Exploiting Parallelism in GPUs

by

Blake Hechtman

Department of Electrical and Computer Engineering
Duke University

Date: ______________________

Approved:

____
Daniel Sorin, Supervisor

____
Benjamin Lee

____
Andrew Hilton

____
Jun Yang

____
Bradford Beckmann

Dissertation submitted in partial fulfillment of the requirements for the degree of
Doctor of Philosophy in the Department of Electrical and Computer Engineering
in the Graduate School of Duke University
2014

Abstract

Exploiting Parallelism in GPUs

by

Blake Hechtman

Department of Electrical and Computer Engineering
Duke University

Date: ____________________________
Approved:

____
Daniel Sorin, Supervisor

____
Benjamin Lee

____
Andrew Hilton

____
Jun Yang

____
Bradford Beckmann

An abstract of a dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Electrical and Computer Engineering
in the Graduate School of Duke University
2014

Abstract

Heterogeneous processors with accelerators provide an opportunity to improve performance within a given power budget. Many of these heterogeneous processors contain Graphics Processing Units (GPUs) that can perform graphics and embarrassingly parallel computation orders of magnitude faster than a CPU while using less energy. Beyond these obvious applications for GPUs, a larger variety of applications can benefit from a GPU's large computation and memory bandwidth. However, many of these applications are irregular and, as a result, require synchronization and scheduling that are commonly believed to perform poorly on GPUs. The basic building block of synchronization and scheduling is memory consistency, which is, therefore, the first place to look for improving performance on irregular applications. In this thesis, we approach the programmability of irregular applications on GPUs by thinking across traditional boundaries of the compute stack. We think about architecture, microarchitecture and runtime systems from the programmers perspective. To this end, we study architectural memory consistency on future GPUs with cache coherence. In addition, we design a GPU memory system microarchitecture that can support fine-grain and coarse-grain synchronization without sacrificing throughput. Finally, we develop a task runtime that embraces the GPU microarchitecture to perform well on fork/join parallelism desired by many programmers. Overall, this thesis contributes non-intuitive solutions to improve the performance and programmability of irregular applications from the programmer's perspective.

This thesis is dedicated to My Sun and Stars.

Contents

List of Tables

List of Figures

List of Abbreviations and Symbols

Abbreviations

GPU	Graphics Processing Unit.
GPGPU	General-purpose Graphics Processing Unit.
CPU	Central Processing Unit.
TLB	Translation lookaside Buffer
MLP	Memory-level Parallelism.
ILP	Instruction-level Parallelism.
TLP	Thread-level Parallelism.
MTTOP	Massively-Threaded Throughput-Oriented Processor
SC	Sequential Consistency.
TSO	Total Store Order.
RC	Release Consistency.
RMO	Relaxed Memory Ordering
DRF	Data-race free
RFO	Read-for-ownership.
WT	Writethrough.
QR	QuickRelease.
AMD	Advanced Micro Devices.
HSA	Heterogeneous Systems Alliance.
TVM	Task Vector Machine.

TV	Task Vector.
TMS	Task Mask Stack
TREES	Task Runtime with Explicit Epoch Synchronization

Acknowledgements

First and foremost I thank my family for their unending support in my academic pursuits. Without their support, none of this research would be possible. Further, my committee—Andrew Hilton, Bradford Beckmann, Benjamin Lee, and Jun Yang—and advisor, Daniel Sorin, were instrumental in the success of my research. I would also like to thank my colleagues for their encouragement and feedback. Especially I thank Songchun Fan, Marisabel Guevara, Luwa Matthews, and Ralph Nathan for their instrumental feedback in the writing of Chapter 5. I would like to thank Bradford Beckmann, Mark Hill, David Wood, Benedict Gaster, Steve Reinhardt, Derek Hower, and Shuai Che for their advice and collaboration during my internship at AMD Research in Bellevue. Finally, I would like to thank the NSF for funding my research since proposing my thesis.

1

Introduction

Traditional techniques to achieve performance on a single core CPU have plateaued and adding more of the same type of core is bounded by a linear performance improvement. To achieve even greater performance improvements, the future of computing will involve heterogeneity in compute resources. Each of these compute resource can perform a more narrow set of tasks with greater energy efficiency than a general-purpose CPU core. Some heterogeneous compute resources in current systems include GPUs, DSPs, video decoders, and sensor processors that cannot do the same tasks as a CPU; however, they specialize in doing a single task at least an order of magnitude faster and while consuming significantly less energy. In this thesis, we will focus on heterogeneous systems containing GPUs, since they are programmable and currently in use for some general purpose computation.

Since heterogeneous systems have the opporunity to vastly improve performance with lower energy consumption, such systems including GPUs that support general purpose computing are ripe to support a wider variety of computing paradigms. Obviously, these systems are useful for graphics applications like computer games. In addition, many data parallel applications look fundamentally similar to graphics where a function is applied to an independent set data with a full global barrier between each stage. However, we want to expand the use case of the tremendous parallelism provided by the GPU beyond graphics and graphics-like applications to less-than-expert programmers.

Extending the programmability of heterogeneous computing requires thinking about the applications that are currently difficult to program on GPUs. There are two ways to approach solving the challenges programmers observe while implementing irregular applications. First, it is possible to redesign the hardware to make it easier to write high-performance programs. Second, software systems that embrace current hardware capabilities can provide a programmable interface that performs efficiently. This thesis approaches supporting emerging parallel applications from both perspectives.

We have found that ideas that seem ludicrous on a CPU seem attractive when considering systems with a GPU because of the massive number of threads available. GPUs have a large number of cores and a vast number of threads per core that can be used to achieve impressive levels of throughput at the cost of increased latency. GPUs are attractive because they efficiently provide a vast degree of memory level parallelism for embarrassingly parallel workloads like graphics or graphics-like applications. However, not all workloads are embarrassingly parallel. As a result, we must consider problems with irregular parallelism when designing future GPUs. Irregular parallel workloads differ from embarrassingly parallel workloads because they require scheduling and synchronization to handle varying amounts of parallelism. As a result, this thesis is focused on attacking the problems of scheduling and synchronization to enable broader use of heterogeneous systems containing GPUs.

To solve scheduling and synchronization problems, we explore and understand memory consistency on GPUs. Memory consistency forms the basis for scheduling and synchronization because it enables a programmer to ensure that work is complete before a synchronization operation is performed or a dependent chuck of work can be scheduled.

To solve the problems of synchronization and scheduling, we first focus on understanding the implications of memory consistency on systems containing GPUs on both current and emerging workloads. Based on this information, we propose a novel memory hierarchy that can gracefully handle fine and coarse grain synchronization operations with a low cost compared to current designs. However, even though fine grain synchronization performs decently with this new memory system, writing these programs is still difficult. To ease the burden on programmers to schedule dependencies, we design a runtime that supports task parallelism and shields the synchronization and scheduling operations from the programmer.

Thesis Statement :

Rethinking architectural, microarchitectural, and runtime systems from the programmer's point of view can improve the performance of throughput-oriented systems with low hardware cost and less programmer effort.

In support of this thesis, there are three major contributions:

- Looking at memory consistency models from a throughput-oriented approach on a hypothetical cache coherent GPU leads to vastly different results than found on a cache-coherent CPU.

- Redesigning a GPU memory system to gracefully support fine-grain and coarse-grain synchronization for irregular parallel applications without penalizing graphics or graphics-like workloads is possible and the resulting system performs well across a wide variety of current and emerging applications.

- Designing a task parallel runtime system from a throughput-oriented mindset enables easy programmability while unlocking a GPUs compute potential by converting a CPU friendly scheduling methodology to a GPU friendly scheduling methodology.

Chapter 2 will introduce a background on GPUs and memory consistency that forms a backbone for the rest of this thesis. The chapter will outline terminology, describe OpenCL, list target platforms, and describe performance guidelines for GPUs. In addition, this chapter will describe memory consistency models used in CPUs and describe past research on how to decide between them.

Chapter 3 lays the groundwork for the thesis by helping to understand how throughput-oriented software and hardware interact with a strong memory consistency model. In this chapter, we evaluate various CPU-style memory consistency models on a future cache-coherent GPU-like processors. Overall, this chapter shows that strong memory consistency models should not be counted out as an option due to performance reason. However, the scope of this work has a variety of limitations that are resolved in the next chapters.

Chapter 4 builds on the limitations of Chapter 3 and looks at supporting fine-grain synchronization operations in a write-through memory system like those in current GPUs. In this chapter, we assume that the memory system is only required to implement a weak consistency model. This enables the use of a high-throughput write-through memory system that only pays for write latencies on synchronization events. The new memory system, QuickRelease, enables locality in future irregular parallel applications with fine-grain synchronization while preserving throughput in current embarrassingly parallel workloads.

Chapter 5 looks at the software limitations posed in Chapter 3 under the realization that even if synchronization support existed on GPUs, how one would use that synchronization in the presence of SIMD. As a result, developing workloads to use synchronization would likely require a runtime that can abstract away the details of

GPU hardware. To this end, we build a runtime, Task Runtime with Explicit Epoch Synchronization, that enables the programmer to see fork/join parallelism while the hardware sees an embarrassingly parallel workload. To do this, we enable many fine-grain synchronization operations to be amortized across an entire GPU with bulk synchronous operations. We call this idea the work-together principle. This research shows it is possible to use current GPUs to accelerate task parallel code compared to popular CPU task parallel runtimes.

Chapter 6 concludes the thesis with overlapping conclusions and insights learned in the research for this thesis. Further, this chapter outlines future research directions.

2

Background

2.1 GPUs

The entire thesis will depend on a cursory understanding of GPUs and how they are different from CPUs. First, we define some general terminology. Second, we introduce OpenCL because it is a platform independent way of programming GPUs. Finally, we describe performance guidelines that result from the interactions between OpenCL and the memory system design.

2.1.1 Terminology

This section outlines terminology used in the rest of this thesis. The OpenCL terminology will be used as it can apply to any kind of GPU or parallel system [41, 20].

- **Work-Item :** a single thread with its own register state.
- **Wavefront :** a group of work-items that share a single program counter.
- **Divergence :** when work-items within a wavefront take different control flow paths.

- **Coalescing :** when memory operations from the same wavefront can merge to a single cache or memory request if they are to the same block.

- **SIMD :** Single Instruction Multiple Data like a standard vector unit.

- **Lane :** a single scalar component of a SIMD vector.

- **SMT :** Simultaneous Multi-Threading where a single lane can support many work-items to tolerate latency.

- **SIMT :** Single Instruction Multiple Threads execution model where the divergent threads in a wavefront are masked off.

- **SIMD Engine :** a pipeline that has a SIMD-width and a SMT-depth, such that SMT-depth wavefronts can be scheduled to tolerate long latency events. The SIMD-width does not necessarily match the wavefront width.

- **Compute Unit :** one or more execution units that often share a cache and scratchpad.

- **Workgroup :** a software defined set of Wavefronts that execute on a single Compute Unit.

- **Barrier :** an execution point where all work-items in a workgroup must reach before any work-items can proceed. Further all prior memory operations must be visible before it completes. This can optionally include a memory fence.

- **NDRange :** a multi-dimensional set of workgroups executing a single kernel.

- **LdAcq :**: Load acquire, a synchronizing load instruction that acts as downward memory fence such that later operations (in program order) cannot become visible before this operation.

- **StRel :** Store release, a synchronizing store instruction that acts like an upward memory fence such that all prior memory operations (in program order) are visible before this store.

2.1.2 OpenCL

OpenCL is an open standard for heterogeneous parallel computing that explicitly separates the code executing on the host (CPU) and the device (often GPU). The host code is written in C/C++ and links to a vendor-specific OpenCL library. The device code is written in a data parallel kernel that supports a subset of C with some additional built-in datatypes and functions. Kernels are compiled by the host before they are executed on the device. Each kernel is launched with a hierarchy of threads. The hierarchy consists of a single NDRange that contains a three dimensional range of work-groups that each contain a three dimensional range of work-items.

The aspects of OpenCL most relevant to this thesis are the synchronization mechanisms. OpenCL 1.2 provides three mechanisms for synchronization. First, at the end of a kernel, all stores are visible so that a future kernel launch will be able to use the values. Second, there are workgroup-wide barriers (but not global barriers) that act as both a synchronization point and a memory fence for a workgroup. Finally, there is support for both intra-workgroup and inter-workgroup atomic operations that enable co-ordination. OpenCL 2.0 formalizes a memory consistency model and enables more forms of synchronization, but it is not yet known how these will be used. Despite the memory consistency model and shared virtual memory that could enable CPU-style synchronization, the SIMD execution model still makes it hard for programmers to express synchronization. The fundamental problem is that synchronization creates dependencies between workgroups in an NDRange and leads to busy-waiting and potentially deadlock.

2.1.3 Platforms

OpenCL generally targets systems containing a discrete GPU and a CPU, but in this thesis, we are focused mostly on systems where the GPU and CPU are integrated on a single chip like the AMD Accelerated Processing Unit (APU). These systems reduce the overhead of communication and synchronization by sharing memory and synchronizing with on-chip resources.

Discrete and integrated GPUs have highly-tuned memory systems designed for throughput at the cost of latency to perform graphics operations. Wavefronts of threads executing the same instruction will combine reads and writes to a single cache block or row buffer into a single operation. Generally this coalescing provides the spatial and temporal locality present in CPU caches. As a result, GPU cache hierarchies are write-through and often write-no-allocate to optimize the use of caches for read-only data. The caches are also deeply pipelined and heavily banked leading to increased access latency and throughput. These caches are designed mostly to filter and buffer bandwidth to the memory system due to low levels of temporal locality of coalesced accesses between different cores. In the end, many GPU applications are constrained by the available memory bandwidth. As a result, if one wants to improve the performance of a GPU, the first place to look is the memory system in hardware and the memory operations in software.

2.1.4 Performance Guidelines

The key to achieving good performance with OpenCL software is for the kernel to exploit two distinctive features of the GPU hardware. First, we want all work-items in a wavefront to perform the same work. When all work-items in a wavefront reach a branch instruction and the decision is not unanimous (i.e., some work-items take the branch and some do not), this situation is called “branch divergence.” Branch divergence degrades performance because the SIMT hardware must serialize the execution of the work-items that take the branch with respect to the work-items that do not take the branch (instead of executing them all in parallel).

The second, and more important GPU feature that OpenCL programmers seek to exploit is memory access coalescing. When a wavefront executes a load or store, accesses to the same cache block (or row buffer if the GPU has no cache) are combined into a single memory request. The ideal memory coalescing occurs when all work-items in a wavefront access memory in a cache-block-aligned unit stride. Memory coalescing reduces the demand on the memory bandwidth of a SIMD core. Coalescing is critical because memory bandwidth is almost always the bottleneck on application performance. In fact, the performance degradation due to branch divergence is largely due to branch divergence's reduction of opportunities for memory coalescing, rather than due to serialization of computation.

Related to the divergence and coalescing is the performance impacts of synchronization. Synchronization operations can easily degrade performance for two primary reasons. First, synchronization operations are likely to include loops that increase register liveness and divergence. The increased liveness will in turn limit the occupancy of the GPU, which will reduce the expected performance. Second, synchronization will require the visibility of other memory operations to be guaranteed with fences in addition to busy waiting with atomic operations. The atomic operations performed in the busy-waiting loop will compete with operations making forward progress for execution resources.

2.2 Memory Consistency

In this section we provide a brief overview on the important aspects of memory consistency that will be used throughout this thesis. These models were designed with general-purpose CPU cores in mind. Beyond the descriptions below, there exist resources with in-depth definitions and working examples in tutorials [1, 54].

Overall, consistency models formalize a relationship between the program order of thread local memory operations and the order write operations to each location in the system or coherence order. A consistency model results in a set of allowable orderings of memory operations in many threads and is defined by a processor's ISA.

2.2.1 Sequential Consistency

Sequential Consistency (SC) requires that the order of memory operations in the system is the union of the program order of all threads and the coherence order of all addresses [34]. Alternatively, in SC there is a total order of loads and stores, across all threads, that respects the program order at each thread. Each load must obtain the value of the last store in the total order. SC is generally considered the strongest consistency model.

2.2.2 Total Store Order / x86

Total Store Order (TSO) is the consistency model for the SPARC and x86 architectures [59, 45]. TSO relaxes the order between a store and a subsequent load. This enables hardware to use a FIFO write buffer to tolerate store latencies. All allowable ordering in TSO are the union of partial orders: the program order of loads with respect to other loads; stores with respect to other stores; loads with respect to subsequent stores; and the coherence order of the system. To achieve an SC ordering, a memory fence can be inserted between a store and a subsequent load. This will force the ordering of the system to be the union of coherence and program orders.

2.2.3 Relaxed Memory Order

SPARC Relaxed Memory Order (RMO) [59], Alpha [52], and the tutorial XC [54] represent a weaker set of consistency models that require explicit memory fence operations to enforce a program order. In these consistency models, only coherence ordering is required in the absence of memory fences. There are three types of fences, which are full fences, acquire fences, and release fences. Full fences require that all operations before the full fence appear before the operations after the full fence in total order. Acquire fences only require that operations after the fence become visible after operations before the fence. Release fences require that operations before the fence become visible before operations after the fence.

2.2.4 SC for Data race free

SC, TSO and the relaxed memory models are all capable of achieving an SC execution with the appropriate fences. However, all of these fences can be expensive to implement to enforce strong consistency. For a set of parallel programs that do not have a data race (concurrent access to a location where at least one operation is a write) with proper synchronization, it is only necessary to insert fence operations at synchronization boundaries. This consistency model is known as SC for data-race-free (SC for DRF) [2].

Our recent work has extended SC for DRF to heterogeneous systems with what are called heterogeneous-race-free memory models [30] that enable synchronization operations to be specified with a scope. This is useful in GPUs since the programming models are explicitly hierarchical, which means that global synchronization can be avoided to improve performance.

2.2.5 The Debate

On CPUs, there was an intense debate about which consistency model was best. Hardware designers prefer weaker memory models since they are simpler to implement. Programmers prefer stronger consistency models because it makes it easier for them to reason about how shared variables could be accessed. The performance gap between SC and weaker memory models has been shown to differ by around 10% to 40%, depending on application characteristics. In the end, all consistency models have resolved to supporting SC for DRF. Further, languages like Java and C++ have adopted the SC for DRF consistency model. In hardware, TSO seems to have been a good balance between performance and programmability for multicore CPUs.

2.3 Cache Coherence

To maintain memory consistency in a system with caches, it is minimally necessary for the hardware to maintain a coherence order where each byte in the memory system has a total order of writes. The most straight forward way to implement a coherence order in a system with caches is with a cache coherence protocol that maintains either a single writer or multiple readers for all cache blocks at all times [54]. Directory-based and snooping cache coherence have been developed explicitly for this purpose. Snooping protocols rely on broadcast all cache misses to all cores in a total order. Directory protocols can relax both the broadcast and the total order by creating a physical ordering point, called a directory, that can multicast cache misses to relevant caches and require explicit acknowledgements [36].

3

Memory Consistency for Massively Threaded Throughput-Oriented Processors

3.1 Introduction

In this chapter, we evaluate a range of CPU memory consistency on hardware similar to GPUs but with a few key differences. First, these systems will implement a write-back cache coherence protocol and support virtual memory to easily integrate into a heterogeneous chip with GPUs. Second, these systems will have a memory consistency model and only support a single address space. However, these systems will still support a vast number of threads like GPUs with the use of wide SIMD, deep SMT, and multiple cores. We will call these systems Massively Threaded Throughput-Oriented Processors (MTTOPs). The combination of these features suggests that there is insight to be learned about implementing memory consistency models on such systems.

Given that MTTOPs differ from multicore CPUs in significant ways and that they tend to run different kinds of workloads, we believe it is time to re-visit the issue of hardware memory consistency models for MTTOPs. It is unclear how these differences affect the trade-offs between consistency models, although one might expect that the extraordinary amount of concurrency in MTTOPs would make the choice of consistency model crucial. It is widely believed that the most prominent MTTOPs, GPUs, provide only very weak ordering guarantees, and conventional wisdom is that weak ordering is most appropriate for GPUs. We largely agree with this conventional wisdombut only insofar as it applies to graphics applications.

For GPGPU computing and MTTOP computing, in general, the appropriate choice of hardware consistency model is less clear. Even though current HLLs for GPGPUs provide very weak ordering, that does not imply that weakly ordered hardware is desirable. Recall that many HLLs for CPUs have weak memory models (e.g., C++ [10], Java [39]), yet that does not imply that all CPU memory models should be similarly weak [29].

In this chapter, we compare various hardware consistency models for MTTOPs in terms of performance, energy-efficiency, hardware complexity, and programmability. Perhaps surprisingly, we show that hardware consistency models have little impact on the performance of our MTTOP system model running MTTOP workloads. The MTTOP can be strongly ordered and often incur only negligible performance loss compared to weaker consistency models. Furthermore, stronger models enable simpler and more energy-efficient hardware implementations and are likely easier for programmers to reason about.

This chapter makes the following contributions:

- Discuss the issues involved in implementing hardware consistency models for MTTOPs.
- Explore the trade-offs between hardware consistency models for MTTOPs.
- Experimentally demonstrate that the choice of consistency model often has negligible impact on performance.

This section will first describe the differences between MTTOPs and current multi-core chips and how that affects consistency. Second, we will describe MTTOP implementations of Sequential Consistency (SC), Total Store Order (TSO) and Relaxed Memory Ordering (RMO). Finally, we will evaluate the MTTOP consistency implementations and conclude.

3.2 Consistency Differences for MTTOPs

This section describes why it is not simply possible to apply the same conventional wisdom learned while implementing memory consistency on multicore CPUs. The key component of this argument is that CPU architectures are latency sensitive and MTTOPs are latency tolerant. On the software side, MTTOP software contains TLP while CPU software leaves out parallelism for the hardware to find.

3.2.1 Outstanding Cache Misses Per Thread → Potential Memory Level Parallelism

A cache and memory hierarchy can only handle a finite number of outstanding memory operations. Since each MTTOP core is simple and in-order, there is little opportunity exploit MLP in an instruction stream. By design MTTOPs can support many concurrent thread contexts. This means that memory consistency models for MTTOPs do not need to focus on achieving MLP for a single thread. It is often unnecessary for MTTOPs to have multiple outstanding memory operations per thread to saturate the system's bandwidth. Weak CPU memory consistency models support more MLP by allowing loads to appear out of program order.

3.2.2 Threads per Core → Latency Tolerance

Each MTTOP core will support a large number of thread contexts to tolerate the latency necessary to access high bandwidth memory systems. This latency on MTTOPs like GPUs can be nearly 50 cycles for an L1 cache access. A memory or L2 cache access would be even longer. These latencies occur with contention when the system is fully loaded. Latency in an MTTOP is more easily accommodated with more threads rather than structures like a Load-Store Queue to reduce memory latency. The design of Load-Store Queues have been critical in evaluating the costs of memory consistency in current multicore systems.

3.2.3 Threads per System → Synchronization and Contention for Shared Data

MTTOPs often support thousands of hardware threads, while current multicore CPU chips support less than a hundred hardware threads. To utilize these threads programmers will need to split problems into smaller chunks that enable the use of more hardware resources to solve a given problem. A simple example would be performing a reduction of a very long list. A CPU implementation would probably have each thread sum a large chunk sequentially and then the results of those chunks would be summed sequentially. A MTTOP implementation would give each thread a single data items and then perform a reduction in $log(N)$ stages. This kind of example exemplifies first that barriers are likely to be frequent in MTTOP code, and that each thread performs a relatively small number of operations on a shared piece of data before sharing it with another thread. Furthermore, contention on coarse-grain locks, used frequently in CPU code, are likely to have true contention on an MTTOP.

3.2.4 Threads per System → Opportunities for Reordering

On the same vein as the prior point, the number of hardware threads in an MTTOP means that there will be few independent memory operations in a given thread. In addition, there are fewer memory operations performed between each synchronization operation. This means that even if a MTTOP core could extract MLP, it is less likely to exist due to the massively parallel software.

3.2.5 Register Spills/Fills → RAW Dependencies

MTTOPs can hold a large amount of program state in its very large combined register file. Although each MTTOP hardware thread may have less registers available than there would be in a CPU hardware thread, the aggregate register capacity should mean that a program can be completed with fewer fills and spills. Further many MTTOP support private memory spaces that can be treated independently of the memory consistency model. An MTTOP compiler would put register fills and spills in this memory space. Once register spills and fills can be ignored, there are very few places were a memory location will be written and then read before a synchronization operation. As a result, MTTOPs are unlikely to benefit from a readable write-buffer unlike CPUs.

3.2.6 Algorithms → Ratio of Loads to Stores

Stencil and linear algebra are often ported to MTTOPs due to inherent parallelism and massive memory bandwidth requirements. Both Stencils and linear algebra involve more reads than writes to memory locations. An n-point stencil requires n loads before store to memory. Linear algebra requires reading a whole row or column to perform a single store in a matrix-matrix or matrix-vector multiply. Even if these problems are chunked, there are still more loads than stores. This observation about MTTOP software leads to a notion that optimizing for stores is far less important for MTTOP software than it is for CPU software. As a result, consistency choices that reduce store latency are not likely to benefit MTTOP software significantly.

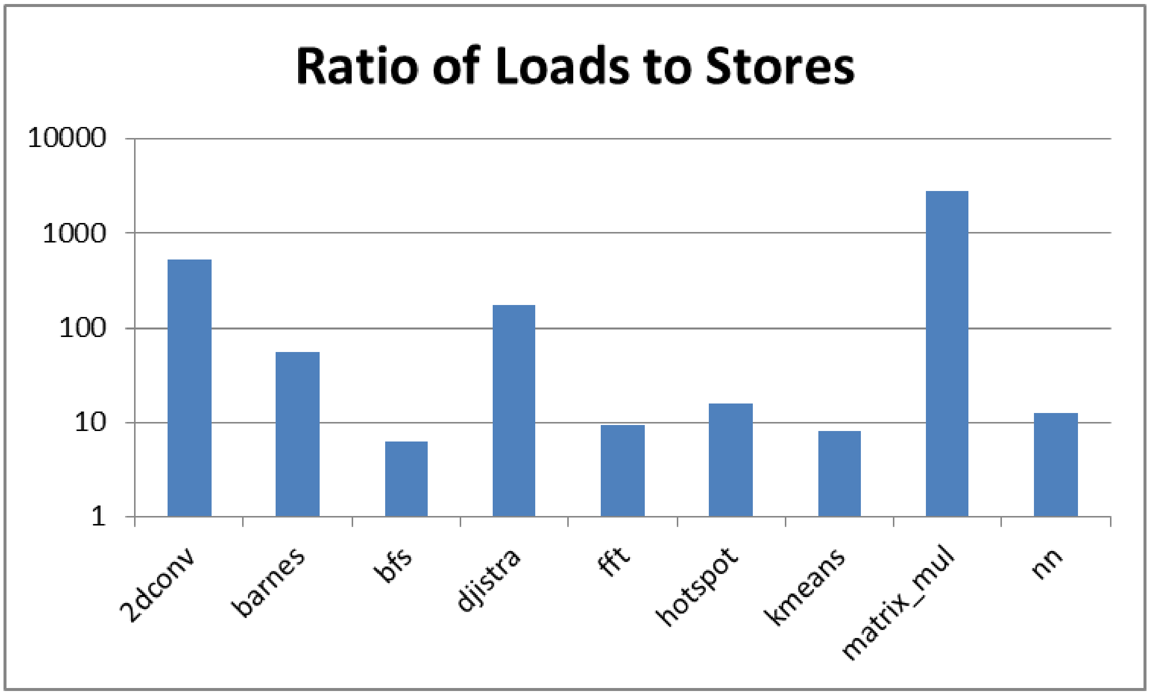


FIGURE 3.1: Load to store ratio for various MTTOP applications

3.2.7 Intermediate Assembly Languages

MTTOPs like GPUs have Intermediate Assembly languages that are finalized by a proprietary compiler to create hardware assembly. This means that even if the Intermediate Assembly Language has a defined consistency model, it is the job of both the proprietary compiler and the hardware to maintain that consistency model. This level of indirection shields the programmer in a way not possible in multicore CPUs. As a result, hardware consistency models should be considered that make the job of the proprietary compiler as easy as possible to reduce the latency of launching a new kernel.

3.2.8 Threads per System → Programmability

Programming a task to run on an MTTOP involves reasoning about various MTTOP execution models. Given the hierarchical thread layout, the programmer must already be aware of the difference between work-items, workgroups, and wavefronts to avoid incorrect producer consumer relationships. By the time the programmer reasons about where data is located on an MTTOP, understanding memory ordering seems relatively simple in comparison.

3.3 MTTOP Memory Consistency Implementations

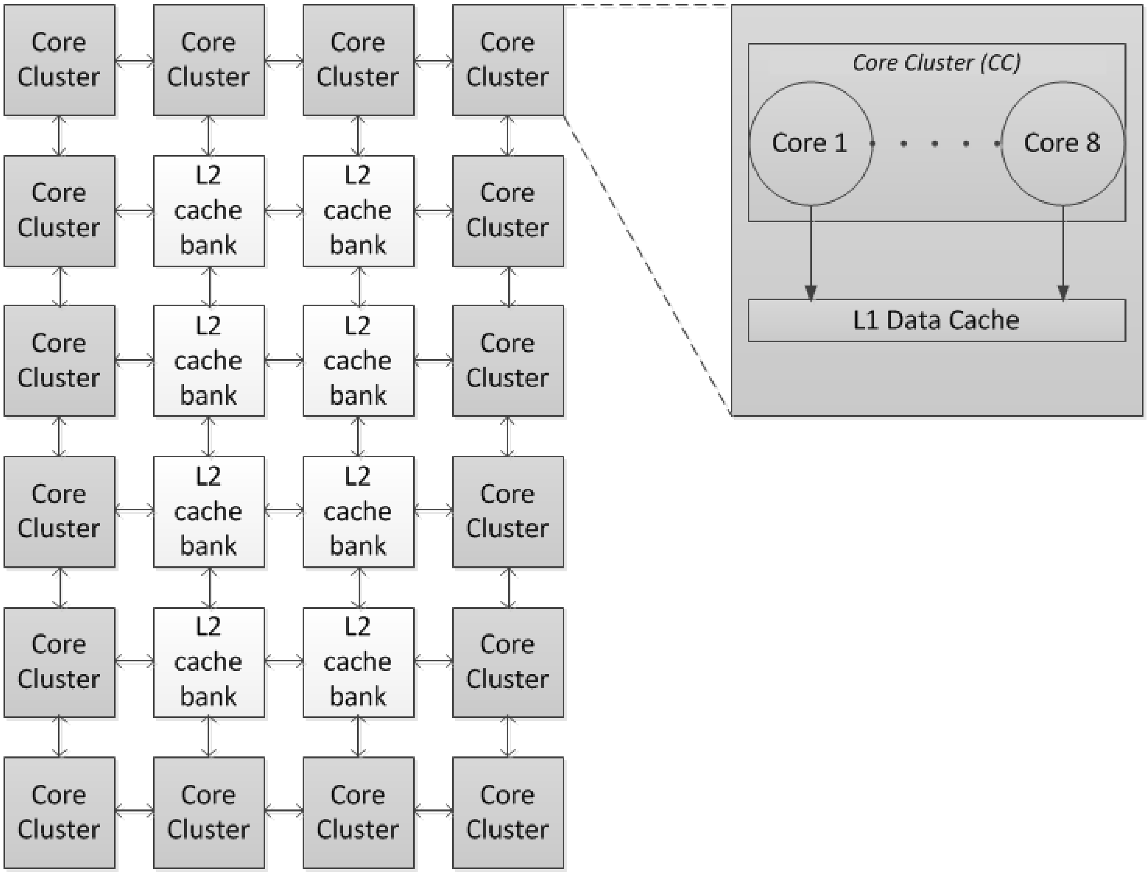
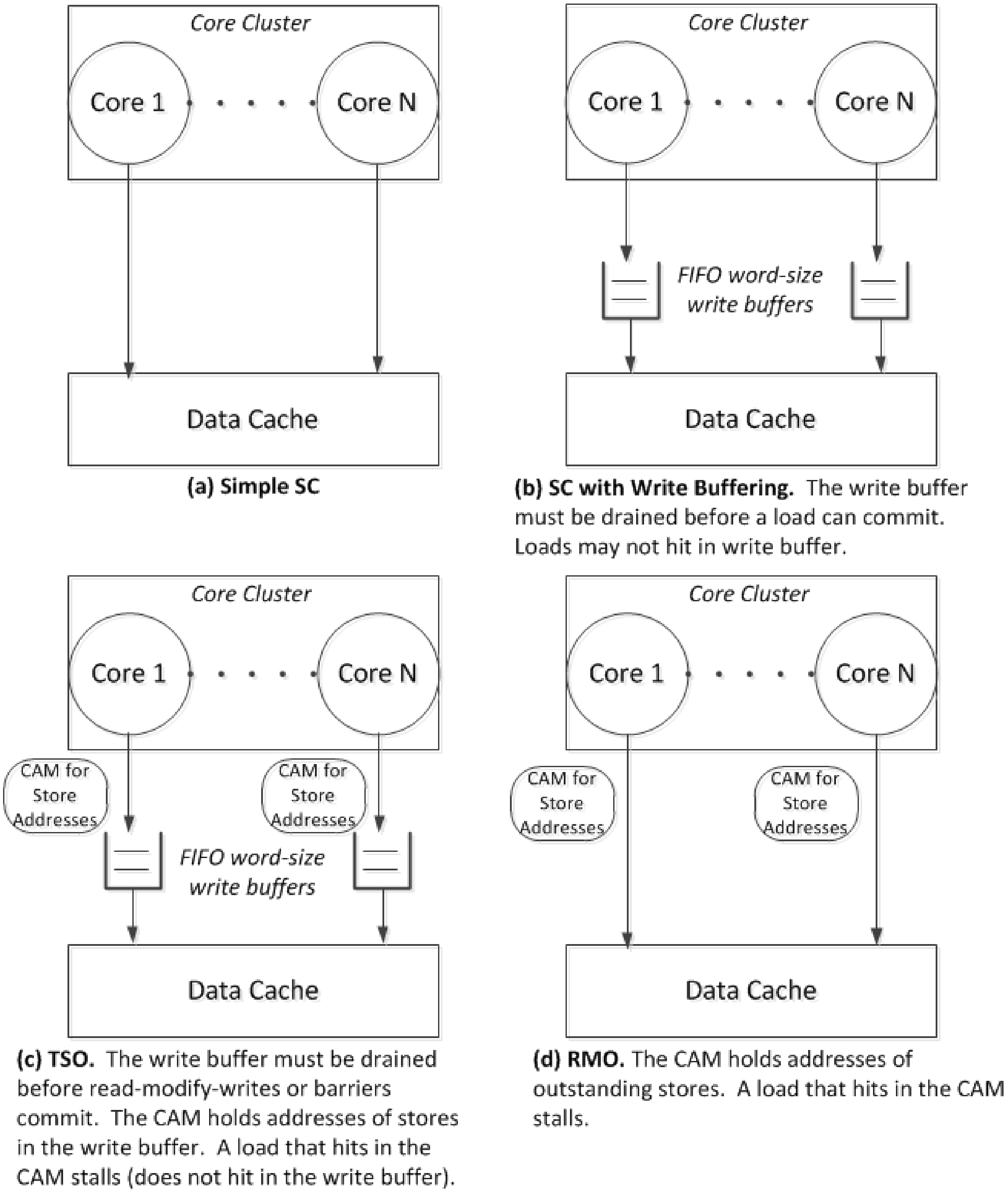


Figure 3.2: MTTOP baseline system model without write buffers

In this section, we will describe the implementations of CPU-like memory consistency models adapted to MTTOPs. This section will focus mostly on how stores are treated as that is where most of the debate about CPU consistency models has happened and where most of the difference between CPU consistency models exist. We will describe implementations of Sequential Consistency(SC), Total Store Order(TSO) and Relaxed Memory Ordering (RMO). All of these consistency models assume a writeback cache coherence protocol in the MTTOP. These MTTOPs contain many Compute Units that contain a single SIMD Engine with an L1 cache. The SIMD Engines have a SIMD width of 8 and the SMT depth of 64. These L1 caches are connected in a 2-D Torus to a banked L2 cache that holds directory state.



(a) Simple SC

(b) SC with Write Buffering. The write buffer must be drained before a load can commit. Loads may not hit in write buffer.

(c) TSO. The write buffer must be drained before read-modify-writes or barriers commit. The CAM holds addresses of stores in the write buffer. A load that hits in the CAM stalls (does not hit in the write buffer).

(d) RMO. The CAM holds addresses of outstanding stores. A load that hits in the CAM stalls.

Figure 3.3: MTTOP Implementations of SC, TSO and RMO

3.3.1 Simple SC (SC$_{simple}$)

The simplest consistency model implementation, SC$_{simple}$, is where each thread can only have a single outstanding memory operation. With this design shown in figure

3.3.a, there is no opportunity for memory operations to be re-ordered. To tolerate load and store latencies, the SIMD Engine will swap execution to another wavefront. This simple implementation's performance can suffer from long store latencies in a cache coherent memory system.

3.3.2 SC with Write Buffering (SC_{wb})

CPUs have seen performance improvements by using a write buffer even with SC. SC_{wb} follows this wisdom and extends SC_{simple} by adding a FIFO write buffer for each thread between the SIMD Engine and the L1 cache as in figure 3.3.b. This write buffer can allow stores to complete instantly. Maintaining SC requires that a threads store buffer be drained before that thread can execute another load. In addition, the write buffer must be drained in program order so that the order of writes is maintained.

3.3.3 Total Store Order (TSO)

The MTTOP TSO implementation, like SC_{wb}, uses a FIFO write buffer to allow stores to complete instantly as shown in figure 3.3.c. The TSO implementation enables loads to be issued and completed when the store buffer is non-empty. The store buffer must be empty before issuing an atomic operation or completing a memory fence. Since RAW dependencies are uncommon, the write buffer is unreadable. To prevent consistency violations, load addresses must check if that address is in the write buffer. The load will be delayed until the write buffer has drained the store with that address.

3.3.4 Relaxed Memory Ordering (RMO)

The MTTOP RMO implementation relaxes both the load and the store ordering. The RMO implementation uses the L1 cache MSHRs to order and outstanding write and read to the same address as shown in figure 3.3.d. Further the RMO implementation enables instructions to execute past loads until a data dependency or a memory fence. All outstanding loads and stores must be completed in order to execute beyond a memory fence. This implementation of RMO should cover most of the possible memory ordering behaviors available to in-order cores.

3.3.5 Graphics Compatibility

For the foreseeable future, the most common MTTOP will be a GPU that will sell primarily for graphics workloads. Graphics workloads do not need a strong or formal consistency model. In this case, all of the above structures for maintaining consistency can be ignored to avoid reducing the performance of graphics.

3.4 Evaluation

This section compares the performance and complexity of the MTTOP consistency models described in the previous section.

3.4.1 Simulation Methodology

We simulate our generalized MTTOP model with a modified version of the gem5 full-system simulator [8]. The parameters for the simulation are shown in table 3.1 and the system looks like figure 3.2.

Table 3.1: Experimental Configurations

Parameter	**Value**
SIMD width	8 threads
SMT depth	64 threads with 32 registers each
SIMD Engine	1GHz clock with Alpha-like ISA
L1 caches	L1D, L1I: 16kB, 4-way, 20-cycle hit
L2 caches	4 shared 32kB banks, 50-cycle hit latency
Off-chip memory	2 GB DRAM, hit latency 100 ns
On-chip network	2-D torus, 12GB/s link bandwidth
Consistency Model Parameters	**Value**
write buffer	perfect, instant access
CAM for store address matching	perfect, instant access

3.4.2 Benchmarks

We consider a wide range of MTTOP benchmarks, listed in table 3.2. A number of these benchmarks were handwritten microbenchmarks. The rest of the benchmarks come from the Rodinia GPGPU benchmark suite [16] that we ported. All benchmarks were written in C/C++ and compiled directly to our MTTOPs Alpha-like hardware ISA. Because the Alpha consistency model resembles our RMO implementation, the code is compiled assuming the correct consistency model. Furthermore, because SC and TSO implementations are stronger than our RMO implementation, code assuming RMO will always work.

Table 3.2: Benchmarks

Handwritten Benchmark	**Description**
barnes-hut	N-body simulation
matrix mult	matrix by matrix multiplication
dijstra	all-pairs shortest path
2D convolution	2D matrix-to-matrix convolution
fft	fast fourier transform
Rodinia Benchmark	**Description**
nn	k nearest neighbors
hotspot	processor temperature simulation
kmeans	K-means clustering
bfs	breadth-first search

3.4.3 Performance Results

In this section we present performance results, in terms of speedup in figure 3.4. All speedups are with respect to the performance of SC$_{simple}$ Across the benchmarks there is little variation between SC$_{simple}$, SC$_{wb}$, and TSO implementations for MTTOPs. This implies that a FIFO write buffer would only affect performance within 5%. The RMO implementation enables multiple outstanding independent loads which significantly improves its performance for 2D convolution and dijstra. The unordered write buffer in the RMO was no more beneficial than a FIFO write buffer from the other consistency implementations.

A few benchmarks, such as kmeans, incur some performance penalty for more relaxed models. These performance penalties, which are also extremely small, are due to resource contention during synchronization. The L1 and L2 cache size and latency configurations will change the absolute performance numbers. However, we found that the speedup results were insensitive to parameters within an order of magnitude of the listed configuration. This sensitivity could surprise many architects accustomed to CPU performance studies, but they are more intuitive after considering the differences between CPUs and MTTOPs discussed in section 3.2.

These results suggests that hardware consistency model has little impact on MTTOP performance, despite its importance for CPUs. Thus, a MTTOP hardware consistency model choice should be based on complexity and energy efficiency or programmability.

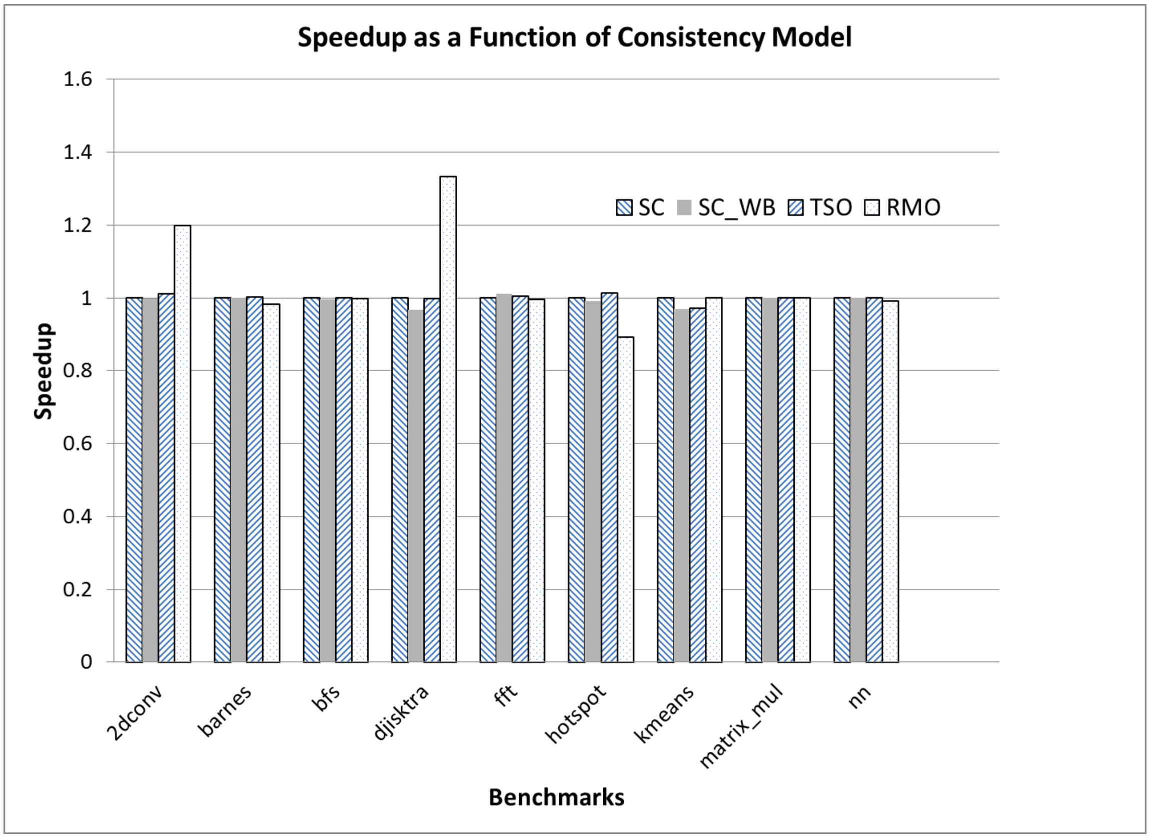


FIGURE 3.4: Performance comparisons of consistency models on MTTOPs

3.4.4 Implementation Complexity and Energy-Efficiency

Since comparing the consistency models based on performance does not create a single winner, implementation complexity and energy-efficiency should be the deciding factor. SC$_{simple}$ would seem to have the simplest hardware implementation. It requires no write buffer and at most MSHR per thread. SC offers programmability advantages, but we argue that user programmability at the hardware level is less critical for MTTOPs than for CPUs. The programmability benefits can be exposed to the writers of compilers, finalizers, and drivers. These kinds of programmers tend to be experts, but may benefit from SC.

3.5 Finalizer Programmability

We have argued that the programmability criterion is less important for choosing a MTTOP hardware consistency model than for choosing a CPU consistency model. This argument rests largely on how current GPGPU programmers are shielded from the hardware ISA; application programmers write in HLLs and can see only as far down as the intermediate language.

This argument applies to the vast majority of GPGPU programmers, but it omits one small yet important class: people who write the finalizers that translate from the intermediate language to the GPUs native hardware language. The finalizer has a difficult job in the process of running a GPGPU program. It must allocate physical registers and generate machine code without syntactic knowledge of the original source code. On CPUs, many synchronization libraries rely heavily on inline assembly code, yet GPGPUs have no such luxury. Many intermediate language instructions may have a simple one-to-one mapping to hardware instructions, but some intermediate instructions have synchronization implications (e.g., Fence, Barrier, atomic read- modify-write). It is likely that the intermediate instructions use heavyweight mechanisms to make sure that all stores ar visible. If these heavyweight mechanisms thrash data out of the cache, the cache may be rendered useless in code with synchronization.

A strong or at least explicit memory model enables the finalizer writer to formalize what the hardware does in a way that can be used to facilitate optimizations. At the very least, a hardware consistency model makes caches with coherence amenable to code with synchronization. Without a well-specified hardware consistency model, the finalizer must understand the details of the hardware implementation. The finalizer is thus likely to be overly conservative and make worst-case assumptions about the hardwares behavior. With an explicit hardware memory model, the hardware designers can enforce that model as aggressively or conservatively as chosen, and the finalizer can ignore hardware implementation details. Trying to reason about all of the possible concurrency issues in a GPGPU implementation, with its vast amounts of possibly concurrency, is a challenge that we would like to avoid.

3.6 Caveats and Limitations

The analysis we have performed in this paper necessarily makes several assumptions, and our conclusions are at least somewhat dependent on these assumptions. The two primary types of assumptions pertain to our system model and workloads, because it is not feasible to explore all possible system models or workloads. In the next two chapters we will address many of these caveats and limitations.

3.6.1 System Model

Our results depend on the MTTOP model we described in Section 4. We believe this MTTOP model is representative of future MTTOPs, yet we are aware that perfectly predicting the future is unlikely. We now discuss the implications on our results and conclusions of some possible variations in the MTTOP model.

- **Register file size:** Our register file is relatively large. A smaller register file could lead to more register spills/fills and thus to more frequent RAW dependences through memory.

- **Scratchpad memory:** Our MTTOP has no scratchpad memory. Including scratchpads is likely to make the choice of consistency model even less important, because scratchpads would reduce the number of accesses to shared memory. However, it is theoretically possible that the performance of this lesser number of accesses to shared memory would be more critical.

- **SIMT width:** If our MTTOP cores had a wider SIMT width, then there would likely be more divergence between threads, and such divergence could increase the impact of memory system performance.

- **Write-through caches:** We have assumed write-back caches, yet current GPUs support write-through caching (which is preferable for graphics). It is possible that write-through caching will persist for many future MTTOPs, although the energy benefits of write-back seem compelling. If write- through caching persists, then the latency of stores becomes more important and consistency models that take store latency off the critical path may be more attractive. However, given the relative rarity of stores, even write-through caching may be insufficient to motivate a weaker model than SC.

- **Non-write-atomic memory systems:** We have assumed memory systems that provide write atomicity [6]. However, future systems may not provide write atomicity, and we would have to adjust our memory consistency model specifications accordingly. It is unclear if or how such a memory system would impact the performance results or conclusions.

3.6.2 Workloads

Our results also depend on the workloads. We have developed and ported workloads that we believe are representative of future MTTOP workloads but, as with expected system models, it is difficult to predict the future. We now consider the impact of different workloads.

- **CPU-like workloads:** We have assumed workloads that have regular behaviors and that are particularly well-suited to MTTOPs. If more CPU-like workloads are ported to MTTOPs, these workloads may resemble CPU workloads in that they have a smaller load-to-store ratio and/or fewer available threads to run in parallel.

- **Hierarchical threading:** We have assumed a programming model with a flat thread organization, but todays GPGPU programming paradigms provide a hierarchy of threads. For example, threads may be grouped into warps, and warps may be grouped into thread blocks. With hierarchical thread grouping, we may wish to consider consistency models that are aware of this hierarchy (e.g., consistency models that provide different ordering guarantees within a warp than across warps).

3.7 Conclusions

After re-visiting the issue of hardware memory consistency models in the context of MTTOPs, the conventional wisdom on the strength of consistency is refuted. Weak consistency models, that are typical of current MTTOPs, are unlikely to be necessary for MLP due to concurrency provided by MTTOP software. As a result, the results suggest SC can achiever performance comparable to weaker consistency models on a variety of MTTOP benchmarks. Though the field of MTTOP memory systems is immature, we can say that SC and strong consistency models should not be discounted as expensive and limiting performance. Strong consistency is likely to enable easier programming of concurrent heterogeneous parallelism.

4

QuickRelease

4.1 Introduction

Graphics processing units (GPUs) provide tremendous throughput with outstanding performance-to-power ratios on graphics and graphics-like workloads by specializing the GPU architecture for the characteristics of these workloads. In particular, GPU memory systems are optimized to stream through large data structures with coarse-grain and relatively infrequent synchronization. Because synchronization is rare, current systems implement memory fences with slow and inefficient mechanisms. However, in an effort to expand the reach of their products, vendors are pushing to make GPUs more general-purpose and accessible to programmers who are not experts in the graphics domain. A key component of that push is to simplify graphics memory with support for flat addressing, fine-grain synchronization, and coherence between CPU and GPU threads [31].

However, designers must be careful when altering graphics architectures to support new features. While more generality can help expand the reach of GPUs, that generality cannot be at the expense of throughput. Notably, this means that borrowing solutions from CPU designs, such as read for ownership (RFO) coherence, that optimize for latency and cache re-use likely will not lead to viable solutions [51]. Similarly, brute-force solutions, such as making all shared data non-cacheable, also are not likely to be viable because they severely limit throughput and efficiency.

Meanwhile, write-through (WT) GPU memory systems can provide higher throughput for streaming workloads, but those memory systems will not perform as well for general-purpose GPU (GPGPU) workloads that exhibit temporal locality [16]. An alternative design is to use a write-back or write-combining cache that keeps dirty blocks in cache for a longer period of time (e.g., until evicted by an LRU replacement policy). Write-combining caches are a hybrid between WT and write-back caches in which multiple writes can be combined before reaching memory. While these caches may accelerate workloads with temporal locality within a single wavefront (warp, 64 threads), they require significant overhead to manage synchronization among wavefronts simultaneously executing on the same compute unit (CU) and incur a penalty for performing synchronization. In particular, write-combining caches require finding and evicting all dirty data written by a given wavefront, presumably by performing a heavy-weight iteration over all cache blocks. This overhead discourages fine-grain synchronization that we predict will be necessary for broader success of GPGPU compute. To this end, no current GPUs use write-combining caches for globally shared data (however, GPUs do use write-combining caches for graphic specific operations such as image, texture, and private writes).

In this chapter, we propose a GPU cache architecture called QuickRelease (QR) that is designed for throughput-oriented, fine-grain synchronization without degrading GPU memory-streaming performance. In QR, we wrap conventional GPU write-combining caches with a write-tracking component called the synchronization FIFO (S-FIFO). The S-FIFO is a simple hardware FIFO that tracks writes that have not completed ahead of an ordered set of releases. With the S-FIFO, QR caches can maintain the correct partial order between writes and synchronization operations while avoiding unnecessary inter-wavefront interference cause by cache flushes.

When a store is written into a cache, the address also is enqueued onto the S-FIFO. When the address reaches the head of the S-FIFO, the cache is forced to evict the cache block if that address is still present in the write cache. With this organization, the system can implement a release synchronization operation by simply enqueueing a release marker onto the S-FIFO. When the marker reaches the head of the queue, the system can be sure that all prior stores have reached the next level of memory. Because the S-FIFO and cache are decoupled, the memory system can utilize aggressive write-combining caches that work well for graphics workloads.

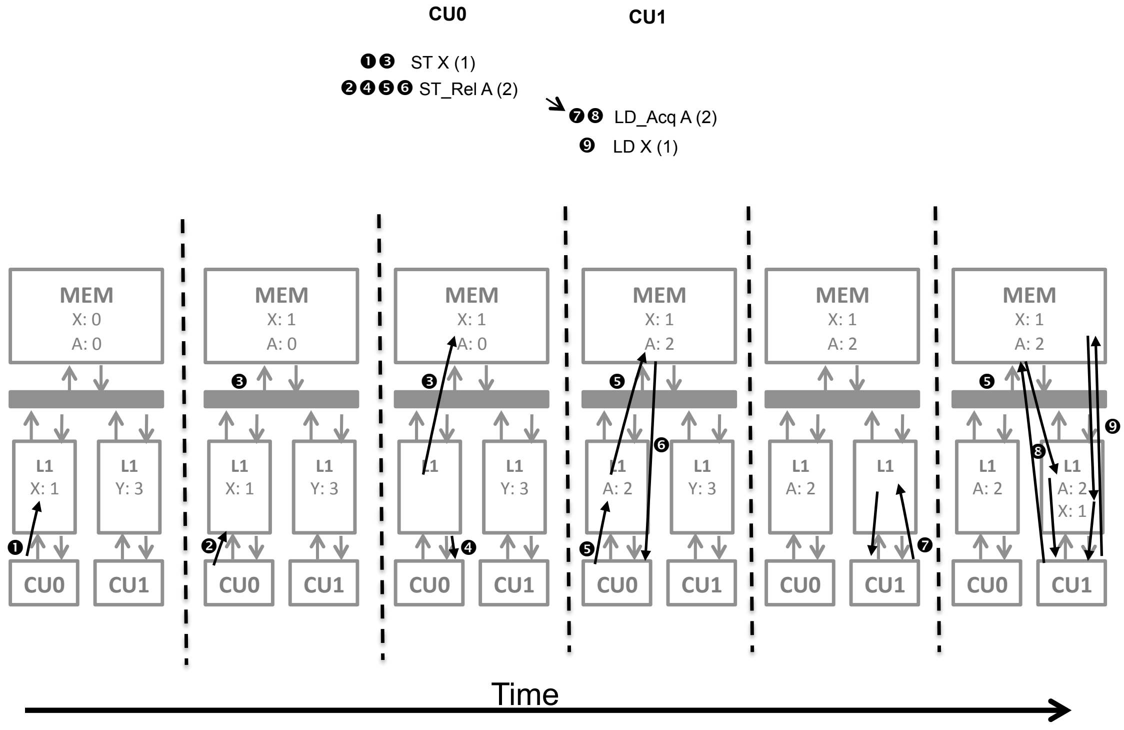


Figure 4.1: Example of QuickRelease in a simple one-level graphics memory system.

Figure 4.1 shows an example of QR. In the example, we show two threads from different CUs (a.k.a. NVIDIA streaming multi-processors) communicating a value in a simple GPU system that contains one level of write-combining cache. When a thread performs a write, it writes the value into the write-combining cache and enqueues the address at the tail of the S-FIFO (time 1). The cache block then is kept in the L1 until it is selected for eviction by the cache replacement policy or its corresponding entry in the FIFO is dequeued. The controller will dequeue an S-FIFO entry when the S-FIFO fills up or a synchronization event triggers an S-FIFO flush. In the example, the release semantic of a store/release operation causes the S-FIFO to flush. The system enqueues a special release marker into the S-FIFO (2), starts generating cache evictions for addresses ahead of the marker (3), and waits for that marker to reach the head of the queue (4). Then the system can perform the store part of the store/release (5), which, once it reaches memory, signals completion of the release to other threads (6). Finally, another thread can perform a load/acquire to complete the synchronization (7) and then load the updated value of X (8).

An important feature of the QR design is that it can be extended easily to systems with multiple levels of write-combining cache by giving each level its own S-FIFO. In that case, a write is guaranteed to be ordered whenever it has been dequeued from the S-FIFO at the last level of write-combining memory. We discuss the details of such a multi-level system in Section 4.3.

Write-combining caches in general, including QR caches, typically incur a significant overhead for tracking the specific bytes that are dirty in a cache line. This tracking is required to merge simultaneous writes from different writers to different bytes of the same cache line. Most implementations use a dirty-byte bitmask for every cache line (12.5% overhead for 64-byte cache lines) and write out only the dirty portions of a block on evictions.

To reduce the overhead of byte-level write tracking, QR separates the read and write data paths and splits a cache into read-only and (smaller) write-only subcaches. This separation is not required, but allows an implementation to reduce the overhead of writes by providing dirty bitmasks only on the write-only cache. The separation also encourages data path optimizations like independent and lazy management of write bandwidth while minimizing implementation complexity. We show that because GPU threads, unlike CPU threads, rarely perform read-after-write operations, the potential penalty of the separation is low [28]. In fact, this separation leads to less cache pollution with write-only data.

Experimental comparisons to a traditional GPGPU throughput-oriented WT memory system and to an RFO memory system demonstrate that QR achieves the best qualities of each design. Compared to the traditional GPGPU memory system, bandwidth to the memory controller was reduced by an average of 52% and the same applications ran 7% faster on average. Further, we show that future applications with frequent synchronization can run integer factors faster than a traditional GPGPU memory system. In addition, QR does not harm the performance of current streaming applications while reducing the memory traffic by 3% compared to a WT memory system. Compared to the RFO memory system, QR performs 20% faster. In fact, the RFO memory system generally performs worse than a system with the L1 cache disabled.

In summary, this chapter makes the following contributions:

- We augment an aggressive, high-throughput, write-combining cache design with precise write tracking to make synchronization faster and cheaper without the need for L1 miss status handling registers (MSHRs).

- We implement write tracking efficiently using S-FIFOs that do not require expensive CAMs or cache walks, which prevent inter-wavefront synchronization interference due to cache walks.

- Because writes require an additional byte mask in a write-combining cache, we optionally separate the read and write data paths to decrease state storage.

In this chapter, Section 4.2 describes current GPGPU memory systems and prior work in the area of GPGPU synchronization. Section 4.3 describes QR by describing its design choices and how it performs memory operations and synchronization. Section 4.4 describes the simulation environment for our experiments and the workloads we used. Section 4.5 evaluates the merits of QR compared to both a traditional GPU memory system and a theoretical MOESI coherence protocol implemented on a GPGPU.

4.2 Background and Related Work

This section introduces the GPU system terminology used throughout the paper and describes how current GPU memory systems support global synchronization. Then we introduce release consistency (RC), the basis for the memory model assumed in the next sub-section and the model being adopted by the Heterogeneous System Architecture (HSA) specification, which will govern designs from AMD, ARM, Samsung, and Qualcomm, among others. We also describe the memory systems of two accelerated processing units (APUsdevices containing a CPU, GPU, and potentially other accelerators) that obey the HSA memory model for comparison to QR: a baseline WT memory system representing todays GPUs, and an RFO cache-coherent memory system, as typically used by CPUs, extended to a GPU. Finally, in Section 4.2.3, we discuss how QR compares to prior art.

4.2.1 Current GPU Global Synchronization

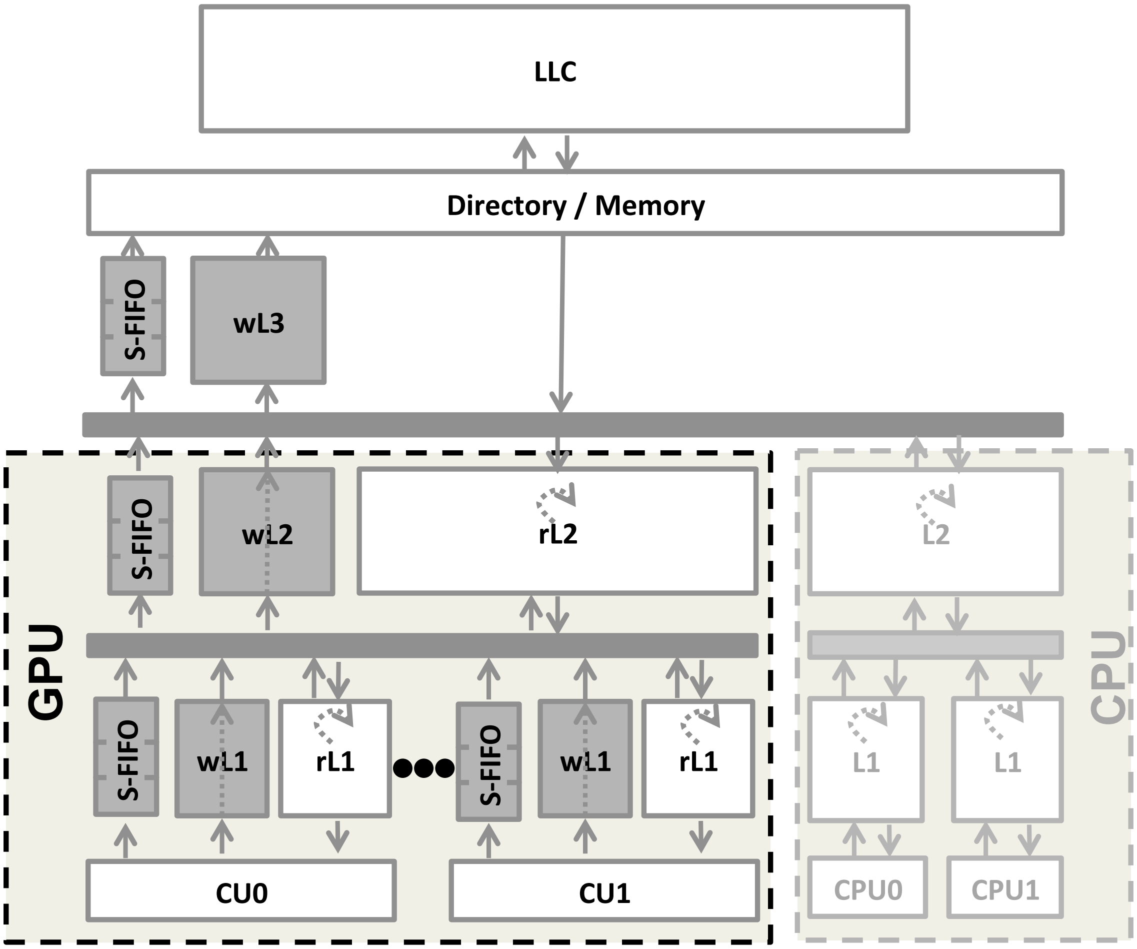


FIGURE 4.2: Baseline accelerated processing unit system. QR-specific parts are all S-FIFOs, wL1s, wL2, and wL3 (all smaller than rL1, rL2 and L4).

Global synchronization support in todays GPUs is relatively simple compared to CPUs to minimize microarchitecture complexity and because synchronization primitives currently are invoked infrequently. Figure 4.2 illustrates a GPU memory system loosely based on current architectures, such as NVIDIAs Kepler [44] or AMDs Southern Islands [4], [?]. Each CU has a WT L1 cache and all CUs share a single L2 cache. Current GPU memory models only require stores to be visible globally after memory fence operations (barrier, kernel begin, and kernel end) [41]. In the Kepler parts, the

L1 cache is disabled for all globally visible writes. Therefore, to implement a memory fence, that architecture only needs to wait for all outstanding writes (e.g., in a write buffer) to complete. The Southern Islands parts use the L1 cache for globally visible writes; therefore, the AMD parts implement a memory fence by invalidating all data in the L1 cache and flushing all written data to the shared L2 (via a cache walk) [4].

4.2.2 Release Consistency on GPUs

RC [3] has been adopted at least partially by ARM [24], Alpha [17], and Itanium [33] architectures and seems like a reasonable candidate for GPUs because it is adequately weak for many hardware designs, but strong enough to reason easily about data races. In addition, future AMD and ARM GPUs and APUs will be compliant with the HSA memory model, which is defined to be RC [31]. The rest of this paper will assume that the memory system implementation must obey RC [48].

The HSA memory model [32] adds explicit LdAcq and StRel instructions. They will be sequentially consistent. In addition, they will enforce a downward and upward fence, respectively. Unlike a CPU consistency model, enforcing the HSA memory model is not strictly the job of the hardware; it is possible to use a finalizer (an intermediate assembly language compiler) to help enforce consistency with low-level instructions. In this paper, we consider hardware solutions to enforcing RC.

4.2.3 Supporting Release Consistency

In this section, two possible baseline APU implementations of RC are described. The first is a slight modification to the system described in Section 4.2.2. The second is a nave implementation of a traditional CPU RFO cache-coherence protocol applied to an APU. Both support RC as specified.

Realistic Write-through GPU Memory System

The current GPU memory system described in Section 4.2.2 can adhere to the RC model between the CPU and GPU requests by writing through to memory via the APU directory. This means that a release operation (kernel end, barrier, or StRel) will need to wait for all prior writes to be visible globally before executing more memory operations. In addition, an acquiring memory fence (kernel begin or LdAcq) will invalidate all clean and potentially stale L1 cache data.

"Read for Ownership" GPU Memory System

Current multi-core CPU processors implement shared memory with write-back cache coherence [54]. As the RFO name implies, these systems will perform a read to gain ownership of a cache block before performing a write. In doing so, RFO protocols maintain the invariant that at any point in time only a single writer or multiple readers exist for a given cache block.

To understand the benefit an RFO protocol can provide GPUs, we added a directory to our baseline GPU cache hierarchy. It is illustrated in Figure 4.2, where the wL2 and wL3 are replaced by a fully mapped directory with full sharer state [37]. The directorys contents are inclusive of the L1s and L2, and the directory maintains coherence by allowing a single writer or multiple readers to cache a block at any time. Because there is finite state storage, the directory can recall data from the L1 or L2 to free directory space. The protocol here closely resembles the coherence protocol in recent AMD CPU architectures [19].

Related Work

Recent work by Singh et al. in cache coherence on GPUs has shown that a nave CPU-like RFO protocol will incur significant overheads [51]. This work does not include integration with CPUs.

Chapter 3 also explored memory consistency implementations on GPU-like architectures and showed that strong consistency is viable for massively threaded architectures that implement RFO cache coherence [28]. QR relies on a similar insight: read-after-write dependencies through memory are rare on GPU workloads.

Similar to the evaluated WT protocol for a GPU, the VIPS-m protocol for a CPU lazily writes through shared data by the time synchronization events are complete [49]. However, VIPS-m relies on tracking individual lazy writes using MSHRs, while the WT design does not require MSHRs and instead relies on in-order memory responses to maintain the proper synchronization order.

Conceptually, QR caches act like store queues (also called load/store queues, store buffers, or write buffers) that are found in CPUs that implement weak consistency models [50]. They have a logical FIFO organization that easily enforces ordering constraints at memory fences, thus leading to fast fine-grain synchronization. Also like a store queue, QR caches allow bypassing from the FIFO organization for high performance. This FIFO organization is only a logical wrapping, though. Under the hood, QR separates the read and write data paths and uses high-throughput, unordered write-combining caches.

Store-wait-free systems also implement a logical FIFO in parallel with the L1 cache to enforce atomic sequence order [56]. Similarly, implementations of transactional coherence and consistency (TCC) [40] use an address FIFO in parallel with the L1. However, TCCs address FIFO is used for transaction conflict detection while QRs address FIFO is used to ensure proper synchronization order.

4.3 QuickRelease Operation

In this section, we describe in detail how a QR cache hierarchy operates in a state-of-the-art SoC architecture that resembles an AMD APU. Figure 4.2 shows a diagram of the system, which features a GPU component with two levels of write-combining cache and a memory-side L3 cache shared by the CPU and GPU. For QR, we split the GPU caches into separate read and write caches to reduce implementation cost (more detail below). At each level, the write cache is approximately a quarter to an eighth the size of the read cache. Additionally, we add an S-FIFO structure in parallel with each write cache.

A goal of QR is to maintain performance for graphics workloads. At a high level, a QR design behaves like a conventional throughput-optimized write-combining cache: writes complete immediately without having to read the block first, and blocks stay in the cache until selected for eviction by a replacement policy. Because blocks are written without acquiring either permission or data, both write-combining and QR caches maintain a bitmask to track which bytes in a block are dirty, and use that mask to prevent loads from reading bytes that have not been read or written.

The QR design improves on conventional write-combining caches in two ways that increase synchronization performance and reduce implementation cost. First, QR caches use the S-FIFO to track which blocks in a cache might contain dirty data. A QR cache uses this structure to eliminate the need to perform a cache walk at synchronization events, as is done in conventional write-combining designs. Second, the QR design partitions the resources devoted to reads and writes by using read-only and write-only caches. Because writes are more expensive than reads (e.g., they require a bitmask), this reduces the overall cost of a QR design. We discuss the benefits of this separation in more detail in Section 4.3.2, and for now focus on the operation and benefits of the S-FIFO structures.

When a conventional write-combining design encounters a release, it initiates a cache walk to find and flush all dirty blocks in the cache. This relatively long-latency operation consumes cache ports and discourages the use of fine-grain synchronization. This operation is heavy-weight because many threads share the same L1 cache, and one thread synchronizing can prevent other threads from re-using data. QR overcomes this problem by using the S-FIFO. At any time, the S-FIFO contains a superset of addresses that may be dirty in the cache. The S-FIFO contains at least the addresses present in the write cache, but may contain more addresses that already have been evicted from the write cache. It is easy to iterate the S-FIFO on a release to find and flush the necessary write-cache data blocks. Conceptually the S-FIFO can be split into multiple FIFOs for each wavefront, thread, or work-group, but we found such a split provides minimal performance benefit and breaks the transitivity property on which some programs may rely [30]. Furthermore, a strict FIFO is not required to maintain a partial order of writes with respect to release operations, but we chose it because it is easy to implement.

In the following sub-sections, we describe in detail how QR performs different memory operations. First, we document the lifetime of a write operation, describing how the writes propagate through the write-only memory hierarchy and interact with S-FIFOs. Second, we document the lifetime of a basic read operation, particularly how this operation can be satisfied entirely by the separate read-optimized data path. Third, we describe how the system uses S-FIFOs to synchronize between release and acquire events. Fourth, we discuss how reads and writes interact when the same address is found in both the read and write paths, and show how QR ensures correct single-threaded read-after-write semantics.

4.3.1 Detailed Operation

Normal Write Operation

To complete a normal store operation, a CU inserts the write into the wL1, enqueues the address at the tail of the L1 S-FIFO, and, if the block is found in the rL1, sets a written bit in the tag to mark that updated data is in the wL1. The updated data will stay in the wL1 until the block is selected for eviction by the wL1 replacement policy or the address reaches the head of the S-FIFO. In either case, when evicted, the controller also will invalidate the block in the rL1, if it is present. This invalidation step is necessary to ensure correct synchronization and read-after-write operations (more details in Section 4.3.1). Writes never receive an ack.

The operation of a wL2 is similar, though with the addition of an L1 invalidation step. When a wL2 evicts a block, it invalidates the local rL2 and broadcasts an invalidation message to all the rL1s. Broadcasting to eight or 16 CUs is not a huge burden and can be alleviated with coarse-grain sharer tracking because writing to temporally shared data is unlikely without synchronization. This ensures that when using the S-FIFOs to implement synchronization, the system does not inadvertently allow a core to perform a stale read. For similar reasons, when a line is evicted from the wL3, the controller sends invalidations to the CPU cluster, the group of CPUs connected to the directory, before the line is written to the L3 cache or main memory.

Completing an atomic operation also inserts a write marker into the S-FIFO, but instead of lazily writing through to memory, the atomic is forwarded immediately to the point of system coherence, which is the directory.

CPUs perform stores as normal with coherent write-back caches. The APU directory will invalidate the rL2, which in turn will invalidate the rL1 caches to ensure consistency with respect to CPU writes at each CU. Because read caches never contain dirty data, they never need to respond with data to invalidation messages even if there is a write outstanding in the wL1/wL2/wL3. This means that CPU invalidations can be applied lazily.

Normal Read Operation

To perform a load at any level of the QR hierarchy, the read-cache tags simply are checked to see if the address is present. If the load hits valid data and the written bit is clear, the load will complete without touching the write-cache tags. On a read-tag miss or when the written bit is set, the write cache is checked to see if the load can be satisfied fully by dirty bytes present in the write cache. If so, the load is completed with the data from the write cache; otherwise, if the read request at least partially misses in the write cache, the dirty bytes are written through from the write-only cache and the read request is sent to the next level of the hierarchy.

While the write caches and their associated synchronization FIFOs ensure that data values are written to memory before release operations are completed, stale data values in the read caches also must be invalidated to achieve RC. QR invalidates these stale data copies by broadcasting invalidation messages to all rL1s when there is an eviction from the wL2. Though this may be a large amount of traffic, invalidations are much less frequent than individual stores because of significant coalescing in the wL1 and wL2. By avoiding cache flushes, valid data can persist in the rL1 across release operations, and the consequential reduction of data traffic between the rL2 and rL1 may compensate entirely for the invalidation bandwidth.

Furthermore, these invalidations are not critical to performance, unlike a traditional cache-coherence protocol in which stores depend on the acks to complete. In QR, the invalidations only delay synchronization completion. This delay is bounded based on the number of entries in the synchronization FIFO when a synchronization operation arrives. Meanwhile, write evictions and read requests do not stall waiting for invalidations because the system does not support strong consistency. As a result, QR incurs minimal performance overhead compared to a WT memory system when synchronization is rare.

QRs impact on CPU coherence is minimal and the CPUs perform loads as normal. For instance, a CPU read never will be forwarded to the GPU memory hierarchy because main memory already contains all globally visible data written by the GPU. A CPU write requires only invalidation messages to be issued to the GPU caches.

Synchronization

While loads and stores can proceed in write-combining caches without coherence actions, outstanding writes must complete to main memory and stale read-only data must be invalidated at synchronization events. QR caches implement these operations efficiently with the help of the S-FIFOs.

To start a release operation (e.g., a StRel or kernel end), a wavefront enqueues a special release marker onto the L1 S-FIFO. When inserted, the marker will cause the cache controller to begin dequeuing the S-FIFO (and performing the associated cache evictions) until the release marker reaches the head of the queue. The StRel does not require that the writes be flushed immediately; the StRel requires only that all stores in the S-FIFO hierarchy be ordered before the store of the StRel. The marker then will propagate through the cache hierarchy just like a normal write.

When the marker finally reaches the head of the wL3, the system can be sure that all prior writes from the wavefront have reached an ordering point (i.e., main memory). An acknowledgement is sent to the wavefront to signal that the release is complete.

When the release operation has an associated store operation (i.e., a StRel), the store can proceed as a normal store in the write path after the release completes. However, for performance, the store associated with the StRel should complete as soon as possible in case another thread is waiting for that synchronization to complete. Therefore, a store from a StRel will also trigger S-FIFO flushes, but it will not send an acknowledgement message back to the requesting wavefront.

Because QR broadcasts invalidations on dirty evictions, ensuring all stale data is invalidated before a release operation completes, acquire operations can be implemented as simple, light-weight loads; the acquire itself is a no-op. If a LdAcq receives the value from a previous StRel, the system can be sure that any value written by the releasing thread will have been written back to main memory and any corresponding value in a read-only cache has been invalidated.

4.3.2 Read/Write Partitioning Trade-offs

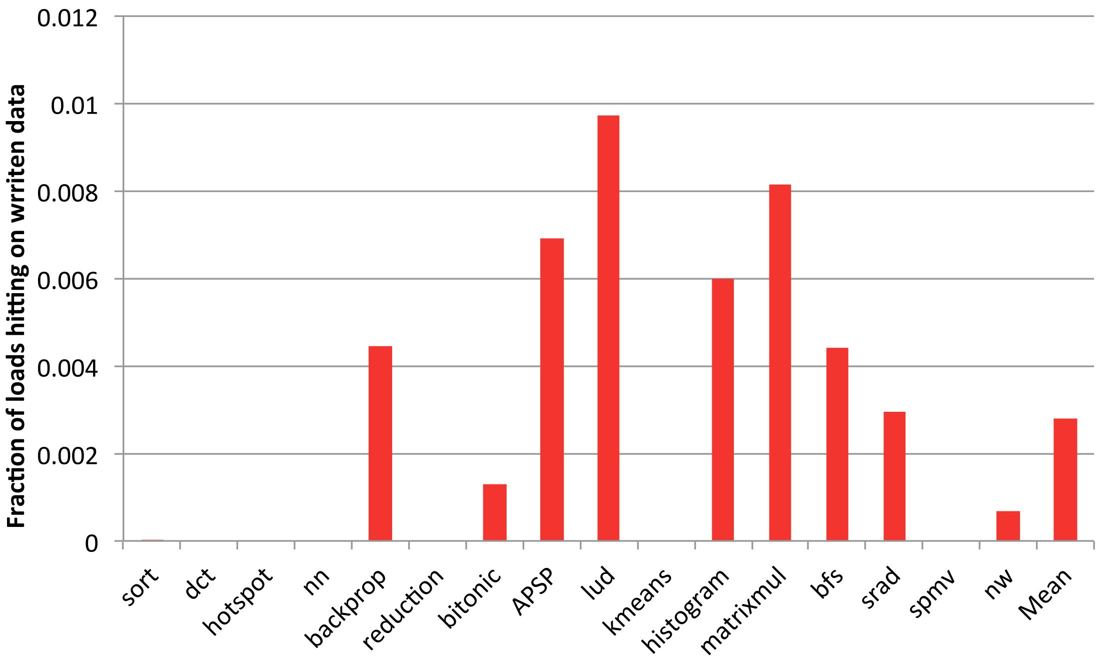


FIGURE 4.3: L1 read-after-write re-use (L1 read hits in M for RFO memory system).

In the QR design, we chose to partition the cache resources for reads and writes. While this choice reduces implementation complexity, it adds some overhead to read-after-write sequences. For example, in QR a load that hits in the write cache requires two tag look-ups and a data look-up: first check the read-cache tags, then check the write-cache tags, then read from the write-cache data array. We can justify this overhead by observing that GPGPU applications rarely demonstrate read-after-write locality.

Figure 4.3 shows the percentage of read requests that hit an L1 cache block that has been written previously (i.e., is in a modified state under RFO). For several evaluated applications, written L1 cache blocks are never re-accessed. This occurs due to a common GPU application design pattern in which a kernel streams through data, reading one data set and writing another. Subsequently, another kernel will be launched to read the written data, but by this time all that data will have been evicted from the cache.

The partitioned design has several implementation benefits. First, it reduces the state overhead needed to support writes in a write-combining cache because the dirty bitmasks are required only in the write caches. Second, it is easier to build two separate caches than a single multi-ported read/write cache with equivalent throughput. Third, the read cache can be integrated closely with the register file to improve L1 read hit latency. Meanwhile the write cache can be moved closer to the L2 bus interface and optimized exclusively as a bandwidth buffer.

4.4 Simulation Methodology and Workloads

4.4.1 The APU Simulator

Our simulation methodology extends the gem5 simulator [8] with a microarchitectural timing model of a GPU that directly executes the HSA Intermediate Language (HSAIL) [31]. To run OpenCL applications, we first generate an x86 binary that links an OpenCL library compatible with gem5s syscall emulation environment. Meanwhile, the OpenCL kernels are compiled directly into HSAIL using a proprietary industrial compiler.

Because the simulation of our OpenCL environment is HSA-compliant, the CPU and GPU share virtual memory and all memory accesses from both the CPU and GPU are assumed to be coherent. As a result, data copies between the CPU and GPU are unnecessary.

In this work, we simulate an APU-like system [11] in which the CPU and the

GPU share a single directory and DRAM controller. The GPU consists of CUs. Each CU has a private L1 data cache and all the CUs share an L2 cache. The L2 further is connected to a stateless (a.k.a. null) directory [18] with a memory-side 4-MB L3 cache, which is writeable only in the RFO system. The configurations of WT, RFO, and QR are listed in Table 4.1.

Table 4.1: Memory System Parameters

Baseline Write-Through(WT)				
Frequency	1 GHz			
Wavefronts	64 wide, 4 cycle			
Compute Units	8, 40 wavefronts each			
Memory	DDR3, 4 Channels, 400 MHz			
	banks	*tag lat.*	*data lat.*	*size*
L1	16	1	4	16 kB
L2	16	4	16	256 kB
QuickRelease(QR)				
wL1	16	1	4	4 kB
wL2	16	4	16	16 kB
wL3	16	4	16	32 kB
S-FIFO1	64 entries			
S-FIFO2	128 entries			
S-FIFO3	256 entries			
total	80 kB			
Read-for-ownership(RFO)				
directory	256 kB			
MSHRs	1024			
total	384 kB kB			

As previously noted, the storage overhead of QR compared to WT is similar to dirty bits for all WT caches. Figure 4.2 summarizes this design with a block diagram. Overall, QR uses 80 kB of additional storage that is not present in the WT baseline. To ensure that the comparison with WT is fair, we tested whether doubling the L1 capacity could benefit the WT design. Further, the RFO design requires nearly double the storage of the baseline WT memory system. We found that the extra capacity provided little benefit because of the lack of temporal locality in the evaluated benchmarks. The benefit is reduced further because WTs caches must be flushed on kernel launches.

4.4.2 Benchmarks

We evaluate QR against a conventional GPU design that uses WT caches and an idealized GPU memory system that uses RFO coherence. We run our evaluation on a set of benchmarks with diverse compute and sharing characteristics. The benchmarks represent the current state-of-the-art for GPU benchmarks. The applications and compute kernels come from the AMD APP SDK [5], OpenDwarfs [21], Rodinia [16], and two microbenchmarks that were designed to have increased data re-use and synchronization. Our microbenchmarks attempt to approximate the behavior of future workloads, which we expect will have more frequent synchronization and data re-use. Here is a brief description of the microbenchmarks:

- **APSP:** Performs a single-source shortest path until converging on an all-pairs shortest path. This application uses LdAcq and StRel to view updates as soon as they are available, to speed convergence, and uses multiple kernel launches to perform frequent communication with the host.

- **sort:** Performs a 4-byte radix sort byte by byte. For each byte, the first step counts the number of elements of each byte; the second step traverses the list to find the value at the thread ID position; and, the final step moves the correct value to the correct location and swaps the input and output arrays.

4.4.3 Re-use of the L1 Data Cache

Figure 4.4 shows the measured L1 read hits as a fraction of read requests (i.e., re-use rate) in the RFO memory system. RFO allows for a longer re-use window than either

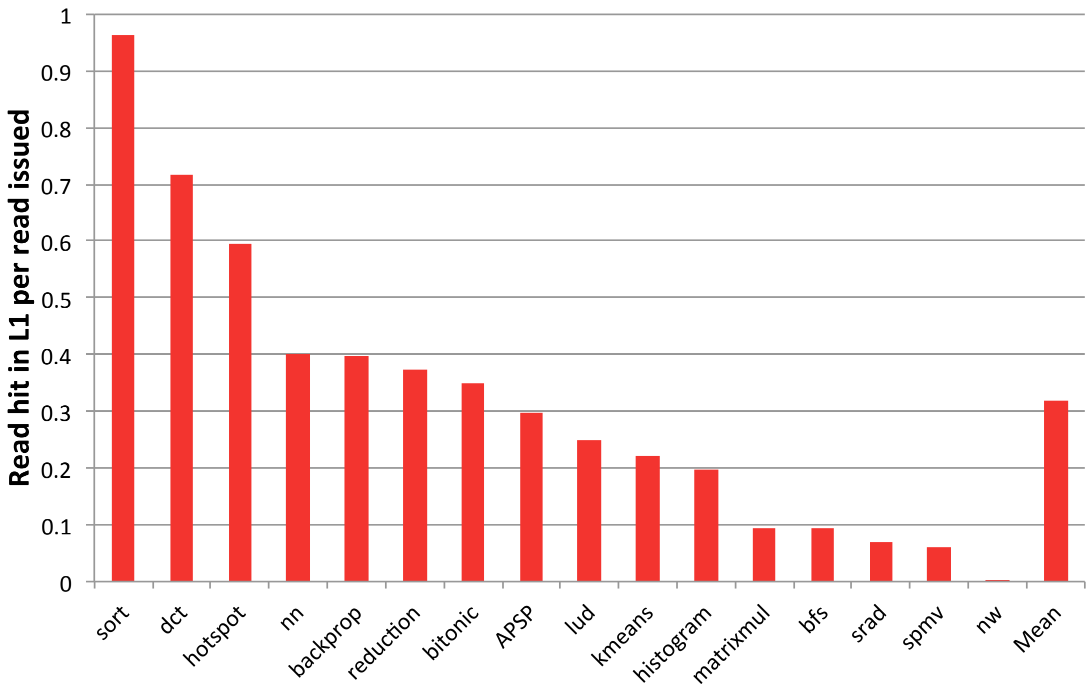


FIGURE 4.4: L1 cache read re-use (read hits per read access in RFO memory system).

the QR or WT memory systems because cache blocks are written only locally and synchronization does not force dirty data to a common coherency point. In contrast, the WT and QR memory systems must ensure all writes are performed to memory before synchronization completes. In addition, WT will invalidate its L1 cache on each kernel launch.

The workloads from Section 4.2 exhibit a huge range of reuse rates, capturing the diverse range of traffic patterns exhibited by GPGPU applications. In either of the extremes of re-use, we expect that all of the memory systems should perform equivalently. In applications with a high re-use rate, L1 cache hits will dominate the run-time. In applications with a low re-use rate, the performance will be bound by the memory bandwidth and latency. Because L1 cache and memory controller designs are effectively equivalent in QR, RFO, and WT, the expected performance is also equivalent.

4.5 Results

4.5.1 Performance

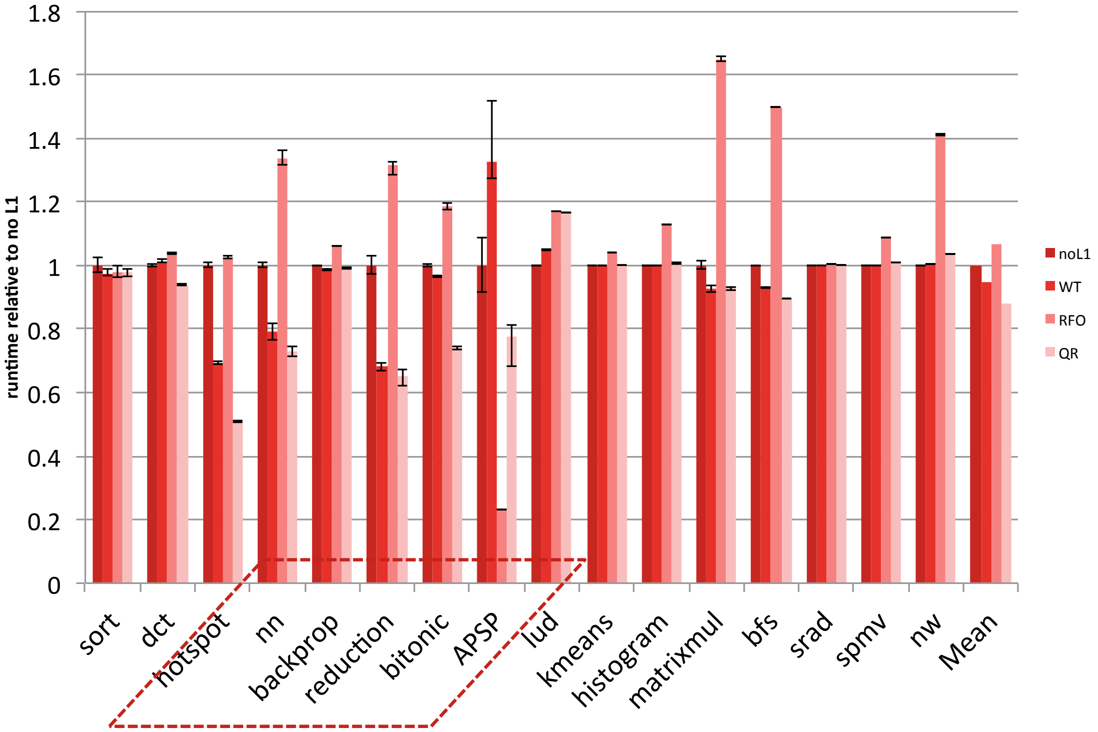


FIGURE 4.5: Relative run-times of WT, RFO, and QR memory systems compared to not using an L1 cache.

Figure 4.5 plots the relative run-times of WT, RFO, and QR relative to a system that disables the L1 cache for coherent traffic, similar to NVIDIAs Kepler architecture. The applications are ordered across the x-axis by their L1 re-use rate (Figure 4.4). The final set of bars shows the geometric mean of the normalized run-times. Overall, QR gains 7% performance compared to WT, which gains only 5% performance compared to not using an L1 cache. On the other hand, the RFO memory system loses 6% performance relative to a memory system with no L1 cache. The RFO performance drop comes from the additional latency imposed to write operations because they first must acquire exclusive coherence permissions.

Figure 4.5 supports the insight that a QR memory system would outperform a WT memory system significantly when there is an intermediate amount of L1 re-use. In particular, QR outperforms WT by 6-42% across six of the seven workloads (dotted-line box in Figure 4.5) because there is significant L1 re-use across kernel boundaries and LdAcqs. In these applications, the WT memory system cannot re-use any data due to the frequency of full cache invalidations. The lone exception is backprop, which is dominated by pulling data from the CPU caches; thus, QR and WT see similar performance.

Across the seven highlighted workloads, APSP is particularly noticeable because of the impressive performance improvement achieved by QR and the even more impressive performance improvement achieved by RFO. APSP is the only benchmark that frequently uses LdAcq and StRel instructions within its kernels. While the QR memory system efficiently performs the LdAcq and StRel operations in a write-combining memory system, the RFO memory system performs the operations much faster at its local L1 cache. The resulting memory access timings for the RFO memory system lead to far less branch divergence and fewer kernel launches compared to the other memory systems because the algorithm launches kernels until there is convergence.

The applications bfs, matrixmul, and dct are on the border between intermediate and high or low re-use. As a result, the performance advantage of QR relative to WT is muted.

Similar to backprop, kmeans and histogram invoke many kernel launches and frequently share data between the CPU and GPU. Their performance also is dominated by pulling data in from the CPU, resulting in QR and WT achieving similar performance.

The one application on which QR encounters noticeable performance degradation is lud. As shown in Figure 4.3, lud exhibits the highest rate of temporal read-after-writes; thus, the extra latency of moving data between QRs separate read and write caches is exposed. Furthermore, lud has a high degree of false sharing between CUs, which lowers the effectiveness of QRs L1 cache compared to WT due to its cache block granular invalidations. Overall, due to its unique behavior, lud is the only benchmark on which simply disabling the L1 cache achieves a noticeable performance improvement relative to the other designs.

The rest of the applications (sort, srad, spmv, and nw) exhibit either very high or very low L1 re-use, which means we would expect a small performance difference due to the on-chip memory system. The results confirm this intuition because all non-RFO memory systems perform similarly.

4.5.2 Directory Traffic

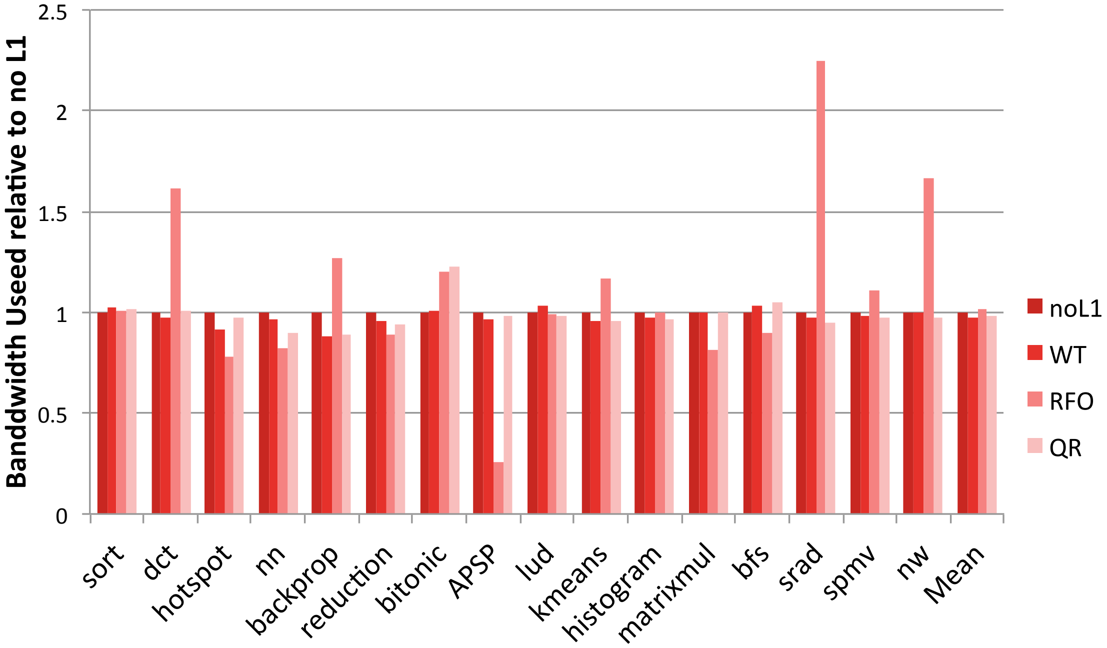


Figure 4.6: L2 to directory bandwidth relative to no L1.

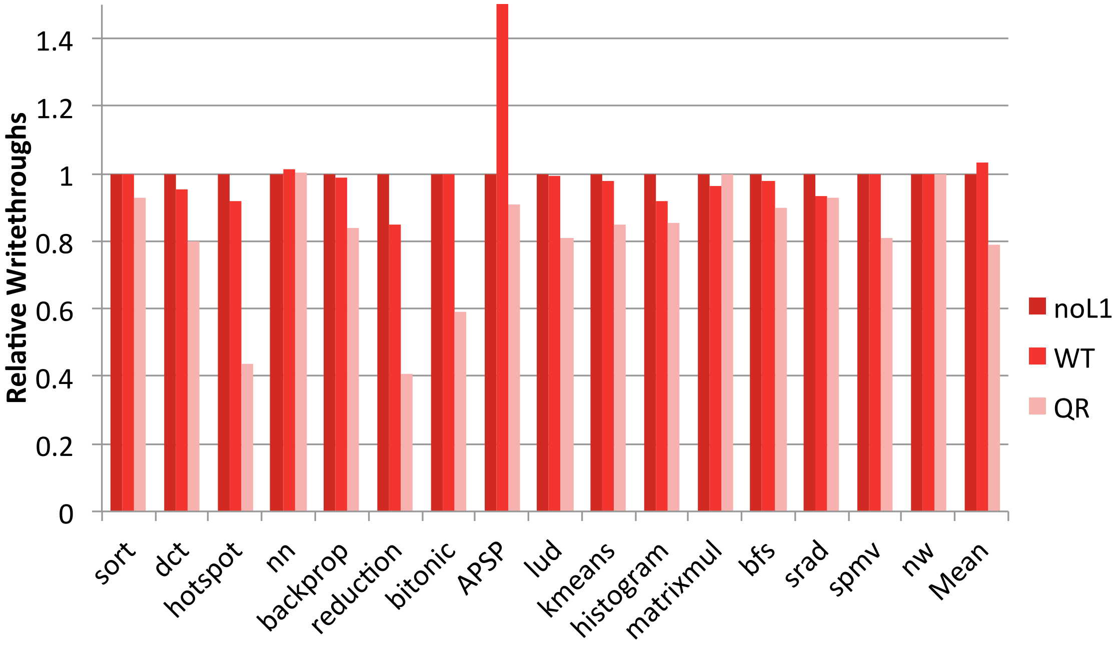


FIGURE 4.7: Write-through requests seen at DRAM relative to a system with no L1.

Figure 4.6 shows the bandwidth between the GPU cache hierarchy and the APU directory for WT, RFO, and QR relative to the system without an L1 cache. Due to aggressive write-combining, QR generates less total write traffic than WT for the same or better performance.

To explore the directory write traffic, Figure 4.7 shows the effectiveness of the write-combining performed by a QR memory system. The RFO memory system includes a memory-side L3 cache, which filters many DRAM writes, so only the no-L1-memory, WT, and QR designs are shown in Figure 4.7. Most applications see significantly fewer write requests at the DRAM in QR compared to a WT or no-L1-memory system due to the write-combining performed at the wL1, wL2, and wL3. As Figure 4.7 shows, applications with the greatest reduction generally achieve the greatest performance gains, indicating that good write-combining is critical to performance. In nn and nw, WT and QR have similar DRAM traffic. In these applications, there is no opportunity to perform additional write-combining in QR because all of the writes are full-cache-line operations and each address is written only once.

4.5.3 L1 Invalidation Overhead

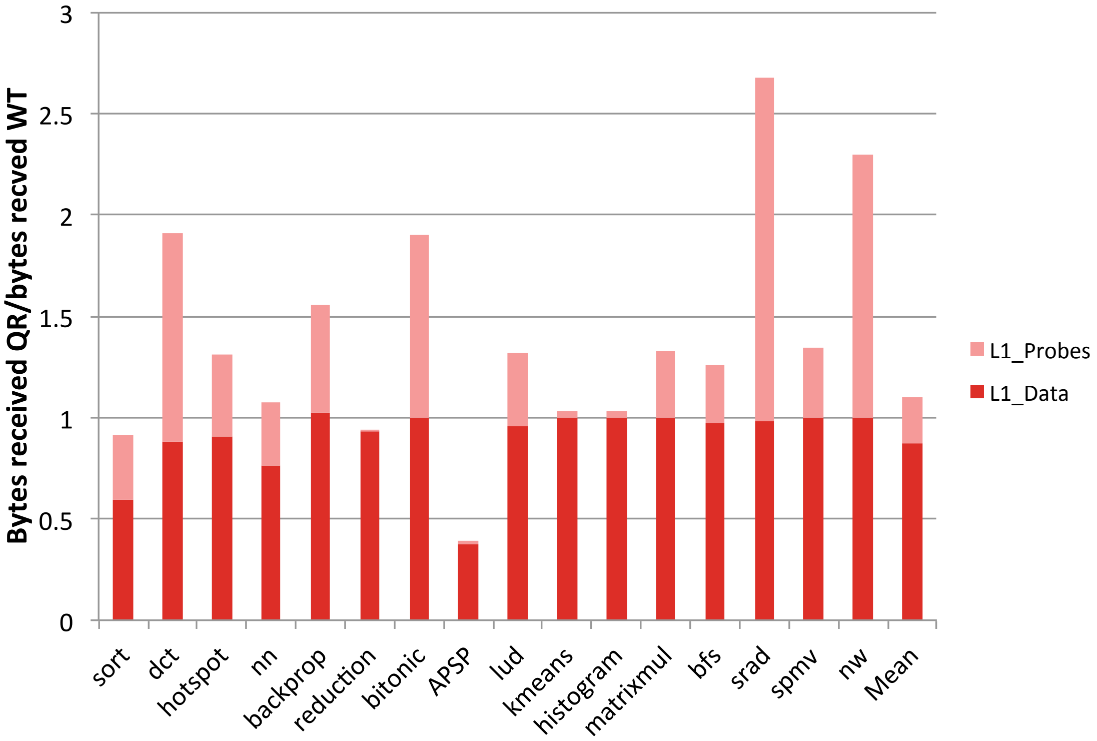


FIGURE 4.8: Invalidation and data messages received at the QR L1 compared to WT data messages.

Figure 4.8 shows both the cost and benefit of broadcasting precise invalidations in QR. Bars represent the normalized number of bytes that arrive at the L1 cache in QR compared to WT. Within each bar, segments correspond to the number of bytes that arrived due to an invalidation probe request or a data response, respectively.

Almost all benchmarks receive equal or fewer L1 data messages in a QR memory system compared to a WT memory system. The only exception is backprop, in which false sharing created additional cache misses for QR due to invalidations after wL2 evictions.

When invalidation traffic is added, the total bytes arriving at the L1 in a QR memory system can be up to three times the number of bytes arriving in a WT system, though on average the number is comparable (103%). Some workloads even experience a reduction in L1 traffic. APSP saw a significant reduction in overall traffic because frequent LdAcqs and the subsequent cache invalidations result in a 0% hit rate at the WT L1. In most workloads, QR and WT have comparable traffic at the L1. QR achieves this comparable traffic despite extra invalidations because it is able to re-use data across kernel boundaries, whereas WTs full L1 cache invalidation cause data to be refetched.

Finally, other workloads see a doubling or more of L1 traffic in QR. This is because they have a significant number of independent writes without re-use between kernels to amortize the cost of invalidations. In the future, we predict that reducing the data required from off-chip likely will trump the cost of additional on-chip invalidation messages, making QR a reasonable design despite this increased L1 traffic.

4.5.4 Total Memory Bandwidth

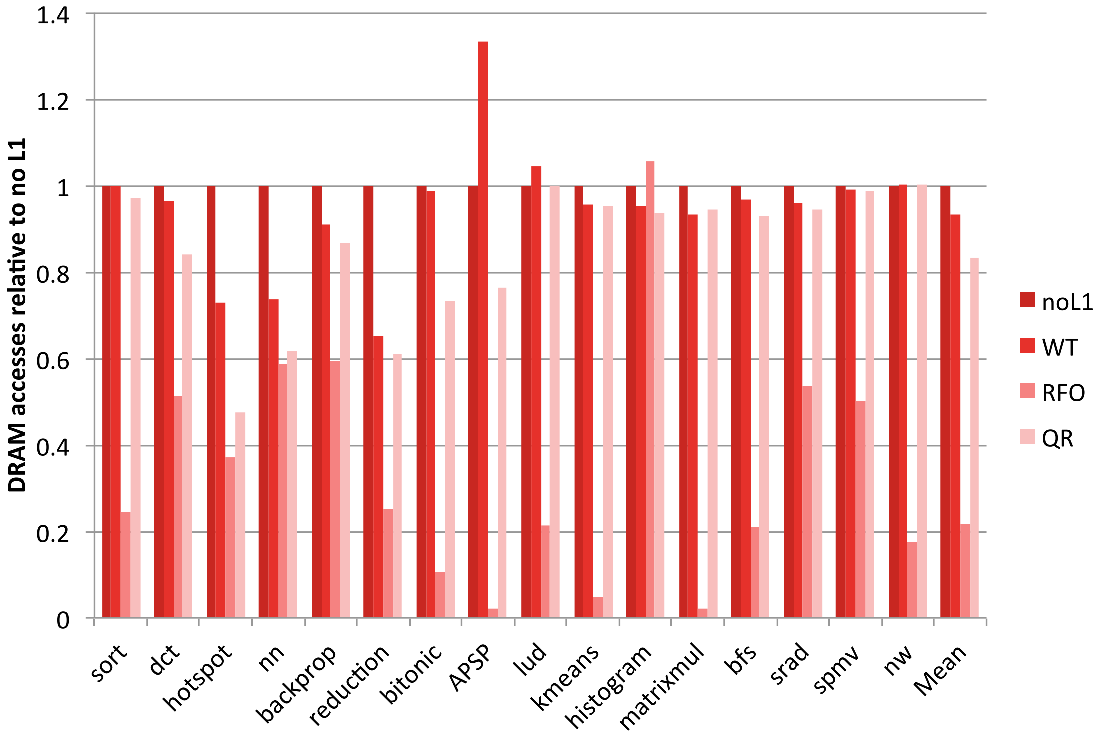


FIGURE 4.9: Total DRAM accesses by WT, RFO and QR relative to no L1.

Figure 4.9 shows the combined number of read and write memory accesses for each benchmark relative to the memory accesses performed by the memory system with no L1. The RFO has fewer memory reads because dirty data is cached across kernel bounds, which is not possible in the QR or WT memory systems because data responses to CPU probes are not supported. This is especially effective because kernels often switch the input and output pointers such that previously written data in the last kernel is re-used in the next kernel invocation.

4.5.5 Power

Combining the results from Figure 4.8 and Figure 4.9, we can estimate the network and memory power of QR and WT. Because GPUWattch showed that memory consumed 30% of power on modern GPUs and network consumed 10% of power [35], we can infer that QR should save 5% of memory power and increase network power by 3%. As a result, it follows that QR should save a marginal amount of power that may be used by the additional write caches. Further, the improved performance of QR relative to WT implies less total energy consumption.

4.5.6 Scalability of RFO

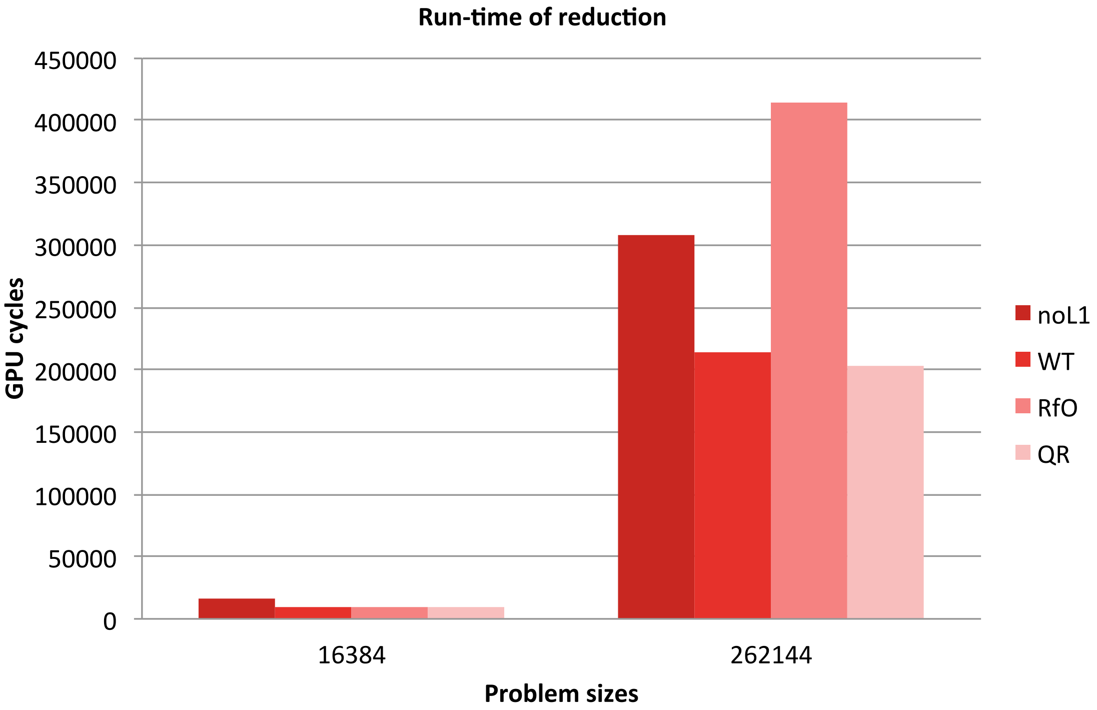


FIGURE 4.10: Scalability comparison for increasing problem sizes of reduction.

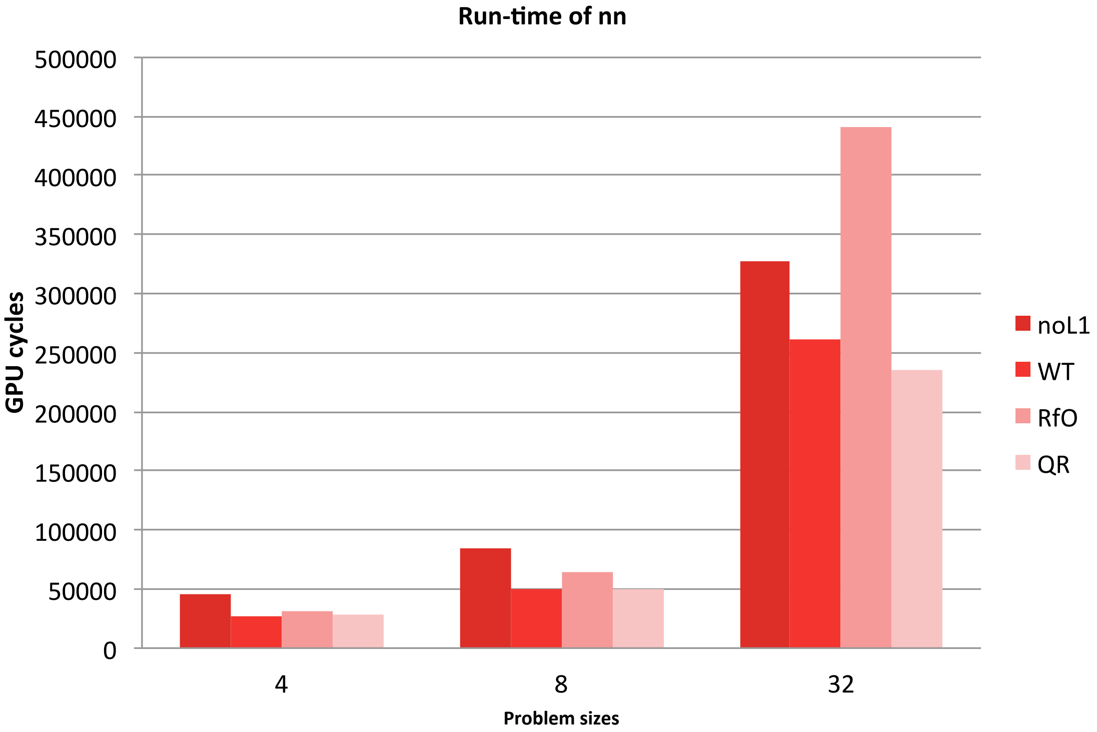


Figure 4.11: Scalability comparison for increasing problem sizes of nn.

To support the claim of increased bandwidth scalability compared to an RFO memory system, nn and reduction are evaluated with smaller inputs to see how well a latency-oriented RFO memory system could perform compared to a throughput-oriented WT or QR memory system. Figures 4.10 and 4.11 shows the performance of reduction and nn for various problem sizes respectively. For small input sets, all memory systems have similar performance. As the input size increases, the demand on the memory system increases and QRs reduced write overhead improves the performance relative to RFO and WT.

4.6 Conclusion

This paper demonstrates that QuickRelease can expand the applicability of GPUs by efficiently executing the fine-grain synchronization required by many irregular parallel workloads while maintaining good performance on traditional, regular general-purpose GPU workloads. The QR design improves on conventional write-combining caches in ways that improve synchronization performance and reduce the cost of supporting writes. First, QR improves performance by using efficient synchronization FIFOs to track outstanding writes, obviating the need for high-overhead cache walks. Second, QR reduces the cost of write support by partitioning the read- and write-cache resources, exploiting the observation that writes are more costly than reads.

The evaluation compares QR to a GPU memory system that simply disables private L1 caches for coherent data and a traditional throughput-oriented write-through memory system. To illustrate the intuitive analysis of QR, it also is compared to an idealized RFO memory system. The results demonstrate that QR achieves the best qualities of each baseline design.

5

Task Runtime With Explicit Epoch Synchronization

5.1 Introduction

GPUs support data parallelism efficiently, but not all problems are data parallel. Non-data parallel problems, which are often called irregular parallel problems, often stress systems to balance load, resolve dependencies, and enforce memory consistency. There is an open question whether GPGPUs can be liberated from the data parallel bottleneck to support general irregular parallel workloads [23]. Though work has achieved good performance on a limited set of irregular parallel workloads on GPUs [13], such advanced techniques are difficult to program. In this work, we seek to solve the problem of irregular parallelism more generally. While irregular parallelism has been extensively studied on CPUs, there are few good approaches to this problem on GPUs.

Irregular parallel programming can be supported with a runtime system that provides a set of operations for generating new work in chunks called tasks and for specifying dependencies between tasks. This type of parallelism is known as task parallelism. Though task parallelism is simple to reason about, it is too high-level to implement directly. To implement an irregular parallel program without a task parallel runtime on GPUs, a programmer would need to be an expert in GPU hardware architectures, parallel programming, and algorithm design. As a result, programmers need a task parallel runtime system that can map task parallel code to execute efficiently on a given platform. Developers of task parallel programs benefit from a runtime system to schedule tasks, balance load, and enforce consistency between dependent tasks. Without a runtime system, these applications would be difficult to program, both in terms of correct functionality and performance. With the runtime system, the programmer only thinks of how to express parallelism constrained by algorithmic measures of complexity.

The performance of a task parallel program is a function of two characteristics: its critical path and its total amount of work to be performed. The work is the execution time on a single processor, and the critical path is the execution time on an infinite number of processors. The maximal level of parallelism is the quotient of the work, T_1, and the critical path, T_∞. The runtime of a system with P processors ,T_P, is bounded by $T_p = O(\frac{T_1}{P}) + O(T_\infty)$ due to the greedy off-line scheduler bound [12, 25].

More precisely, we introduce the work overhead,o_1, and the critical path overhead, o_∞. When a runtime system's overhead are included, the execution time becomes $T_p = o_1 \frac{T_1}{P} + o_\infty T_\infty$. Different implementations of a shared task queue will affect o_1 and the o_∞ differently. For example, pulling a task from a global task queue could require a lock before work can be executed, which can create a large value of o_1. Using a local task queue can avoid the lock, but balancing load will increase o_∞.

The primary example of a task parallel runtime is cilk [9, 47]. Cilk, which targets shared memory CPU platforms, uses work-stealing to schedule tasks and balance load in software. Originally, cilk had no preference for placing runtime overheads for work-stealing on the work or critical path. Cilk-5, however, optimizes performance with its work-first principle that avoids placing overheads in the work even at the cost of additional overhead on the critical path [22]. In the equation above, when $\frac{T_1}{P} \gg T_\infty$, the critical path overhead, o_∞, does not significantly affect the runtime, T_p, compared to the work overhead, o_1. The work-first principle gets its name from optimizing the work overhead first even at the expense of the critical path overhead.

Cilk's work-first principle applied to a work-stealing scheduler requires fine-grain communication between threads, which is acceptable on a CPU but expensive on GPUs. In particular, protecting a local task queue from thieves (threads trying to steal work) requires locks and fences. Locks and fences degrade the performance of a GPU's memory system and thus slow down the execution of work. As a result, the use of locks and fences *on a GPU* violates the work-first principle because o_1 and o_∞ are coupled. To this end, we need a new way to decouple runtime overheads on the critical path from work overheads.

We propose the work-together principle to decouple the runtime overheads on the work and the critical path. The work-together principle extends the work-first principle to state that the overhead on the critical path should be paid by the entire system at once and that work overheads should be paid co-operatively. These tenets contradict the adversarial design of a work-stealing task scheduler. Thus, we propose an new task scheduling technique that can obey the work-together principle.

Since GPUs are not efficient at the fine-grain communication and synchronization required for work-stealing schedulers like that of cilk, we must design a *non-work-stealing* scheduler that executes efficiently on a GPU. This scheduler should leverage the work-together principle to balance load, schedule dependent tasks, and maintain memory consistency. Such a scheduler on a GPU can use hardware mechanisms and CPU integration to amortize these costs. GPUs, unlike CPUs, offer hardware techniques for bulk synchronization that enable many threads to work together to pay for scheduling dependencies and memory consistency in hardware and thus obey the work-together principle.

We implement this work-together principle in a Task Runtime with Explicit Epoch Synchronization (TREES). In TREES, computation is divided into massively parallel epochs that are synchronized in bulk. These epochs are the critical path of a task parallel program in TREES. TREES provides an efficient and high-performing backend for task parallel programs that works well on current GPUs. We suspect the performance advantage of TREES on future GPUs and heterogeneous systems will increase because the bulk synchronization overheads are shrinking and core counts are increasing. TREES can handle the execution of a vast number of tasks with complex dependency relationships with a small effect on the expected runtime given program. To this end, TREES helps to achieve the theoretical speedup ($\frac{T_1}{T_P} = O(P)$) that a P-processor GPU could provide for a task parallel algorithm.

To achieve this performance with the work-together principle, TREES amortizes the cost of fine-grain fork and join operations using hardware mechanisms for bulk synchronization. Further, TREES leverages tight coupling with the CPU to remove work overhead due to software scheduling on the GPU. To limit the critical path overhead of communication between the CPU and GPU, TREES summarizes the fork and join operations in stages of computations called epochs. The GPU communicates the number of forks and join using a single cache block of data transferred back to the CPU. The CPU schedules a stack of epochs that represent a breadth-first expression of the dynamic task dependency graph generated by a task parallel program. These design decisions enable the GPU hardware to perform load-balancing, the software to lazily enforce memory consistency at epoch boundaries, and the CPU to schedule dependent tasks in bulk on the GPU's behalf. As a result TREES obeys the work-together principle, and provides the programmability of fork/join parallelism with the power of a GPU.

In this work we make the following contributions:

- We propose the work-together principle to guide the design of irregular parallelism on GPUs
- We develop a new runtime system, TREES, that efficiently supports task parallelism on GPUs using the work-together principle.
- We show how TREES can provide competitive performance to CPU task parallel runtimes.
- We experimentally evaluate the performance of TREES and show that its generality comes at a very low cost.

The remainder of this chapter first describes fork/join parallelism in Section 5.2. Sections 5.3 and 5.4 delve into detail on the work-first principle and work-together principle respectively. Section 5.5 describes how TREES implements the work-together principle in detail. Section 5.6 presents a series of case studies that motivate TREES and the work-first principle.

5.2 Fork/join parallelism

Fork/join parallelism makes it easy to turn a recursive algorithm into a parallel algorithm where independent recursive calls can be forked to execute in parallel. Often forks apply directly to the divide stage in a divide-and-conquer algorithm. Any task parallel runtime will incur overhead for the implementation of a fork operation in the work and critical path of the program. Optimizing fork operations is important because all work that is performed must first be created.

Join operations wait for forked tasks to complete before executing. Joins often perform the conquer in a divide-and-conquer algorithm where a reduction needs to be performed on the results of independently forked tasks. Joins must be scheduled after all forked operations complete. Scheduling a join will incur overhead to ensure the completion and memory consistency of forked tasks. This consistency and notification of completion has the potential to add overhead to the work of an algorithm.

One can use the master theorem to understand the complexity of the the work and the critical path [38]. When calculating the critical path latency we only look at operations that cannot be executed concurrently. When calculating the work, we ignore the fork operations and assume they execute sequentially.

Dividing the work and the critical path gives the maximum available parallelism of an algorithm. It is important to balance this available processing between processors in a balanced manner to avoid unecessary serialization of execution. However, it is important that the runtime does not impose a large overhead on the work (o_1) of the algorithm in order to balance the load since the runtime (T_p) on P-processors can be estimated by $T_p = o_1 \frac{T_1}{P} + o_\infty T_\infty$.

5.3 Work-first principle

The work-first principle states that overhead should be on the critical path instead of the work of the algorithm. The reason here is that the algorithm's parallelism ($\frac{T_1}{T_\infty}$) is generally much greater than the hardware can provide (P). As a result, applying overhead (o_1) to this parallelism is extremely costly. However, overhead to balance load ($o_\infty T_\infty$) will be incurred on the critical path [22] and be proportional to the number of processors in the system because steal operations that balance load are proportional to PT_∞. To this end, the critical path can afford a much higher overhead than the work to maintain linear speedups with increasing core counts (approximately $\frac{T_1}{P}$) as long as $o_1 \frac{T_1}{P} \gg o_\infty T_\infty$.

The work-first principle enables nearly linear speedups with increasing core counts, although these speedups are bounded by the critical path. The bounded linear speedup is competitive with a greedy offline schedule ($T_p = O(\frac{T_1}{P}) + O(T_\infty)$). Since the work-first principle applies to an online system, the greedy offline schedule is an optimal lower bound on the execution time.

In a work-stealing algorithm, the work-first principle places the overhead on thieves attempting to steal more work . Placing the overhead on the thieves is derived from the fact that it can be shown that number of steal operations is bounded by the product of the number of processors and the critical path ($O(PT_\infty)$). This overhead is performed in parallel and thus the overhead for stealing is bounded by the critical path ($O(T_\infty)$). Moreover, synchronization overheads of the fork and join operations should be only be incurred if a thief can contend with a local worker. Cilk-5 does this with the THE synchronization protocol where pushing a task to a local task queue requires only a non-atomic increment and pulling a task from a local task queue only uses a lock if a thief is competing for the same task [22].

5.4 Work-together Execution Model

The work-together principle, like the work-first principle, is intended for runtime systems that support fork/join parallelism. However, unlike the work-first principle —which is primarily intended for CPUs—the work-together principle leverages a GPU's ability to synchronize in bulk and co-operate on memory operations to reduce runtime overheads. The critical difference between CPUs and GPUs is that memory fences can degrade the performance of the entire GPU. As a result, memory fences will not only lengthen the critical path, they will interfere with the performance of the work. To correct the work-first principle for interference and support task parallelism on GPUs, the work-together principle considers the overhead of the runtime due to interference from memory fences.

The work-together principle can be stated in two tenets:

- Pay critical path overheads at one time to avoid interfering with the execution of work.
- Co-operatively incur the runtime system's work overhead to reduce the impact on execution time.

To visualize the work-together principle, we present an abstract machine, the Task Vector Machine (TVM), that creates a formal model for fork/join parallelism on GPUs. As a result, the TVM helps reason about the computational and space complexity of a task parallel program executing with the work-together principle. Further, we use the TVM as the basis for implementing TREES which obeys the work-together principle.

5.4.1 Work-together principle

The work-together principle extends the work-first principle to consider a GPU's strong coupling between threads with the two tenets listed above. Without extending the work-first principle, it is easy to not fully consider the challenges of supporting fork/join parallelism on GPUs that have vastly differernt execution characteristics to a CPU. The first tenet leverages hardware support for global synchronization and the second tenant leverages SIMD and support for memory coalescing. Many of the challenges associated with programming a GPU would apply in implementing a task parallel runtime for other accelerators.

A requirement of the first tenet is that the overhead on the critical path does not vary with number of cores or effect the work. To apply this tenet to fork/join parallelism, a runtime will expand parallelism in a breadth-first manner and execute each level of the generated task dependency graph in a bulk-synchronous manner. This will be more precisely described in Section 5.4.2 with an abstract machine to reason about a work-together interpretation of a task parallel program. By executing a task parallel program in this manner, the GPU hardware can handle load balancing and memory consistency in the same way it does for graphics operations, in bulk; further, the CPU can resolve task dependencies in bulk on the GPU's behalf. The aforementioned operations are all paid at once along the critical path.

A requirement of the second tenet of the work-together principle is that the work overhead is reduced by performing the same operation at the same time. To apply this tenet, a runtime should leverage SIMD operations to reduce the overhead of fork and join operations by the SIMD width. A well designed runtime will make a best effort to ensure that forking, joining, and finding tasks leverage memory coalescing and avoid branch divergence. Further, atomic operations should be kept to a minimum and SIMD units should combine atomic operations using a GPU's local memory.

The two tenets of the work-together principle provide the same performance bound as the work-first principle ($T_p = o_1 \frac{T_1}{P} + o_\infty T_\infty$). However, the critical path overhead is no longer proportional to the number of processors as it is with work-stealing; in fact, the critical path is constant with both work and number of processors. Since the work-together principle does not require fine-grain synchronization, a work-together runtime avoids work overheads due to interference caused by the memory fences. On GPUs, interference results from the implementations of memory fences that flush caches and halt core execution for work-items that share a compute unit.

The work-together principle enables task parallelism on a GPU because the hardware provides efficient mechanisms for bulk-synchronous operations (first tenet) and coalescing memory operations (second tenet). The bulk-synchronous operations, further, can be used to guarantee memory consistency and the completion dependent of tasks level-by-level. Load balancing can be performed efficiently by built-in hardware. Finding tasks, forking a new task, and scheduling joins can leverage the SIMD width and memory coalescing to reduce the overhead incurred by these operations on the work.

5.4.2 TVM : Thinking about work-together

The Task Vector Machine (TVM) is an abstract machine with N cores that enables directly understanding the execution of a task parallel program using the work-together principle. The TVM contains a N-wide vector of tasks (function name and arguments), or Task Vector (TV), whose execution is predicated by a stack of N-wide execution masks, or Task Mask Stack (TMS) as shown in Figure 5.1. Each execution mask in the TMS is what will be called an *epoch* of an algorithm's critical path and the nubmer of valid bits to ever be in the TMS represent the work of an algorithm.

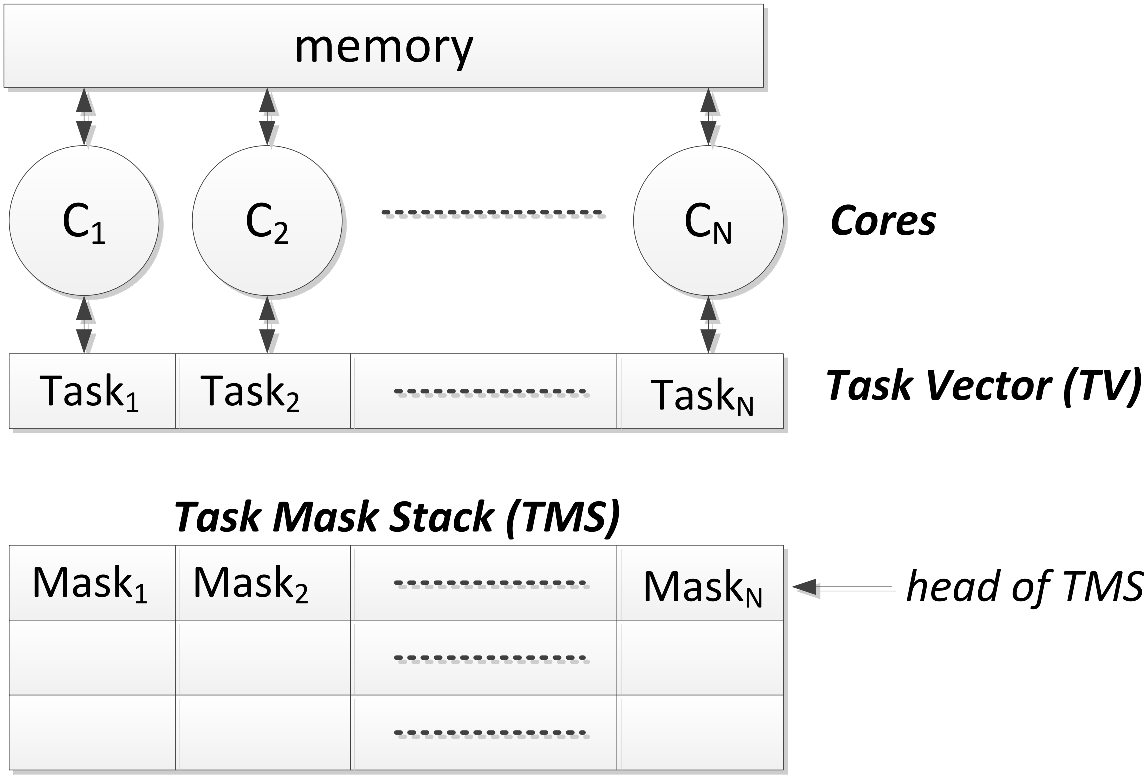


FIGURE 5.1: The Task Vector Machine (TVM)

TVM interface Program

```
void visit(node) {
        //DO VISIT
}
task void preorder (node) {
        if (node != nil) {
                visit(node)
                fork preorder(node.right);
                fork preorder(node.left);
        }
}
task void postorder (node) {
        if (node != nil) {
                fork postorder(node.right);
                fork postorder(node.left);
                join visitAfter(node);
        }
}
task void visitAfter(node) {
        visit(node);
}
```

FIGURE 5.2: Preorder and postorder tree traversal on TVM.

Figure 5.2 gives an example of programming to the TVM interface that traverses a TREES in preorder or postorder. The key operations are **fork**, **join**, and **emit** that create a new task, wait for created tasks to complete, and return a value respectively. Currently TVM interface programs require explicit continuation passing [55] like the original cilk [9]. In the future, we would expect that a variety of task parallel programming languages could be compiled into the TVM interface.

Data Parallel Tasks :

A TVM interface cannot entirely work-together without the ability to leverage the SIMD nature of a target platform. Since GPUs have a high-bandwidth and low-latency local memory available to work-groups, a data parallel **map** operation will launch a workgroup to execute part of a task parallel program. Using **map** can incur a much smaller overhead per amount of computation performed than a **fork** operation.

TVM execution model

Each *epoch* of TVM execution can be divided into three parts and an initialization stage.

Initialize:

When the TVM is constructed, it is parameterized by N, and an initial task to start executing. The TMS and TV start out empty (full of zeros) and an index to next available entry in both the TV and TMS is set to zero. When the initial task is added to the TVM: the next available entry is incremented to one; a new execution mask is pushed on to the TMS containing a one at index zero and a 0 in the rest of the mask; and index zero of the TV is filled with the function and arguments for the initial task (*postorder(root)* in figure 5.2). At this point the TVM will proceed to execute phase 1.

Phase 1:

When the TVM enters phase 1: an execution mask is popped from the head of the TMS; a mask for fork operations (i.e. fork mask) is reset to all zeros; and a mask for join operations (i.e. join mask) is reset to all zeros. At this point the TVM proceeds to phase 2. If the TMS was empty when popped, execution is complete and the TVM is destructed instead of proceeding to phase 2.

Phase 2:

When the TVM enters phase 2 each core checks the execution mask for a value of one. If a core reads a value of one then it executes the task specified in the TV (for example *postorder(root)*). During the execution of a task a core may: **fork** an new task (i.e. **fork** *postorder(node.right)*); schedule a **join** to continue after forked task have completed (i.e. **join** *visitAfter(node)*); return a value to a parent task with **emit**; schedule a data parallel task with **map**; and access memory. After all cores have check their execution mask and ,if valid, executed the task in their indexes into the TV the TVM continues to phase 3.

Fork:

When a core calls **fork**, that core atomically increments the index to the next available index into the TV and TMS. The return value of the atomic, or fork index, of the atomic operation is associated with a core that has an empty TV and TMS entry. To complete the fork operation: the fork mask at the fork index is set to one; the TV entry at fork index is filled with the specified function and arguments (*postorder(node.right)*).

Join:

When a core calls **join**: the core resets its entry in the TV to a new task to execute after any forked task (i.e. *visitAfter(node)*); the core sets the index into the join mask at its index to be one; and the core terminates execution of the current task.

Emit:

When a core calls **emit**, the core resets its entry in the TV to hold the return value of that task and terminates execution of the current task.

Map:

When a core calls **map**, a data parallel task is executed asynchronously before the next *epoch* begins.

Phase 3:

When the TVM enters phase 3: the TVM performs a population count on the join mask and if the count is non-zero, the join mask is pushed on to the TMS; then the TVM performs a population count on the fork mask and if the count is non-zero, the fork mask is pushed onto the TMS; and finally, the TVM executes any **map** operations specified during phase 2. The TVM proceeds back to phase 1 to begin the next *epoch*.

Time Complexity

The time complexity of the execution of a program on the TVM can, like a task parallel program, can be broken into the critical path (T_∞) and work (T_1). The critical path is the number of *epoch*s. On an ideal TVM with $O(T_1)$ cores, the execution time is $O(T_\infty)$. The work of the algorithm is the sum of all times the execution Mask has a non-zero value. Since the TVM targets a GPU, the runtime on system with P-processors that are each W-wide SIMD would be a decent approximation of a GPU. Such a system would yield an execution time to be $T_{P,W} = o_1\frac{log(W)T_1}{PW} + o_\infty T_\infty$, because we pessimistically expect an average divergence penalty to be $log(W)$. The best case execution time is when the SIMD width executes without divergence and $T_{P,W} = o_1\frac{T_1}{PW}) + o_\infty T_\infty$. Alternatively, the execution can be upper bounded by the maximum nesting of branches (D) where ($2^D < W$) to be $T_{P,W} = o_1\frac{2^D T_1}{PW} + o_\infty T_\infty$.

Space Complexity

Each core in the TVM requires space for a function, arguments, and a stack of bits. As a result, the space complexity of an algorithm running on the TVM is upper-bounded by the work ($O(T_1)$) and lower bounded by the parallelism ($\Omega(\frac{T_1}{T_\infty})$) of an algorithm. The upper bound holds because each task can only use one function, one set of arguments, and one stack of bits that each require memory to compute the work. The lower bound holds because there needs to be at least one function, one set of arguments, and one stack of bits for each active task (parallelism). The TV must be sized to the work of the problem for divide and conquer problems, because there are $O(T_1)$ join operations to perform the conquer operations. The lower-bound occurs when there are no join operations like in a graph traversal. This means that, there only need to be enough TV entries for the maximal amount of the graph traversed.

5.4.3 Current work-together systems

Recent work in irregular parallel programs on GPUs can be interpreted to abide by the work-together principle. These programs take a data-driven rather than topology driven approach to graph algorithms [42] where there is an input worklist and an output worklist. The input worklist is read in a coalesced fashion and synchronization of completing writes to the output worklist is paid at kernel boundaries. Since this data-driven technique performs competitively with a topological approach, this bodes well for the applicability of the work-together principle for a wider array of graph problems.

5.5 TREES: Work-together on GPUs

The work-together principle has the ability to enable efficient fork/join parallelism on GPUs. In this section, we show how to implement a runtime that obeys the work-together principle. The Task Runtime with Explicit Epoch Synchronization (TREES) implements the TVM execution model described in Section 5.4.2. A source to source compiler converts a TVM interface program into a OpenCL program that contains the TREES runtime. Technically, TREES could be implemented in another heterogeneous programming language, but OpenCL provides portability.

TREES gets its name from from the use of the same epochs from the TVM. Each epoch runs to completion before another epoch is executed. As a result, the use of epochs explicitly synchronizes all dependent tasks without individually updating task dependencies by using the last-in-first-out behavior of the TMS. TREES executes each epoch in the same set of phases as the TVM and TREES initializes the TVM in the same way.

5.5.1 TVM to TREES

Task Vector:

Structurally, TREES represents the task vector with a structure of N-wide arrays of integers. The functions are represented as an array of N enumerations of the tasks in a TVM interface program. The arguments are split in to 4 byte chunks as an array of N integers. The arrangement in a structure of arrays enables memory coalescing when reading or writing any of the arrays in the TV. The arrays are allocated in the GPU's memory space.

Task Mask Stack:

TREES replaces the TMS with an epoch number (EN) that is used to encode the enumerations in the array of functions in the TV to created an array of epoch encoded functions or entries ($entry = function + EN * NumFunctions$). When an entry is decoded ($entry - EN * NumFunctions$) only valid functions are executed on the GPU. The host CPU takes on the role of managing which epoch number needs to be executed, where the CPU uses a kernel argument to pass the epoch number to the GPU. Instead of a TMS, the CPU maintains a stack of epoch numbers, or join stack.

Cores:

TREES represents each TVM core as an OpenCL work-item and each epoch as a kernel launch with an NDRange that holds at least the TVM cores that can execute their TV entry in the current epoch. The work-items are split into work-groups of 256 work-items that are scheduled to GPU compute units with a GPU hardware scheduler. The NDRange of each epoch depends on fork, join and map operations and is maintained in an NDRange stack that parallels the join stack.

Forks, Joins, and Maps:

To handle forks, joins, and maps, TREES maintains three shared values that are transferred to the GPU between phase 1 and phase 2. After phase 2 completes, these values are transferred back to the CPU. One value is the next available slot in the task vector, **nextFreeCore**. Another is whether a join was scheduled in the last epoch,

joinScheduled. The third value is whether any data parallel map operations were scheduled in the last epoch, **mapScheduled**. To fully represent fork operations, the value of **nextFreeCore** is saved in phase 1 to **oldNextFreeCore**.

5.5.2 Initialize

To initially fill to TV, the CPU launches a single task to the GPU to fill the TV with the first task specified by the TVM interface program (i.e. *postorder*(*root*)). Using a kernel on the GPU prevents the entire TV from being copied from the CPU memory space into the GPU memory space.

At this point the epoch number is zero, **nextFreeCore** is one, **oldNextFreeCore** is zero, the join stack is empty, **joinScheduled** is zero, and **mapScheduled** is zero. TREES pushes epoch number 0 onto the join stack and pushes and NDRange containing work-items 0 through 1 onto the NDRange stack. TREES proceeds to phase 1 of execution.

5.5.3 Phase 1

Determine Epoch Number and NDRange:

To determine the epoch number, TREES pops the join stack. To determine the NDRange associated with that epoch, TREES pops the NDRange stack. If the join stack is empty, the TVM interface program is complete.

Prepare Shared Variables:

Before TREES can enter phase 2: **mapScheduled** and **joinScheduled** are set to zero; **oldNextFreeCore** is set to the current value of **nextFreeCore**; and the values of **oldNextFreeCore**, **joinScheduled**, and **mapScheduled** are transferred to the GPU. TREES then enters phase 2.

5.5.4 Phase 2

CPU

TREES sets up an OpenCL kernel to execute the current state of the task vector with the epoch number and NDRange determined in phase 1. The CPU enqueues the kernel and waits for the GPU to complete the execution of the epoch. After waiting for the GPU to complete the execution of an epoch, the CPU enqueues a transfer of **nextFreeCore**, **joinScheduled**, and **mapScheduled** from the GPU memory space back the CPU memory space. At this point the CPU waits for the GPU to complete the execution of the enqueued kernel.

GPU

Each work-item in the current NDRange loads its entry from the Task Vector and decodes the entry with the epoch number to determine what function to execute with an if-else tree. These work-items will execute a non-recursive function that can access memory, fork a new task, join a continuation, return a value, and schedule a data parallel task. When all 256 work-items in a workgroup complete, a new workgroup is launched on that compute unit with the hardware scheduler. The hardware scheduler provides load balancing at very little cost to the work. Entering the driver to launch the kernel and transferring the shared variables create the critical path overhead. Trends in GPU hardware and drivers suggest that these overheads will become ever smaller.

Fork:

When a core calls a fork: the core atomically increments **nextFreeCore** using a local memory reduction to ensure a single atomic operation per wavefront; each core uses the return value of the atomic to index into a new slot in the task vector; each core uses a coalesced write to the TV to set the entry to a function enumeration encoded with the current epoch number plus one ($entry = function + (EN + 1) *$

$NumFunctions$); and each uses a coalesced write for each argument in the TV to set all of the arguments to the function that was forked.

Join:

When a core calls a join: the core sets **joinScheduled** to one that is coalesced automatically across the wavefront; each core uses its own index to write the task vector; each core uses a coalesced write to the TV to set the entry to a function enumeration encoded with the current epoch number ($entry = function + EN * NumFunctions$); and each uses a coalesced write for each argument in the TV to set all of the arguments to the function that will continue after forked tasks execute.

Emit:

When a core calls a join: each core uses its own index to write the task vector; each core uses a coalesced write to the TV to set the entry to zero; and each uses a coalesced write for each return value in the TV to set all of the arguments slots so that a parent's join can use those values.

Map:

When a core calls a map: the core atomically increments **nextFreeCore** using a local memory reduction to ensure a single atomic operation per wavefront; each core also sets **mapScheduled** to one so that can be read later; each core uses the return value of the atomic to index into a new slot in the task vector; each core uses a coalesced write to the TV to set the entry to a function enumeration encoded to only be valid in a kernel that launches map operations ($entry = map_function - NumFunctions$); and each uses a coalesced write for each argument in the TV to set all of the arguments to the function that was forked. Currently it would seem trivial to include fork, join, emit, and map operations, but future implementations can use dynamic parallelism for this operation which would separate the execution of these tasks from the execution of the TVM.

5.5.5 Phase 3

TREES checks the value of **joinScheduled**; if the value is one, the current epoch number and NDRange is pushed onto join stack and the NDRange stack respectively. TREES then compares **oldNextFreeCore** and **nextFreeCore**; if the values differ an epoch number one greater than the current is pushed onto the join stack and an NDRange from **oldNextFreeCore** to **nextFreeCore** is pushed onto the NDRange stack. TREES checks the value of **mapScheduled** and, if it is one, launches a kernel consisting of the data parallel map operations scheduled during the last epoch. This kernel runs to completion before TREES returns to phase 1.

5.5.6 TREES Example

This section steps through the state of TREES running a TVM interface program traversing a tree in postorder. The tree is shown in Figure 5.3. Table 5.1 shows the state of TREES as the epochs progress. The TV and TMS column **bolds** the entries valid to execute in the epoch specified on that row due to a one in the execution mask. Initially the system performs a postorder traversal of the root node, A. This will **fork** the postorder traversal of nodes B and C and **join** a visit to node A. The traversal will execute on the GPU which will atomically increment the counter to the next free slot in the TV and initialize each of those entries to do a traversal of B and C. Further, the execution also writes the schedule join variable to true which tells the CPU push epoch 0 onto its stack of join operations. This same process occurs until epoch 3. At this point, no new tasks are forked and no joins were scheduled. As a result, the next free entry can be moved back to slot 7 and the head of the stack of joins can be popped to determine the next epoch. This process continues until the stack of joins is empty and the program is complete.

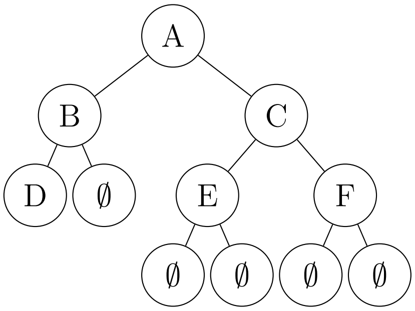


FIGURE 5.3: Example binary tree with 6 nodes

Table 5.1: Postorder tree traversal in TREES of Figure 5.3 where taskType of postorder = 1 and visitAfter = 2

Epoch	TV and TMS = $\frac{2*epoch+taskType}{node}$	next free entry	schedule join?	stack of joins
0	$\vert\frac{\mathbf{1}}{\mathbf{A}}\vert$	$1 \to 3$	$false \to true$	$[] \to [0]$
1	$\vert\frac{2}{A}\vert\frac{\mathbf{3}}{\mathbf{B}}\vert\frac{\mathbf{3}}{\mathbf{C}}\vert$	$3 \to 7$	$false \to true$	$[0] \to [1,0]$
2	$\vert\frac{2}{A}\vert\frac{4}{B}\vert\frac{4}{C}\vert\frac{\mathbf{5}}{\mathbf{D}}\vert\frac{\mathbf{5}}{\mathbf{E}}\vert\frac{\mathbf{5}}{\emptyset}\vert\frac{\mathbf{5}}{\mathbf{F}}\vert$	$7 \to 13$	$false \to true$	$[1,0] \to [2,1,0]$
3	$\vert\frac{2}{A}\vert\frac{4}{B}\vert\frac{4}{C}\vert\frac{6}{D}\vert\frac{6}{E}\vert\frac{6}{\emptyset}\vert\frac{6}{F}\vert\frac{\mathbf{7}}{\emptyset}\vert\frac{\mathbf{7}}{\emptyset}\vert\frac{\mathbf{7}}{\emptyset}\vert\frac{\mathbf{7}}{\emptyset}\vert\frac{\mathbf{7}}{\emptyset}\vert\frac{\mathbf{7}}{\emptyset}\vert$	$13 \to 7$	$false \to false$	$[2,1,0] \to [2,1,0]$
2	$\vert\frac{2}{A}\vert\frac{4}{B}\vert\frac{4}{C}\vert\frac{\mathbf{6}}{\mathbf{D}}\vert\frac{\mathbf{6}}{\mathbf{E}}\vert\frac{\mathbf{6}}{\emptyset}\vert\frac{\mathbf{6}}{\mathbf{F}}\vert$	$7 \to 3$	$false \to false$	$[2,1,0] \to [1,0]$
1	$\vert\frac{2}{A}\vert\frac{\mathbf{4}}{\mathbf{B}}\vert\frac{\mathbf{4}}{\mathbf{C}}\vert$	$3 \to 1$	$false \to false$	$[1,0] \to [0]$
0	$\vert\frac{\mathbf{2}}{\mathbf{A}}\vert$	$1 \to 0$	$false \to false$	$[0] \to []$

5.6 Experimental Evaluation

The goal of this results section is to show that GPUs can be used for a set of applications conventionally believed to be a bad idea for GPUs. The proof of TREES impressive performance is exemplified when comparing to current GPU and CPU parallel programming techniques. The benefits of TREES are exemplified in three case studies that point out the flexibility and performance of TREES implementing the TVM. We focus on building intuition rather than a large number of benchmarks that show the exact same point. Other workloads were implemented and show no more interesting behavior than the ones shown here. Further, we anecdotally analyze how easy it is to program the TVM interface.

The first case study shows that fork/join parallelism in TREES outperforms fork/join parallelism in the cilk-5 runtime on CPUs while being limited to the same power and memory constraints. The second case study shows that support for task parallelism in TREES enables competitive performance on emerging graph algorithms based on work-lists. Finally, the third case study shows that TREES can perform similar to native OpenCL on regular data parallel operations with the use of data parallel map operations. In the end, this evaluation shows that the TVM provides an expressive interface that is efficiently implemented with TREES.

5.6.1 Programming the TVM interface

Beyond these case studies, we anecdotally analyzed the expressiveness of a program using the TVM interface with undergraduate programmers with no parallel programming experience. Many of the undergrads had only taken undergraduate computer architecture and introduction to data structures and algorithms. We found that undergrads were able to write a task parallel nqueens, matrix multiply, traveling salesman, breadth-first search, and simulated annealing. Many of these programs were done with 10-20 hours of work. Most of this work was understanding the algorithm. Implementation on the GPU generally took less than 2-4 hours depending on the state of the runtime. The original undergrads needed to write code directly in OpenCL. After creating a compiler from the TVM interface, programming sped up dramatically. By comparison, I took around 20 hours to create the first TREES prototype application that computed Fibonacci numbers. The next program which was a simple merge sort took me about 1 hours to modify the OpenCL from the Fibonacci. The FFT, bfs, and sssp in case studies 1 and 2 each took less than 1 hour to implement after figuring out the algorithms.

5.6.2 Case Study Methodology

Case studies 1 through 3 will be evaluated with the AMD A10-7850k APU the exemplifies the type of system that is the best case for TREES. This is the first chip to support shared virtual that allows for low latency kernel launch and memory transfers. All experiments are run with the Catalyst 14.1 drivers on the Linux 3.10 kernel with OpenCL 1.2.

5.6.3 Case Study 1: Outperforming CPUs running cilk

In this section, we compare the TVM and TREES on GPU with both a sequential CPU and a CPU running cilk. We will look into two extreme cases, the first is a naive implementation of calculating Fibonacci numbers and the latter is calculating an FFT. The Fibonacci example contains many tasks with almost no work to do, while the FFT contains many tasks that actually perform a significant amount of computations. We will present results for both, the time spent in computation as well as the time spent running the entire program (including OpenCL compilation and initialization overheads).

Figure 5.4 shows results for Fibonacci because it stresses a runtime system to consist almost entirely of overhead. When excluding the initial OpenCL overheads in TREES, we find that the GPU can outperform the parallel performance of cilk with 4 processor. Excluding the OpenCL overheads is reasonable since it is not related to the benefits TREES provides. Since the relative performance does not vary with problem size, TREES balances load similarly to the cilk-5 runtimes. When considering the OpenCL overheads TREES does perform worse than cilk. To overcome the initialization overheads we evaluate a task parallel program with more computation.

FFT performs a significant amount of computation and we evaluate a sequential, cilk, and TREES implementation of FFT. In all cases we separate the computational section of the code from the entire program runtime in Figure 5.5. We also show the entire runtime of the program in Figure 5.6. In this case study we do not use data parallel map operations. When excluding initialization costs, TREES always outperforms the sequential and cilk implementation. When including initialization costs, an FFT must be larger than 1M to see a benefit from using the GPU.

These experimental results are particularly compelling because both the CPU and GPU in this example are constrained to the same memory bandwidth and power supply. In fact, the GPU and CPU use up similar die area on the APU. Since the speedup using only the GPU for this task parallel computation exceeds 8x over the sequential version, it is useful to have placed the GPU on the chip.

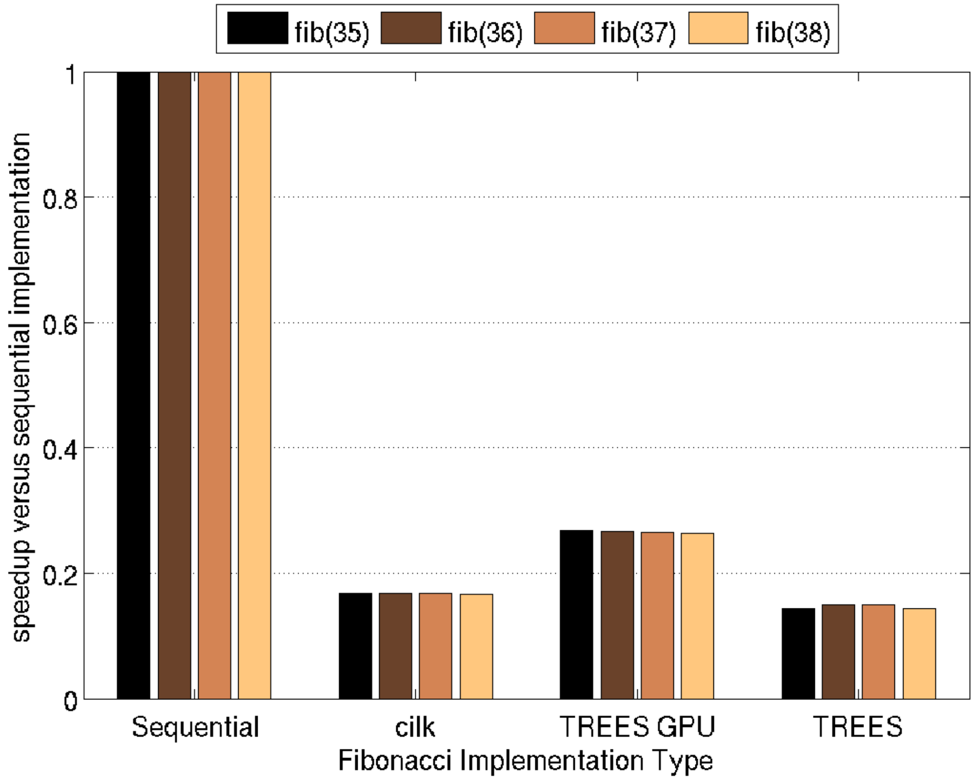


Figure 5.4: Performance of Fibonacci

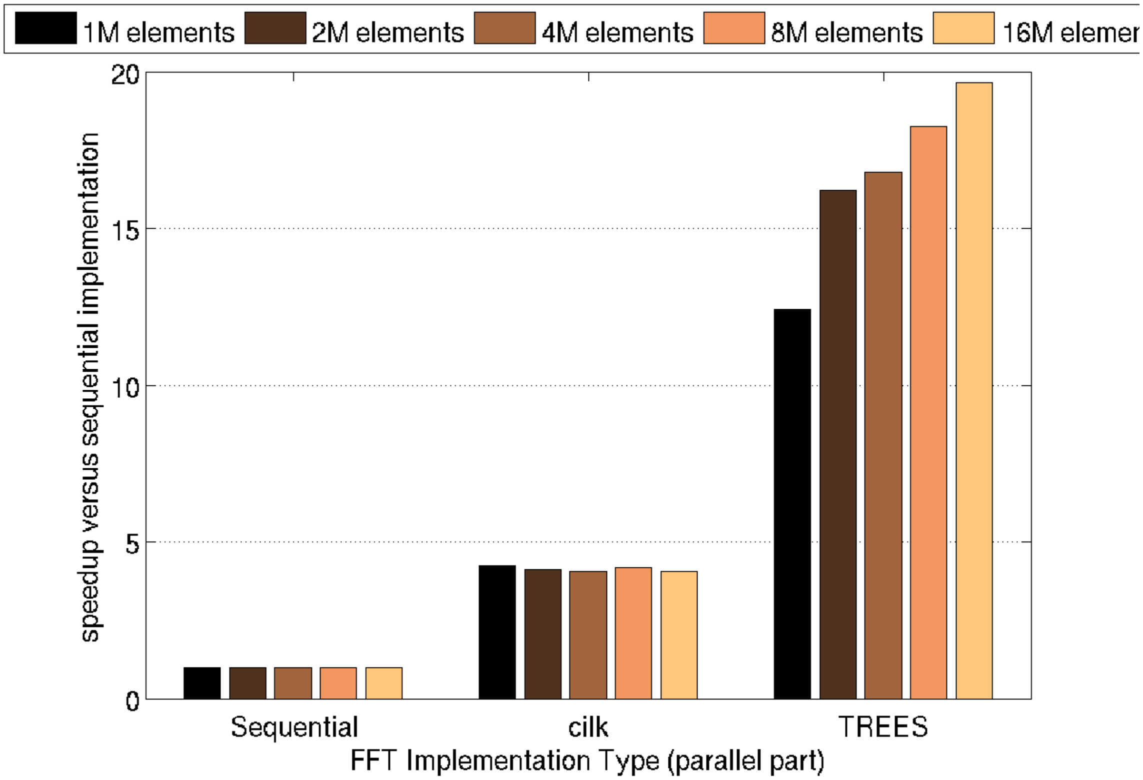


Figure 5.5: Performance of FFT Kernel

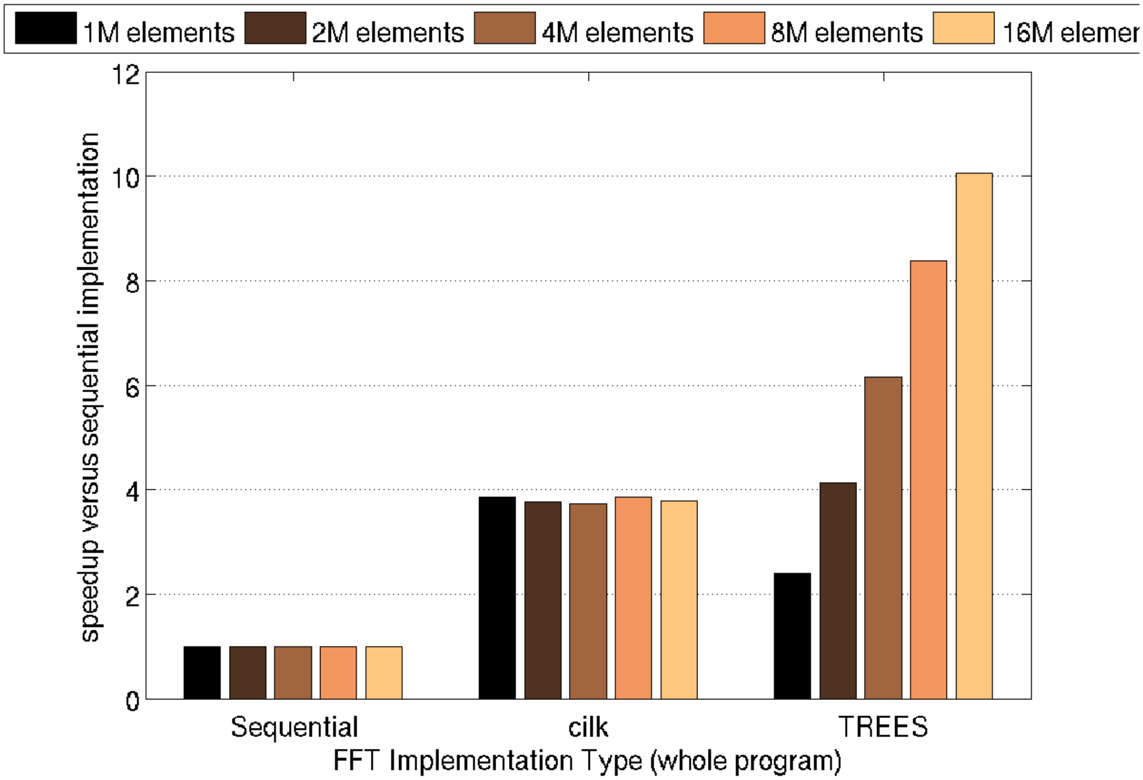


FIGURE 5.6: Performance of FFT Whole Program

5.6.4 Case Study 2: Comparison to work lists

Emerging work in GPU graph algorithms have begun to use work-lists to exploit the irregular parallelism in graph algorithms. In many ways the techniques used in this work are a subset of the TREES implementation. The Lonestar GPU benchmark suite uses work-lists to implement bfs (breadth-first search) and sssp (single-source shortest-path). These benchmarks use an input and output work-list to allow efficient push and pull operations. The pull operation is data parallel on the input work-list. Pushing to the output work-list uses a single tail pointer that is atomically incremented with new vertices to explore. After a kernel execution has completed, the host transfers a single int to see if a new relaxation kernel is necessary. If a new relaxation kernel is necessary, the input and output work-lists will be swapped and the launch bounds of the next kernel will be determined by the size of the old output worklist. This execution continues until no new nodes are explored during a relaxation kernel. Fundamentally, this is what occurs in the TREES runtime, with the exception of using separate work-lists. Unsurprisingly, TREES performs nearly the same as an OpenCL port of Lonestar's work-list based bfs and sssp. Both TREES and the work-list techniques perform worse than a topology driven execution.

Figures 5.7 and 5.8 show the performance of TREES versus an OpenCL port of the worklist versions of bfs and sssp form the Lonestar GPU benchmark suite. We can see that TREES is never more than 6% slower than the Lonestar equivalent benchmark. The performance difference likely comes from the extra load to determine the task type of the bfs and sssp. Overall, these results show that we are paying minimal cost for the generality of TREES. In these experiments we consider only the portion of the program executing on the GPU to show the worst case for TREES.

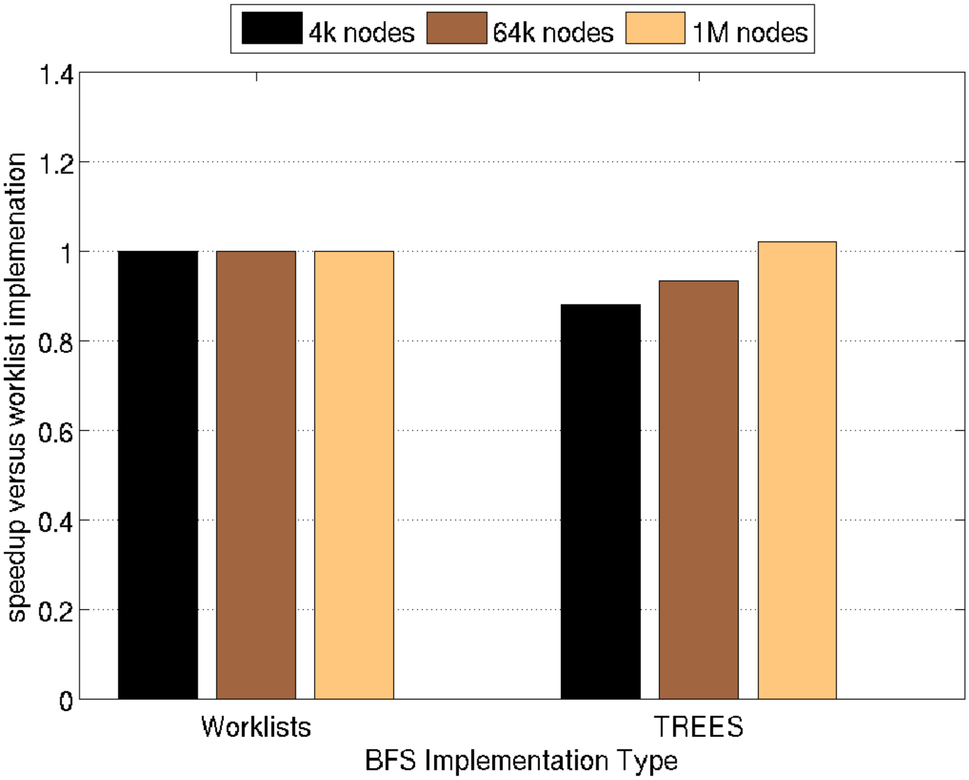


Figure 5.7: Performance of BFS

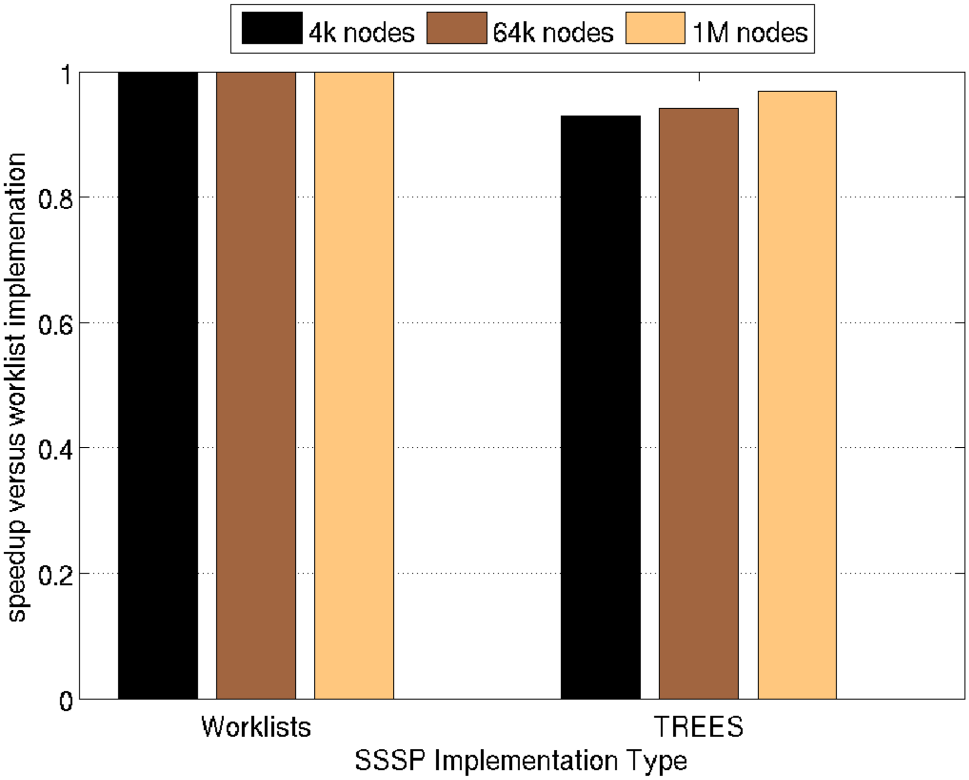


FIGURE 5.8: Performance of SSSP

5.6.5 Case Study 3: Optimization with mappers

In this section, we study the use of data parallel map operations to optimize a task parallel merge sort. This case study does not represent the typical use of TREES since the parallelism available is highly regular. However, it does show that TREES still enables near-native OpenCL performance with programmer effort.

We note that the basic implementation of merge sort in TREES performs significantly worse than the data parallel native OpenCL bitonic sort. To overcome this difference we implemented an equivalent algorithm in TREES. The TREES implementation has double the kernel launches, additional memory copies, and reads arguments from global memory instead of parameter space. After all of these overheads, the performance of TREES is only half of that of the native OpenCL kernel.

From this analysis, it is likely that a worst case performance loss between a native data parallel program and TREES would be 2-3x.

Figure 5.9 confirms that the performance of a Naive implementation of merge sort on TREES will perform vastly worst than an native OpenCL implementation using data parallel kernels. We show that we can easily bridge this gap using data parallelism in TREES. In both the naive version and the data parallel extension, we show consistent performance across many list sizes. Though is disappointing that there is a performance hit for TREES in this case. The upshot is that the overheads of TREES are scalable and independent of the problem size

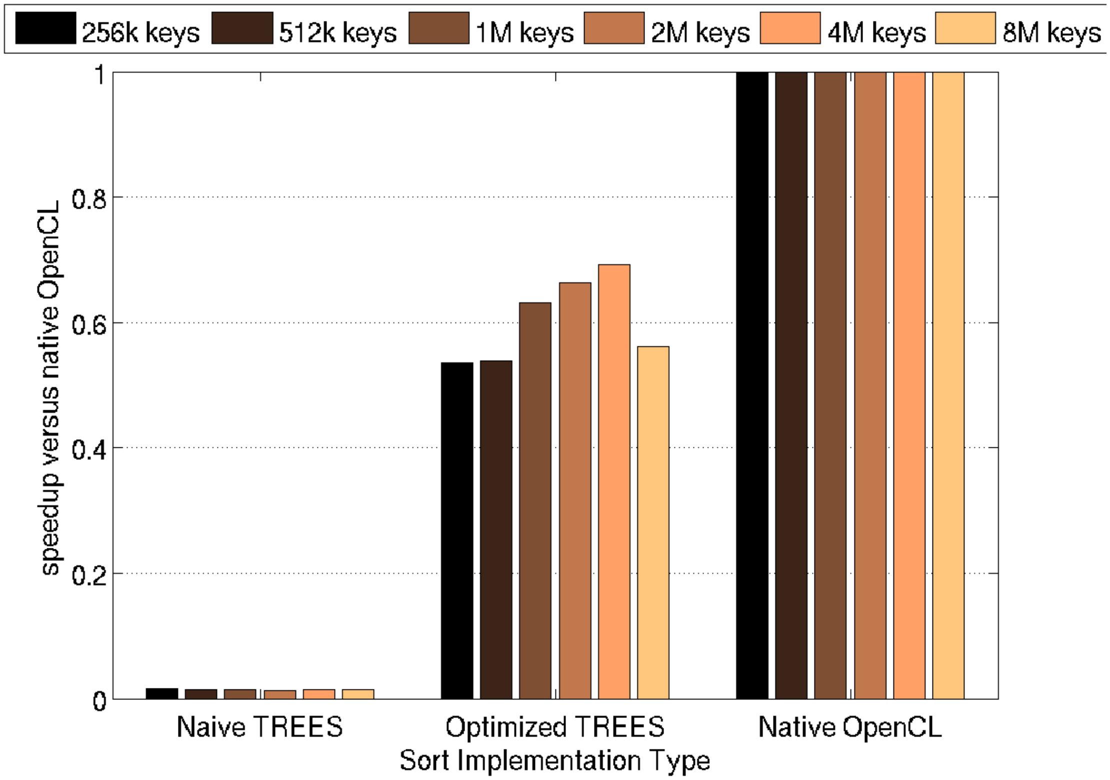


FIGURE 5.9: Performance of Sort

5.7 Related Work

There have been two avenues of prior work in providing some aspects of our desired programming model for GPUs. First, both CUDA 5.0 and the recently announced OpenCL 2.0 support "dynamic parallelism," in which a GPU kernel can directly launch a new kernel [44, 57]. Dynamic parallelism thus facilitates task-parallel programming, but with three significant drawbacks. First, there is a tradeoff between programmability and performance. If one writes a task with a single thread, which is how CPU programmers write task-parallel software, the kernel will have a single thread and will suffer from poor performance on the GPU. The second drawback to dynamic parallelism is potential deadlock: if a parent task is waiting for a child task to complete and the parent task suffers any branch divergence, then deadlock can occur [27]. The third drawback is hardware cost and portability: dynamic parallelism requires a modification of the GPU hardware, and the vast majority of current GPUs do not support dynamic parallelism.

The other avenue of prior work in providing task parallelism on GPUs is based on persistent threads [26]. A persistent thread, unlike a typical GPU thread, loops instead of running to completion. Although the original paper [26] did not specifically propose using persistent threads to support task-parallel programming, subsequent work has done just this [15, 58]. That is, in each loop iteration, each persistent thread finds a task to execute, executes the task, and then optionally adds one or more new tasks to a queue. Like dynamic parallelism, this prior work in using persistent threads for task parallelism achieves poor performance on the single-threaded tasks used in CPU programming models, due to an inability to exploit the GPU hardware. The performance of persistent threads also suffers when persistent threads are idle yet contending for resources (e.g., the global memory holding the task queue). Persistent threads also suffers from being extremely difficult to debug, because GPUs require a kernel to complete before providing any access to debugging information (e.g., host or GPU printf). Furthermore, not all hardware supports interrupting execution, in which case a buggy persistent threads program can require a hard reboot.

The StarPU runtime [7] supports task parallelism on heterogeneous systems, including CPU/GPU systems, but it has a somewhat different goal than TREES or the previously discussed related work. StarPU seeks to provide a uniform and portable high-level runtime that schedules tasks (with a focus on numerical kernels) on the most appropriate hardware resources. StarPU offers a much higher level interface to a system, with the corresponding advantages and disadvantages.

The OmpSs programming model [46] extends OpenMP to support heterogeneous tasks executing on heterogeneous hardware. OmpSs, like most prior work, would only achieve good performance on GPUs if the tasks themselves are data-parallel.

Inspiring all prior work on task parallelism for GPUs is the large body of work on task parallelism for CPUs. Cilk [47] and X10 [14] are two notable examples in this space, and both prior work and TREES borrow many ideas and goals from this work. The TVM, in particular, strives to provide a programming model that is as close to the CPU task programming model as possible.

Other researchers have explored the viability of running non-data-parallel, irregular algorithms on GPUs [13, 43]. This work has shown that GPUs can potentially achieve good performance on irregular workloads. Interestingly, Nasre et al. [42] have looked into irregular parallel algorithms from a data-driven rather than topology-driven approach, and this approach uses a task queue to manage work. TREES could complement this work by providing portable support for this task queue.

5.8 Conclusion

TREES provides a new use case for emerging heterogeneous systems, where an integrated GPU can be used to greatly improve the performance of irregular parallel workloads. Further, the TVM proposes a new way to easily reason about the performance of these irregular workloads. As the TVM effectively exposes task parallelism to the programmer and TREES implements this task parallelism efficiently, this work helps break the data parallel bottleneck that limits the use of GPUs. TREES embraces increased coupling between heterogeneous computing to offload dependency scheduling, memory consistency, and load-balancing to the CPU and built-in hardware mechanisms. Thus, we enable greater GPU occupancy by avoiding those challenges in software running on the GPU. We believe that such techniques can be used for other specialized but programmable hardware.

6

Conclusions

The conclusion of this thesis will summarize the tangible results, the intangible lessons learned, and future directions suggested. This format is useful because it can inspire future researchers to look at parallelism in a different light without the requirement for absolute correctness.

6.1 Summary

In this thesis, we showed it is often possible to rethink throughput-oriented architectures from the programmers prospective to achieve high performance and easy programmability. First, we showed that write-buffers were essentially useless on a future cache coherent massively-threaded accelerator. This observation enables strong consistency models with little performance overhead. Second, we showed that it is possible to design a GPU memory system that seamlessly supports future applications with fine-grain synchronization and current workloads with coarse-grain synchronization, which enables programmers to over-synchronize code without a loss in performance. Finally, we showed it is possible for the programmer to reason about fine-grain synchronization operations, but present bulk-synchronization operations to the GPU hardware, which enables highly efficient performance on task parallelism on a GPU.

It is useful to understand that the underlying tenets of all of these areas of research have the same foundation in thinking about problems across levels of the computation stack from the programmers perspective. When considering multiple layers of the stack concurrently, it is more clear what to optimize and why. When considering the many cores in a GPU, memory-level parallelism is most importantly achieved with thread-level parallelism rather than exposing instruction-level parallelism. When designing a write-through memory system with fences, assume the programmer needs to be overly careful. As a result, it is possible to design a high-throughput system whose performance is not killed by synchronization. However, after thinking about ways to implement synchronization, it is often better to rethink what the programmer wants from a synchronization operation. In general the programmer is attempting to schedule dependent events. When designing synchronization, we give the programmer what he wants while providing a sensible interface to hardware.

6.2 Lessons Learned

Through the research process many lessons are learned and this is a non-exhaustive list of what future researchers in GPUs and memory consistency should know. As an architect, the first goal in research was to think about how to make the hardware better. However, if one ignores the software, it is very easy just to optimize something entirely useless. As a result, it is necessary to truly understand Amdahl's law when thinking about parallelism. The obvious problem then becomes understanding the common case. The common case is good parallel code where corner cases in coherence and consistency do not occur. In general systems will naturally execute in sequentially consistent order due to the lack of data races. However, significant effort is spent on the performance and correctness of corner cases where non-sequentially consistent executions are possible.

To this end, it is best to think about consistency and coherence from the perspective of what it provides rather than how complex it is or how much it costs. For example, cache coherence and shared memory enable the use of work-stealing schedulers. Such schedulers efficiently balance the load of a task parallelism. Further, coherence is useful because the performance to access private data is as fast as a cache access, while fine-grain sharing only slightly degrades that performance.

Memory consistency is important when scheduling dependent operations. However, this is a very difficult problem on a throughput architecture since the system needs to both write a result and inform the scheduling system the write is done. By definition, this requires strong consistency or a fence operation. In a throughput-oriented system, strong consistency and fence operations will limit throughput. Such scheduling systems will also require busy-waiting for dependencies to complete. Throughput architectures should not busily wait for dependent operations to complete to enable the maximal throughput.

Never think that a GPU has enough threads and memory to solve an NP-complete problem in parallel. The brute force answer is still way too complex for thousands of threads to make a dent. This is more obvious when thinking about the number of cores in a system. Often CPU based systems have dozens of cores or $log(n)$, while GPU system may have at best thousands of cores of $log^2(n)$. When considering a problem that is 2^n, a logarithmic factor does not help much. Clearly, randomized algorithms are the only way forward, but generating random numbers in parallel is either expensive or less random.

Finally, don't be afraid to go against conventional wisdom or research inertia. Sometimes the old problem is old because it has been solved.

6.3 Future Directions

Hardware Support for synchronization:

GPUs could likely benefit from hardware support for synchronization operations. Building support in hardware for TREES-like fork and join operations could significantly improve the performance. Further, if hardware is aware of locks and condition variables writing correct irregular code could become much easier.

TVM on FPGA/ASIC:

The TVM is an interesting formal abstract machine and is a good candidate for direct implementation. The cost of fork, join, and communication with the host can be almost entirely factored out of the system. The hardware implementation would not reduce the space complexity of the TVM and TREES, but we could imagine storing the Task Vector in a large low power memory because its access pattern is highly predictable.

System calls on TREES:

The design of TREES enables multiple types of specialized kernels to interact as well as resolve any dependencies in bulk. This type of design could work for IO operations and other various system calls. This use of TREES would enable new levels of programmability without requiring hardware support. However, it would be possible to use hardware support to gracefully improve performance.

Apply work-first/work-together to out-of-order core design:

Instruction-level parallelism is not all that different from task parallelism at a fundamental level. One way to think about an out-of-order core currently, is that out-of-order cores obey the work-first principle. Out-of-order cores create a longer pipeline (i.e. critical path) in order to keep the pipeline as full as possible (i.e. work). In fact, it is possible to think of Multiscalar [53] as the work-together principle applied to out-of-order core design.

Bibliography

[1] S. V. Adve and K. Gharachorloo, "Shared memory consistency models: A tutorial," *IEEE Computer*, vol. 29, no. 12, pp. 66–76, Dec. 1996.

[2] S. V. Adve and M. D. Hill, "Weak ordering: A new definition," in *Proceedings of the 17th Annual International Symposium on Computer Architecture*, 1990, pp. 2–14.

[3] S. V. Adve and M. D. Hill, "A unified formalization of four shared-memory models," *IEEE Transactions on Parallel and Distributed Systems*, Jun. 1993.

[4] AMD, *Southern Islands Series Instruction Set Architecture*, 2012.

[5] AMD, *Accelerated Parallel Processing (APP) SDK*, 2013.

[6] Arvind and J.-W. Maessen, "Memory model = instruction reordering + store atomicity," in *Proceedings of the 33rd Annual International Symposium on Computer Architecture*, Jun. 2006, pp. 29–40.

[7] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "StarPU: A unified platform for task scheduling on heterogeneous multicore architectures," *Concurrency and Computation: Practice and Experience, Special Issue: Euro-Par 2009*, pp. 187–198, 2011.

[8] 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 Comput. Archit. News*, p. 17, Aug. 2011.

[9] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An efficient multithreaded runtime system," *Journal of parallel and distributed computing*, vol. 37, no. 1, 1996.

[10] H.-J. Boehm and S. V. Adve, "Foundations of the c++ concurrency memory model," in *Proceedings of the Conference on Programming Language Design and Implementation*, Jun. 2008.

[11] A. Branover, D. Foley, and M. Steinman, “AMD’s llano fusion APU,” *IEEE Micro*, 2011.

[12] R. P. Brent, “The parallel evaluation of general arithmetic expressions,” *Journal of the ACM (JACM)*, vol. 21, no. 2, 1974.

[13] M. Burtscher, R. Nasre, and K. Pingali, “A quantitative study of irregular programs on GPUs,” in *IEEE International Symposium on Workload Characterization*, 2012, pp. 141–151.

[14] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, “X10: An object-oriented approach to non-uniform cluster computing,” in *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, 2005, pp. 519–538.

[15] S. Chatterjee, M. Grossman, A. Sbîrlea, and V. Sarkar, “Dynamic task parallelism with a GPU work-stealing runtime system,” in *Languages and Compilers for Parallel Computing*, ser. Lecture Notes in Computer Science, S. Rajopadhye and M. Mills Strout, Eds. Springer Berlin Heidelberg, 2013, vol. 7146, pp. 203–217.

[16] S. Che, M. Boyer, J. Meng, D. Tarjan, J. Sheaffer, S.-H. Lee, and K. Skadron, “Rodinia: A benchmark suite for heterogeneous computing,” in *Proceedings of the IEEE International Symposium on Workload Characterization*, 2009, pp. 44–54.

[17] Compaq, *Alpha 21264 Microprocessor Hardware Reference Manual*, Jul. 1999.

[18] P. Conway and B. Hughes, “The AMD opteron northbridge architecture,” *IEEE Micro*, vol. 27, no. 2, pp. 10–21, Apr. 2007.

[19] P. Conway, N. Kalyanasundharam, G. Donley, K. Lepak, and B. Hughes, “Cache hierarchy and memory subsystem of the AMD opteron processor,” *IEEE Micro*, vol. 30, no. 2, pp. 16–29, Apr. 2010.

[20] P. Du, R. Weber, P. Luszczek, S. Tomov, G. Peterson, and J. Dongarra, “From CUDA to OpenCL: towards a performance-portable solution for multi-platform GPU programming,” *Parallel Computing*, 2011.

[21] W.-c. Feng, H. Lin, T. Scogland, and J. Zhang, “Opencl and the 13 dwarfs: a work in progress,” in *Proceedings of the third joint WOSP/SIPEW international conference on Performance Engineering.* ACM, 2012.

[22] M. Frigo, C. E. Leiserson, and K. H. Randall, “The implementation of the Cilk-5 multithreaded language,” *ACM Sigplan Notices*, vol. 33, no. 5, 1998.

[23] B. Gaster and L. Howes, “Can gpgpu programming be liberated from the data-parallel bottleneck?” *Computer*, vol. 45, no. 8, pp. 42–52, August 2012.

[24] J. Goodacre and A. N. Sloss, “Parallelism and the ARM instruction set architecture,” *IEEE Computer*, vol. 38, no. 7, pp. 42–50, Jul. 2005.

[25] R. L. Graham, “Bounds on multiprocessing timing anomalies,” *SIAM Journal on Applied Mathematics*, vol. 17, no. 2, 1969.

[26] K. Gupta, J. Stuart, and J. Owens, “A study of persitent threads style GPU programming for GPGPU workloads,” in *Proceedings of InPar*, May 2012.

[27] M. Harris, “Many-core GPU computing with NVIDIA CUDA,” in *Proceedings of the 22nd Annual International Conference on Supercomputing*, 2008, p. 11.

[28] B. A. Hechtman and D. J. Sorin, “Exploring memory consistency for massively-threaded throughput-oriented processors,” in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, 2013.

[29] M. D. Hill, “Multiprocessors should support simple memory consistency models,” *IEEE Computer*, vol. 31, no. 8, pp. 28–34, Aug. 1998.

[30] D. R. Hower, Hechtman, Blake A., Beckmann, Bradford M., Gaster, Benedict R., Hill, Mark D., Reinhardt, Steven K., and Wood, David A., “Heterogeneous-race-free memory models,” in *ASPLOS ’14*, 2014.

[31] HSA Foundation, “Deeper look into HSAIL and it’s runtime,” Jul. 2012.

[32] HSA Foundation, “Standards,” 2013.

[33] Intel, *A Formal Specification of Intel Itanium Processor Family Memory Ordering*, 2002.

[34] L. Lamport, “How to make a multiprocessor computer that correctly executes multiprocess programs,” *IEEE Transactions on Computers*, pp. 690–691, Sep. 1979.

[35] J. Leng, T. Hetherington, A. ElTantawy, S. Gilani, N. S. Kim, T. M. Aamodt, and V. J. Reddi, “GPUWattch: enabling energy optimizations in GPGPUs,” in *proc. of ISCA*, vol. 40, 2013.

[36] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy, *The Directory-based Cache Coherence Protocol for the DASH Multiprocessor.* ACM, 1990, vol. 18, no. 3a.

[37] D. Lenoski, J. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. Lam, “The stanford DASH multiprocessor,” *IEEE Computer*, vol. 25, no. 3, pp. 63–79, Mar. 1992.

[38] P. A. MacMahon, “Combinatory Analysis,” *Press, Cambridge*, 1915.

[39] J. Manson, W. Pugh, and S. V. Adve, “The java memory model,” in *Proceedings of the 32nd Symposium on Principles of Programming Languages*, Jan. 2005.

[40] A. McDonald, J. Chung, H. Chafi, C. C. Minh, B. D. Carlstrom, L. Hammond, C. Kozyrakis, and K. Olukotun, “Characterization of TCC on chip-multiprocessors,” in *Parallel Architectures and Compilation Techniques, 2005. PACT 2005. 14th International Conference on*, 2005, p. 6374.

[41] A. Munshi, “OpenCL,” *Parallel Computing on the GPU and CPU, SIGGRAPH*, 2008.

[42] R. Nasre, M. Burtscher, and K. Pingali, “Data-Driven Versus Topology-driven Irregular Computations on GPUs,” in *IEEE 27th International Symposium on Parallel and Distributed Processing (IPDPS)*, 2013.

[43] R. Nasre, M. Burtscher, and K. Pingali, “Morph Algorithms on GPUs,” in *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2013.

[44] NVIDIA, “NVIDIA’s next generation CUDA compute architecture: Kepler GK110.”

[45] S. Owens, S. Sarkar, and P. Sewell, “A Better x86 Memory Model: x86-TSO,” ser. TPHOLs ’09. Berlin, Heidelberg: Springer-Verlag, 2009, p. 391407.

[46] J. Planas, L. Martinell, X. Martorell, J. Labarta, R. M. Badia, E. Ayguade, and A. Duran, “OmpSs: A proposal for programming heterogeneous multi-core architectures,” *Parallel Processing Letters*, vol. 21, no. 02, pp. 173–193, 2011.

[47] K. H. Randall, “Cilk: Efficient multithreaded computing,” Ph.D. dissertation, Massachusetts Institute of Technology, 1998.

[48] M. Raynal and A. Schiper, “From causal consistency to sequential consistency in shared memory systems,” 1995, pp. 180–194.

[49] A. Ros and S. Kaxiras, “Complexity-effective multicore coherence,” in *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, 2012.

[50] T. Sha, M. M. K. Martin, and A. Roth, “Scalable store-load forwarding via store queue index prediction,” in *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture*, Nov. 2005, pp. 159–170.

[51] I. Singh, A. Shriraram, W. W. L. Fung, M. O’Connor, and T. M. Aamodt, “Cache coherence for GPU architectures,” in *Proceedings of the 19th IEEE International Symposium on High-Performance Computer Architecture*, Feb. 2013, pp. 578–590.

[52] R. L. Sites, Ed., *Alpha Architecture Reference Manual.* Digital Press, 1992.

[53] G. Sohi, S. Breach, and T. Vijaykumar, “Multiscalar processors,” in *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, Jun. 1995, pp. 414–425.

[54] D. J. Sorin, M. D. Hill, and D. A. Wood, “A primer on memory consistency and cache coherence,” *Synthesis Lectures on Computer Architecture*, vol. 6, no. 3, pp. 1–212, May 2011.

[55] G. J. Sussman and G. L. Steele Jr., “Scheme: An interpreter for extended lambda calculus,” in *MEMO 349, MIT AI LAB*, 1975.

[56] A. A. Thomas F. Wenisch and A. Moshovos, “Mechanisms for store-wait-free multiprocessors,” in *Proceedings of the 34th Annual International Symposium on Computer Architecture*, Jun. 2007.

[57] R. Tsuchiyama, T. Nakamura, T. Iizuka, A. Asahara, S. Miki, and S. Tagawa, “The opencl programming book,” *Fixstars Corporation*, vol. 63, 2010.

[58] S. Tzeng, B. Lloyd, and J. Owens, “A GPU Task-Parallel Model with Dependency Resolution,” *IEEE Computer*, vol. 45, no. 8, 2012.

[59] D. L. Weaver and T. Germond, Eds., *SPARC Architecture Manual (Version 9).* PTR Prentice Hall, 1994.

Biography

Blake Alan Hechtman was born on October 6, 1988 in Miami, FL. He has earned a BS, MS, and PhD at Duke University in the Department of Electrical and Computer Engineering in 2010, 2012, and 2014 respectively. He first published work in logical time cache coherence in WDDD 2012. He then published a poster at ISPASS 2013 on cache-coherent shared virtual memory for chips with heterogeneous cores. The follow-on to that work was published in ISCA 2013 and studies the memory consistency for massively-threaded throughput-oriented processors. While at AMD, Blake worked on the theory and implementation (QuickRelease) of heterogeneous-race-free memory models which were published in ASPLOS 2014 and HPCA 2014 respectively.