Song. Perception-Oriented 3D Rendering Approximation for Modern Graphics Processors. In *Proceedings of the 24th International Symposium on High Performance Computer Architecture (HPCA)*, pages 362–374. IEEE, 2018. → page 120

[246] C. Xie, F. Xin, M. Chen, and S. L. Song. OO-VR: NUMA Friendly Object-Oriented VR Rendering Framework for Future NUMA-Based Multi-GPU Systems. In *Proceedings of the 46th International Symposium on Computer Architecture (ISCA)*, pages 53–65. IEEE, 2019. → pages 12, 115, 120, 133

[247] C. Xie, X. Zhang, A. Li, X. Fu, and S. Song. PIM-VR: Erasing Motion Anomalies In Highly-Interactive Virtual Reality World With Customized Memory Cube. In *Proceedings of the 25th International Symposium on High Performance Computer Architecture (HPCA)*, pages 609–622. IEEE, 2019. → page 120

[248] Y. Xu, R. Wang, N. Goswami, T. Li, L. Gao, and D. Qian. Software Transactional Memory for GPU Architectures. In *Proceedings of Annual International Symposium on Code Generation and Optimization (CGO)*, pages 1–10. ACM, 2014. → page 133

[249] Y. Xu, L. Gao, R. Wang, Z. Luan, W. Wu, and D. Qian. Lock-based Synchronization for GPU Architectures. In *Proceedings of the International Conference on Computing Frontiers (CF)*, pages 205–213. ACM, 2016. → page 45

[250] Y. Yao, G. Wang, Z. Ge, T. Mitra, W. Chen, and N. Zhang. Efficient Timestamp-Based Cache Coherence Protocol for Many-Core Architectures. In *Proceedings of the 30th International Conference on Supercomputing (ICS)*, pages 1–13. ACM, 2016. → page 131

[251] K. C. Yeager. The MIPS R10000 Superscalar Microprocessor. *IEEE Micro*, 16:28, Apr 1996. → page 19

[252] L. Yen, J. Bobba, M. R. Marty, K. E. Moore, H. Volos, M. D. Hill, M. M. Swift, and D. A. Wood. LogTM-SE: Decoupling Hardware Transactional Memory from Caches. In *Proceedings of the 13th International Symposium on High Performance Computer Architecture (HPCA)*, pages 261–272. IEEE, 2007. → pages 13, 14, 49, 133

[253] V. Young, A. Jaleel, E. Bolotin, E. Ebrahimi, D. Nellans, and O. Villa. Combining HW/SW Mechanisms to Improve NUMA Performance of Multi-GPU Systems. In *Proceedings of the 51th International Symposium on Microarchitecture (MICRO)*, pages 339–351. IEEE, 2018. → pages 4, 12, 74, 75, 76, 95, 103

[254] H. Yu, C. Wang, and K.-L. Ma. Massively Parallel Volume Rendering Using 2–3 Swap Image Compositing. In *Proceedings of the conference on Supercomputing (SC)*, pages 1–11. IEEE, 2008. → page 105

[255] X. Yu and S. Devadas. TARDIS: Time Travelling Coherence Algorithm for Distributed Shared Memory. In *Proceedings of the 24th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 227–240. IEEE, 2015. → pages 24, 25, 30, 35, 131, 138

[256] X. Yu, M. Vijayaraghavan, and S. Devadas. A proof of correctness for the tardis cache coherence protocol. *CoRR*, abs/1505.06459, 2015. URL http://arxiv.org/abs/1505.06459. → page 28

[257] X. Yu, H. Liu, E. Zou, and S. Devadas. Tardis 2.0: Optimized Time Traveling Coherence for Relaxed Consistency Models. In *Proceedings of the 25th International Conference on Parallel Architecture and Compilation Techniques (PACT)*, pages 261–274. IEEE, 2016. → pages 25, 36, 37, 132, 138

[258] R. J. Zerr and R. S. Baker. SNAP: SN (discrete ordinates) Application Proxy Description. *Los Alamos National Laboratories, Tech. Rep. LAUR-13-21070*, 2013. → pages 6, 77, 92

[259] F. Zyulkyarov, V. Gajinov, O. S. Unsal, A. Cristal, E. Ayguadé, T. Harris, and M. Valero. Atomic Quake: Using Transactional Memory in an Interactive Multiplayer Game Server. In *Proceedings of the 14th Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 25–34. ACM, 2009. → page 47