COMMUNICATION AND COORDINATION PARADIGMS FOR HIGHLY-PARALLEL ACCELERATORS

by

Marc S. Orr

A dissertation submitted in partial fulfillment of

the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN—MADISON

2016

Date of final oral examination: December 5th, 2016

The dissertation is approved by the following members of the Final Oral Committee:

Bradford M. Beckmann, PhD, Advanced Micro Devices, Inc. (AMD)
Mark D. Hill, Professor, Computer Sciences
Nam Sung Kim, Associate Professor, Electrical Engineering (UIUC)
Karthikeyan Sankaralingam, Associate Professor, Computer Sciences
Michael M. Swift, Associate Professor, Computer Sciences
David A. Wood (Advisor), Professor, Computer Sciences

For my wife, Lily.

For our daughters, Melanie and Michaela.

ACKNOWLEDGMENTS

First, I want to thank my wife and best friend, Lily. Thanks for challenging me by being the biggest dreamer that I know. Thanks for celebrating with me during our highest highs and supporting me during our lowest lows. Finally, thanks for being a single working mom when you needed to be.

I owe a big thanks to my older daughter, Melanie, who suffered being the child of a grad student for the first five years of her life! Thanks for letting me leave you to work when I needed to. Thanks for playing with me when I wanted to forget about work. Thanks for motivating me to focus and succeed. Finally, thanks for your unconditional love and always believing in your dad.

I also want to thank my younger daughter, Michaela, who only suffered being the child of a grad student for the first year of her life. Thanks for reminding me what's important in life. Thanks for showing me when to yell and when to listen. Finally, thanks for motivating me to finish.

I owe a big thanks to my advisor, David Wood. Thanks for advising me through thick and thin. Thanks for bringing out the best in me by having such high expectations. Thanks for encouraging me to collaborate with others. Finally, thanks for teaching me the value of articulating my thoughts, focusing on high-level insights, and giving good presentations.

I also owe a big thanks to Brad Beckmann for mentoring me the last five years. Thanks for inviting me to work on your team at AMD, where I got to contribute to exciting and impactful projects! Thanks for teaching me how to do research. Finally, thanks for being optimistic and encouraging when my research got stalled or my papers got rejected.

Thanks to the rest of my committee—Mark Hill, Nam Sung Kim, Karu Sankaralingam, and Mike Swift. I got to collaborate with Mark Hill for a bit and during that time I came to appreciate his talent to distinguish meaningful insights from frivolous details. I also found Mark's positive outlook on people and research refreshing. Thanks to Nam for participating on my committee,

even after moving to another university. Thanks to Karu for joining my committee in the last year so that I could meet the university's requirements. Finally, thanks to Mike for reviewing my dissertation so carefully.

I want to thank Steve Reinhardt, Mark Oskin, John Alsop, Shuai Che, Ayse Yilmazer, Jason Lowe-Power, and Joel Hestness for various collaborations. Thanks to Steve for encouraging me to explore GPU-wide aggregation, which became the foundation of my dissertation. Thanks to Mark Oskin for steering me out of oblivion when I got lost working on Gravel. Thanks to John for continuing my work on remote scope promotion. Thanks to Shuai for helping me work through many low-level research issues. Thanks to Ayse for developing the cache model for remote scope promotion. Thanks to Jason and Joel for continuing to work on gem5-gpu when I switched to using AMD's simulation infrastructure.

I want to thank Jason Lowe-Power for being a close friend and confidant. Over the years, I've enjoyed our spirited discussions, which often took place over coffee or beer (sometimes Jason's delicious homebrew!). I also want to thank Tiffany Lowe-Power for her friendship. Specifically, I'm grateful for the time that Jason and Tiffany spent with my family (including taking Melanie to the Madison zoo!). I'm also grateful for the countless number of times that they helped me out.

I have several miscellaneous thanks to make. First, I want to thank Alaa Alameldeen for sparking my interest in computer architecture, which led me to Wisconsin. I want to thank Matt Sinclair for encouraging me to apply for a summer research assistantship at the end of my first year. I want to thank Lena Olson, Hongil Yoon, and Jason Lowe-Power for the summer we spent together preparing for the daunting qualifier exam. Reviewing all of those papers should've been miserable, but somehow I remember it fondly. I want to thank Tony Nowatzki for his comradery when we were teaching assistants for the same course and later when we were students in the same

atrocious summer course that shall remain unnamed. I want to thank Jacob Nelson for imparting his hard-earned Grappa wisdom to me. Coincidentally, it was only after speaking with Jacob that I started to make progress on Gravel. Finally, I want to thank several folks for being helpful over the years, including Newsha Ardalani, Shoaib Atlaf, Arka Basu, Brad Benton, Yasuko Eckert, Jayneel Gandhi, Ben Gaster, Nick George, Joe Greathouse, Gagan Gupta, Tony Gutierrez, Derek Hower, Lee Howes, Gabe Loh, Sooraj Puthoor, Somayeh Sardashti, Mike Schulte, Rathijit Sen, and Nilay Vaish.

I want to thank my family and Lily's family. Thanks to my parents, Les and Debbie Orr, and my brother, David Orr, for helping to shape me into who I am. Thanks to Lily's parents, Chau Huynh and Hue Nguyen, for treating me like their own son. Thanks to Lily's brother and sisters—Chi Huynh, Coty Huynh, Sici Huynh, and Noda Huynh—for treating me like their own brother. Thanks to all of my nephews and nieces—Be Lily, Nova, Preston, Keagan, Connor, and Amelia—for many wonderful get-togethers over the years.

Finally, I want to thank the university and AMD for funding me while I was a student. I also want to thank Angela Thorp for helping me to work through the grad school's arcane rules.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

As CPU performance plateaus, many communities are turning to highly-parallel accelerators such as graphics processing units (GPUs) to obtain their desired level of processing power. Unfortunately, the GPU's massive parallelism and data-parallel execution model make it difficult to synchronize GPU threads. To resolve this, we introduce *aggregation buffers*, which are producer/consumer queues that act as an interface from the GPU to a system-level resource. To amortize the high cost of producer/consumer synchronization, we introduce *leader-level synchronization*, where a GPU thread is elected to synchronize on behalf of its data-parallel cohort.

One challenge is to coordinate threads in the same data-parallel cohort accessing different aggregation buffers. We explore two schemes to resolve this. In the first, called *SIMT-direct aggregation*, a data-parallel cohort invokes leader-level synchronization once for each aggregation buffer being accessed. In the second, called *indirect aggregation*, a data-parallel cohort uses leader-level synchronization to export its operations to a hardware aggregator, which repacks the operations into their respective aggregation buffers.

We investigate two use cases for aggregation buffers. The first is the channel abstraction, which was proposed by Gaster and Howes to dynamically aggregate asynchronously produced fine-grain work into coarser-grain tasks. However, no practical implementation has been proposed. We investigate implementing channels as aggregation buffers managed by SIMD-direct aggregation. We then present a case study that maps the fine-grain, recursive task spawning in the Cilk programming language to channels by representing it as a flow graph. We implement four Cilk benchmarks and show that Cilk can scale with the GPU architecture, achieving speedups of as much as 4.3x on eight GPU cores.

The second use case for aggregation buffers is to enable PGAS-style communication between threads executing on different GPUs. To explore this, we wrote a software runtime called Gravel, which incorporates aggregation buffers managed by indirect aggregation. Using Gravel, we distribute six applications, each with frequent small messages, across a cluster of eight AMD accelerated processing units (APUs) connected by InfiniBand. Compared to one node, these applications run 5.3x faster, on average. Furthermore, we show that Gravel is more programmable and usually more performant than prior GPU networking models.

1. INTRODUCTION

For over four decades, the number of transistors on a chip (forecasted by Moore's law [1]) and the frequency that those transistors were clocked (forecasted by Dennard scaling [2]) increased exponentially. Specifically, Dennard scaling forecasted that a chip's power density remains constant as MOSFET feature sizes and switching delay become smaller. 2005 brought the end of Dennard scaling as we know it. Dennard scaling stopped because leakage currents, which had been negligible when transistors were larger, made up a non-trivial fraction of the power consumption [3]. Today, at a given power density, smaller transistors are only slightly faster than transistors at the previous feature size because of leakage currents. Frequency scaling ended at the same time that micro-architectural innovations, which also played an important role in improving single-thread performance into the 2000s, began to provide diminishing returns [4].

These dual crises have pushed parallelism to the forefront. Initially, this parallelism was realized with multicore processors. Later, architects embraced parallelism with more zeal by building highly-parallel accelerators, like GPUs and the many integrated cores (MIC) architecture. For instance, consider the Green500 list, which tracks the most energy-efficient supercomputers—all of the top ten systems incorporate highly-parallel accelerators [5]. At the commodity end, cloud infrastructures like Amazon's elastic compute cloud (EC2) provide GPU-compute resources [6] and these GPUs are being used to accelerate applications in a number of domains, ranging from high-performance computing [7] to machine learning [8][9][10][11].

Two attributes typically characterize highly-parallel accelerators. First, they prefer a large number of simple, low-performance threads over a small number of complex, high-performance threads. Thus, given more transistors, a highly-parallel accelerator can be built with more hardware

threads. Importantly, this approach does not require increasing the accelerator's clock frequency or designing new, complex, and power-hungry speculation logic.

The second attribute that characterizes *most* highly-parallel accelerators is a significant number of data-parallel execution units, which apply a single instruction stream to multiple threads simultaneously. Data-parallel hardware amortizes the power consumed by the accelerator's front-end (i.e., the pipeline's fetch and decode stages), but limits the parallel codes that can fully leverage a highly-parallel accelerator.

1.1 GPUs: An Important Sub-class of Highly-parallel Accelerators

In this thesis, we focus on GPUs, which are an important sub-class of highly-parallel accelerators. For example, earlier we cited the fact that all of the systems on the Green500 list incorporate highly-parallel accelerators. Of note, nine of these systems use GPUs! Unlike other highly-parallel accelerators, GPUs are manufactured by multiple companies and are in high-demand in non-compute markets (i.e., graphics). These market factors make GPUs a more cost-effective solution than other highly-parallel accelerators.

The second reason that we focus on GPUs is because their high thread count and abundant data-parallel hardware epitomize the highly-parallel architecture. For example, consider NVIDIA's Fermi architecture—a GPU architecture that comprises thousands of simple threads and data-parallel hardware to execute them. Notably, Fermi's design is eight times more energy efficient than Intel's Westmere architecture (a comparable CPU) [12].

While it is true that compared to CPUs, GPUs have the potential to execute parallel applications faster and more efficiently, they are also more difficult to program. In addition to well-known parallel programming challenges (e.g., dynamic load balancing, concurrency bugs, non-determinism), their highly-parallel architecture exposes several new challenges. First, their

high thread count can cause increased cache and register pressure, severe thread contention (e.g., when all threads contend for the same lock), and high memory bandwidth demand. Second, data-parallel hardware can introduce performance and correctness issues. Specifically, data-parallel hardware becomes underutilized when *adjacent threads*, which are threads executing the same instruction stream, have different control flow. Furthermore, in GPUs, memory operations become more expensive when adjacent threads access non-adjacent adjacent memory locations.

In recent years, the research community has made progress in identifying and improving many of the difficulties in programming GPUs. Specifically, recent work has explored how to correctly use general purpose synchronization primitives on GPUs [13][14]. At the same time, researchers have been exploring how to mitigate the utilization [15][16] and memory system [17] issues that plague GPUs. In this thesis, we focus on a third programming issue that has received less attention up to now—*system-level operations*.

1.2 Defining and Understanding System-level Operations

Fundamentally, a system-level operation is an operation that requires system-level coordination. Here are some concrete examples:

- **network access:** threads access the network through a centralized network interface,
- **dynamic memory allocation:** threads allocate memory through a process-level heap,
- **system calls:** threads rely on the OS to execute a system call on their behalf.

Similarly, launching a new task on the current GPU can be viewed as a system-level operation because it involves coordinating through the GPU's centralized control processor (introduced in Chapter 2). System-level operations are commonplace in CPU applications. In contrast, GPUs have historically provided little to no support for system-level operations.

Executing a system-level operation on the GPU poses two challenges. The first is correctly and efficiently coordinating GPU thread access to the underlying resource. The GPU's data-parallel hardware can make it difficult to correctly synchronize. For example, introducing a dependency between two threads in the same data-parallel cohort can cause deadlock. At the same time, the GPU's massive parallelism can make it difficult to synchronize efficiently.

The second challenge is executing the operation itself. For example, the operation could be executed from the GPU directly. Prior work on accessing the network from the GPU-ported pieces of the InfiniBand library onto the GPU and found that the resulting code performed poorly [18].

An alternative approach is to execute a system-level operation outside of the GPU's data-parallel hardware. Specifically, the operation can be executed on a scalar core inside the host CPU, the network interface controller (NIC), or even on an embedded core inside of the GPU (e.g., the GPU's control processor, discussed in Chapter 2). In this dissertation, we choose to export operations to be executed outside of the GPU's data-parallel hardware because we are primarily concerned with system-level coordination.

1.3 Executing a System-level Operations from a GPU

The key challenge in executing a system-level operation is to route it from the GPU's data-parallel hardware to the target device (e.g., the host CPU, another GPU, the NIC). There are two aspects to this challenge. First, how can we provide application programming interfaces (APIs) that expose system-level operations to programmers and are easy to use? Second, how are operations routed from a GPU thread to their target device in an efficient manner?

Importantly, the APIs influence how operations are routed. In particular, some programming abstractions naturally correspond to coarse-grain GPU-to-device interactions, at GPU kernel granularity, while others correspond to fine-grain interactions, at GPU thread granularity. In

general, coarse-grain interactions are more performant because they amortize redundant overhead, but less flexible. In contrast, fine-grain interactions tend to be less performant but provide programmers greater flexibility.

Several approaches to routing system-level operations and exposing them to programmers have been proposed in prior work. Specifically, we consider three prior proposals that dominate the literature. In each case, we encounter difficulties in the programming model or, even worse, fundamental performance limitations. The first proposal, called the *coprocessor model* [19][20], does not provide GPU threads with the capability to execute a system-level operation directly. Instead, programmers write CPU code to handle system-level operations before and after offloading computation to a GPU. This model's poor programmability is partially offset by its bulk synchronous behavior, which encourages coarse-grain exchanges that amortize the overhead of routing a system-level operation to its target.

In the second proposal, called the *operation-per-lane model* [18][21][22][23], GPU threads execute system-level operations directly and independently. This model follows the CPU paradigm. Compared to the coprocessor model, this model simplifies programming, but it can lead to high-overhead producer/consumer interactions. Finally, in the third proposal, called *coalesced APIs* [24][25][26], GPU threads execute system-level operations directly after coordinating with their neighbors. Compared to the operation-per-lane model, this model is harder to program, but it does a better job of amortizing overheads. However, interactions are still small and high-overhead compared to the coprocessor model.

1.4 Efficiently Enabling Fine-grain System-level Operations for GPUs

The goal of this thesis is to provide the programmability of the operation-per-lane model at a level of performance that equals or exceeds the coprocessor model. There are two high-level ways to

approach this problem. The first is to build special hardware structures (or specialized coherence mechanisms) to route a system-level operation to its target device. For example, one might look to the hardware task queues proposed by Kim and Batten for inspiration [27]. The problem with this approach is that often GPU vendors have been reluctant to dedicate silicon to large non-graphics features (smaller features are less controversial).

The second approach is to leverage the memory system's built-in coherence mechanisms to route an operation to its target. Specifically, producer/consumer synchronization can be used to export a system-level operation to its target device. The problem with this approach is that the coherence mechanisms in current GPUs tend to incur a high performance penalty—especially for small kernels or kernels that frequently invoke producer/consumer synchronization.

Nonetheless, to increase the potential for impact, we focus on solutions that leverage the coherence mechanisms available in today's GPUs. Specifically, in this thesis, we propose a simple, but crucial mechanism called *leader-level synchronization*, which operates as follows. Given a group of data-parallel threads, a single leader thread is elected to invoke producer/consumer synchronization on behalf of its entire group. The key insight is that producer/consumer synchronization can be used to export operations from the GPU and that the high-cost of producer/consumer synchronization can be amortized across the GPU's data-parallel lanes. For example, in our channels prototype, discussed in Chapter 4, producer/consumer synchronization is amortized across a wavefront (up to 64 work-items on current AMD GPUs). Similarly, in our Gravel prototype, discussed in Chapter 5, synchronization is amortized across a work-group (up to 256 work-items on current AMD GPUs).

It is trivial to use leader-level synchronization to route system-level operations from a group of data-parallel threads to a *single* target (e.g., the host CPU). Notably, if that single target device

is the host CPU then the GPU can execute *any* operation (e.g., `printf`, `malloc`, reading/writing files, system calls) by "reverse offloading" it to the CPU!

Nonetheless, it is less clear how operations originating from the same group of data-parallel threads can be routed to multiple targets. For example, one may wish to distribute the operations across an array of processors comprising CPUs and GPUs. Furthermore, some of these processors may be connected through a network, which requires system-level operations to be routed through the NIC.

Even when every data-parallel thread executes the same system-level operation, it can make sense to route the operations to multiple targets. For example, in Chapter 4 we explore dynamically aggregating scalar tasks into data-parallel tasks. In this work, we aim to support one system-level operation—launching a new task—but we explore dynamically organizing those tasks into multiple task queues that correspond to the function being spawned. Similarly, in Chapter 5 we explore GPU-initiated network messages and find that it is desirable to dynamically organize messages into multiple message queues that correspond to the node where a message is being sent.

1.5 Contributions

Our first and most broadly applicable contribution is to show how system-level operations can be exposed to GPU programmers through the operation-per-lane model at a level of performance that approaches or even exceeds the coprocessor model. As discussed earlier (Section 1.3) and reinforced by the first section of Chapter 3, the operation-per-lane model provides the best programmability. This fact, coupled with the reality that a performant implementation of the operation-per-lane model has remained elusive, makes our first contribution significant.

Recall that leader-level synchronization, introduced in Section 1.4, is sufficient when system-level operations have a single target (e.g., the host CPU), but things become more complicated

when there are multiple targets. To solve this problem, we introduce *aggregation buffers*, which are producer/consumer queues that act as an interface from the GPU to a system-level resource. For example, Figure 1-1 shows how aggregation buffers might be used to route system-level operations to a CPU, a NIC, and a GPU. Aggregation buffers can be used to sort operations by other attributes as well (e.g., a network message's destination).

In Chapter 3, we propose two aggregation schemes to group system-level operations into aggregation buffers. Both strategies build off of leader-level synchronization. In the first scheme, called *SIMT-direct aggregation*, data-parallel groups route operations directly to their respective aggregation buffer. In the second scheme, called *indirect aggregation*, data-parallel groups offload operations to a dedicated aggregator, which repacks them into their respective aggregation buffer. We found that SIMT-direct aggregation is more efficient when there are a small number of aggregation buffers, but indirect aggregation is better at exploiting the GPU's underlying data-parallel hardware and is more scalable with respect to the number of aggregation buffers.

The second contribution of this thesis is to explore two novel use cases for fine-grain system-level operations, which are enabled by the operation-per-lane model. The following subsections summarize these two efforts.

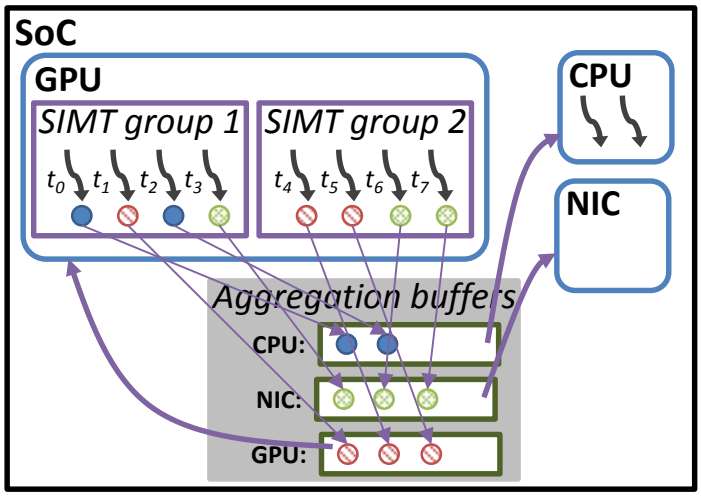


Figure 1-1. Routing system-level operations to multiple targets.

1.5.1 Fine-grain Task Aggregation and Coordination on GPUs

Chapter 4 explores how to dynamically aggregate scalar tasks, initiated by GPU threads, into data-parallel tasks (e.g., a GPU kernel). Specifically, in general-purpose GPU (GPGPU) computing, data are processed by concurrent threads executing the same function. This model, dubbed single-instruction/multiple-thread (SIMT), requires programmers to coordinate the synchronous execution of similar operations across thousands of data elements. To alleviate this programmer burden, Gaster and Howes outlined the channel abstraction, which facilitates dynamically aggregating asynchronously produced fine-grain work into coarser-grain tasks [28]. However, no practical implementation had been previously proposed.

To this end, we propose implementing channels as aggregation buffers that are managed by SIMD-direct aggregation. To demonstrate the utility of channels, we present a case study that maps the fine-grain, recursive task spawning in the Cilk programming language to channels by representing it as a flow graph. To support data-parallel recursion in bounded memory, we propose a hardware mechanism that allows wavefronts to yield their execution resources. Through channels and wavefront yield, we implement four Cilk benchmarks. We show that Cilk can scale with the GPU architecture, achieving speedups of as much as 4.3x on eight compute units.

1.5.2 Gravel: Fine-grain GPU-initiated Network Messages

Chapter 5 explores how to dynamically aggregate network messages initiated by GPU threads into large buffers that are more suitable for network transmission. Specifically, GPUs are prevalent in distributed systems because they provide massive parallelism in an energy-efficient manner. Unfortunately, existing programming models (summarized earlier in Section 1.3) make it difficult to efficiently route a GPU-initiated message through the network. For example, the coprocessor

model forces programmers to manually route messages through the host CPU. Other models allow GPU-initiated communication, but are inefficient for small messages (e.g., less than a kilobyte).

To better enable the GPU to efficiently initiate small messages, we introduce Gravel, which draws inspiration from the GPU's memory coalescer. The coalescer operates across a data-parallel instruction to combine accesses to the same cache line, which increases bandwidth by amortizing memory system overhead. Similarly, Gravel uses aggregation buffers, managed by indirect aggregation, to combine GPU-initiated messages being sent to the same network destination. A key optimization is to amortize shared-memory synchronization across as many data-parallel lanes as possible, but branch divergence can limit the number of lanes that execute together. Thus, we explore diverged work-group-level semantics to enable optimal synchronization from divergent code.

Using Gravel, we distribute six applications, each with frequent small messages, across a cluster of eight AMD APUs connected by InfiniBand. Compared to one node, these applications run 5.3x faster, on average. Furthermore, we show that Gravel is more programmable and usually more performant than prior GPU networking models.

Next, Chapter 2 provides a brief primer on the GPU's programming model, architecture, and execution model. Chapter 2 is also useful in that it introduces OpenCL's terminology for GPU concepts, which is used for the remainder of this document. Those familiar with GPU architecture and OpenCL terminology may wish to skip ahead to Chapter 3.

2. GPU PRIMER

This section gives an overview of today's GPU programming abstractions and how they help programmers coordinate structured parallelism so that it executes efficiently on the GPU's data-parallel hardware.

2.1 GPU Programming Model

The GPU's underlying execution resource is the single-instruction/multiple-data (SIMD) unit, which is a number of functional units, or lanes, that execute in lockstep (64 on AMD GPUs and 32 on NVIDIA GPUs [29]). GPGPU languages, like OpenCL™ and CUDA, are called single-instruction/multiple-thread (SIMT) because they map the programmer's view of a thread to a SIMD lane. Threads executing on the same SIMD unit in lockstep are called a wavefront (warp in CUDA). In SIMT languages, a task is defined by three components:

1. A function (called a kernel).
2. Data (the kernel's parameters).
3. A dense 1- to 3-dimensional index space of threads called an NDRange (grid in CUDA).

Figure 2-1 shows an OpenCL NDRange. The smallest unit is a *work-item* (thread in CUDA), which is a SIMT thread that maps to a SIMD lane. Work-items are grouped into 1- to 3-

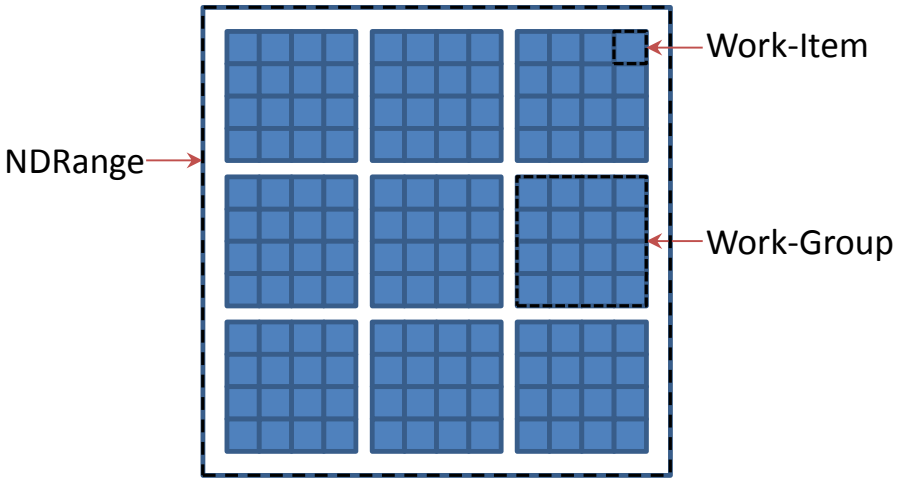


Figure 2-1. OpenCL NDRange.

dimensional arrays called *work-groups* (thread blocks in CUDA). Multiple work-groups are combined to form the *NDRange*. The NDRange helps programmers schedule structured parallelism to the GPU's data-parallel hardware, but makes mapping unstructured parallelism difficult.

Work-items in a work-group communicate using work-group-level barriers and hardware caches—including a programmer-managed scratchpad cache. These primitives enable work-group-level operations, which use the work-items in a work-group to index and process a data array. An important work-group-level operation is reduction, which reduces an array to a single result (e.g., sum, maximum). For example, given the array, `A=[2,1,0,5]`, reduce-to-sum returns `2+1+0+5=8`. Another important operation is prefix-sum, which calculates an array's running total. For example, the prefix sum of `A` is `[0, 0+2=2, 0+2+1=3, 0+2+1+0=3]`.

2.2 GPU Architecture

Figure 2-2 highlights important architectural features of a generic GPU. Compute units (CUs, called streaming multiprocessors in CUDA), are defined by a set of SIMD units, a pool of

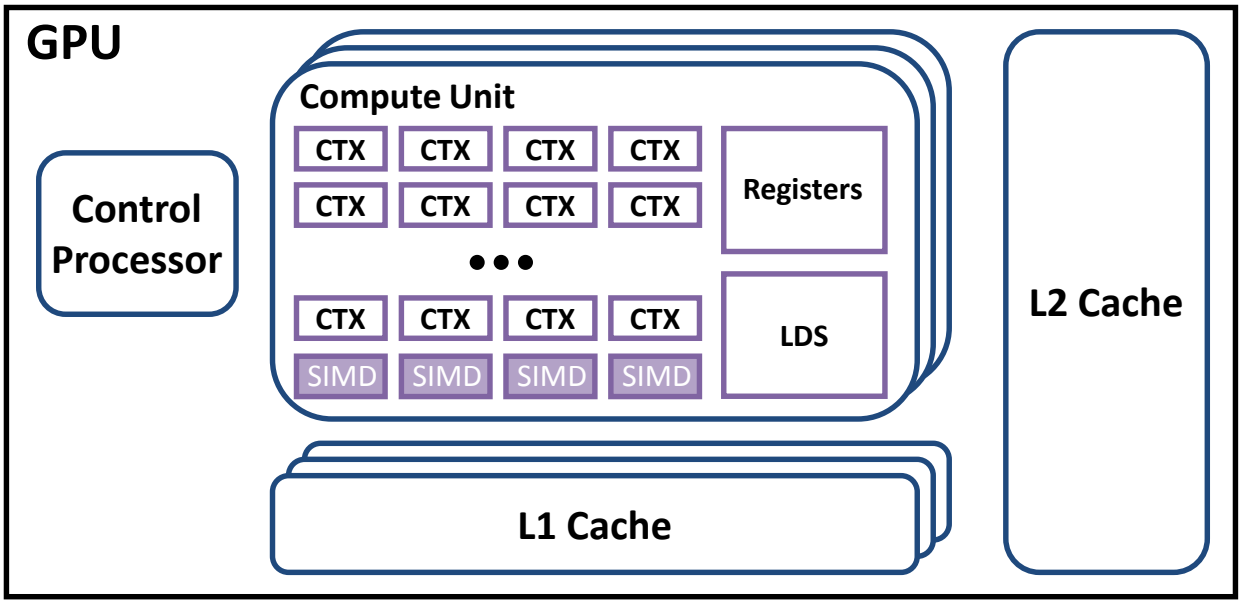


Figure 2-2. Generic GPU architecture.

wavefront contexts (CTX), a register file, and a programmer-managed cache called local data store (LDS, or shared memory in CUDA). A CTX maintains state for an executing wavefront. Each wavefront owns a slice of the register file that partially defines its state. Each CU has a private L1 cache that feeds into a shared L2 cache.

The control processor, also shown in the generic GPU architecture picture (Figure 2-2), obtains SIMT tasks from a set of task queues that it manages. To schedule a task, the control processor assigns its work-groups to available CUs. The control processor also coordinates simultaneous graphics and compute, virtualizes GPU resources, and performs power management. To carry out its many roles, this front-end hardware has evolved from fixed function logic into a set of scalar processors managed by firmware.

Figure 2-3 shows an HSA-compatible AMD APU, which integrates the GPU (depicted earlier in Figure 2-2) onto the same die as the CPU. While all of today's discrete GPUs and many of today's integrated GPUs provide some combination of coherent and incoherent caches, integrated GPUs that adhere to heterogeneous system architecture (HSA) [30] or Intel's Graphics Gen9 GPU architecture [31] both provide a fully coherent cache hierarchy. For example, the GPU and CPU in Figure 2-3 could be kept coherent through a memory-side directory (not shown). HSA also provides a virtual address space that spans the CPU and the GPU. Finally, it is feasible to allow

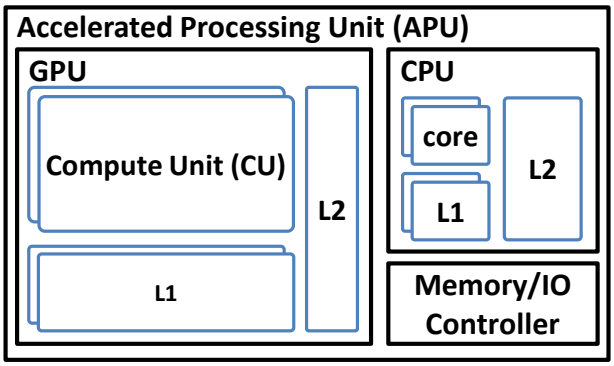


Figure 2-3. CPU-GPU integration.

the GPU to access the APU's I/O controller. Nonetheless, most GPU drivers and programming languages lack APIs to access I/O because it is not clear how to do so efficiently and correctly.

2.3 Single-instruction/multiple-thread (SIMT) Effects

GPU programming languages, like CUDA [29] and OpenCL [32], are SIMT because they present a work-item as the unit of execution. But GPUs execute wavefronts, which exposes two performance effects. Branch divergence, depicted in Figure 2-4a, occurs when work-items in a wavefront encounter different control paths. GPUs use hardware predication to execute branches, which causes some execution units to be idle.

Each CU has a coalescer, which operates across the CU's single wavefront-level cache port to combine memory operations that target the same cache line. Memory divergence, depicted in Figure 2-4b, occurs when work-items in a wavefront access different cache lines and is bad because wavefronts stall until all of their cache lines are accessed.

(a) Branch divergence. *(b) Memory divergence.*

Figure 2-4. SIMT effects.

3. MODELS FOR GPU-INITIATED SYSTEM-LEVEL OPERATIONS

In Chapter 1, we introduced three programming models to execute a system-level operation from the GPU. We briefly review each model and its associated pros and cons below.

1. **Coprocessor model:** Disallows a GPU work-item to directly execute a system-level operation [19][20]. Instead, programmers are forced to refactor their programs so that system-level operations are executed by the host CPU either before or after a GPU kernel executes. To obtain peak performance, programmers must explicitly software pipeline network communication and GPU computation. This can be tedious as it requires breaking the computation into sub-computations that can execute independent of each other.
2. **Operation-per-lane model:** Enables work-items to directly execute system-level operations at any point in a GPU kernel. While this is the ideal programming model and it naturally overlaps communication with computation, previous implementations of the operation-per-lane model are either inefficient [23], require eccentric hardware [18][22], or burden programmers to manage SIMT effects related to exporting a system-level operation [21][23].
3. **Coalesced APIs:** Enables a system-level operation to be executed from the GPU, but requires every work-item in the same work-group to participate in a coalesced API call with identical arguments [24][25][26]. This constraint can be awkward when adjacent work-items execute incompatible system-level operations (e.g., send network messages to different destinations). Similar to the operation-per-lane model, this model implicitly overlaps communication and computation.

In Section 3.1, we use an example to demonstrate that the coprocessor model and coalesced APIs can be awkward programming interfaces for *irregular applications*, which are applications that use system-level operations frequently, unpredictably, and in a fine-grain manner. The

example is inspired from our work on fine-grain GPU-initiated network messages (Chapter 5), but many of the issues apply more broadly to executing a system-level operation from the GPU.

In Section 3.2, we describe how the operation-per-lane model can be supported efficiently on a modern GPU. Our key insight is to use aggregation buffers as a producer/consumer medium to coordinate system-level operations and to amortize the high-cost of producer/consumer synchronization across the GPU's data-parallel lanes.

3.1 Case Study: Distributing GUPS in each Model

This section explores the programmability and operation of each model for executing system-level operations from the GPU. In particular, the three previously proposed models—coprocessor, operation-per-lane, and coalesced APIs—were not designed to handle the fine-grain and unpredictable system-level operations that frequently occur in irregular applications. Thus, we first try to understand how these prior models can accommodate such operations, which incur very high overhead with a naïve implementation. Specifically, we study how to distribute the giga-updates per second (GUPS) micro-benchmark, which requires system-level operations to send network messages. Sending messages is achieved by routing them through the network interface (NI), which is external to the GPU.

In GUPS, a distributed array, A, is incremented at random offsets obtained from a second local data structure [33]. Table 3-1 shows pseudo-code for each model and Table 3-2 shows line counts for real code. A recurring theme is to amortize per-message overhead by combining messages that share the same destination into aggregation buffers, which act as large *per-node queues* that are suitable for network transmission. Thus, from the GPU's perspective, system-level operations (i.e., outgoing network messages) have multiple targets (i.e., messages target the per-node queue that corresponds to their destination).

Table 3-1. GUPS pseudo-code.

`A` is the array being updated. There is a slice of `A`, at the same virtual address, on each node. `B` is a local array of offsets into `A`. `C` is a local array of destinations. `GRID_ID` is a per-work-item identifier used to index data.

```
--- GPU kernel ---
 1: gups(B, C, Qs):
 2:   for each node targeted by my work-group:
 3:     if node == C[GRID_ID]:
 4:       MyOff = leader_level_reserve(&Qs[node])
 5:       Qs[node][MyOff] = B[GRID_ID]

--- host code ---
 6: for (idx = 0; idx < len(B), idx += Q_SZ):
 7:   gups(&B[idx], C, Qs) # on GPU, GRID_WIDTH=Q_SZ
 8:   for each node:
 9:     send Qs[node] to node
10:   for each node:
11:     receive Q from node
12:     for each offset in Q:
13:       A[offset]++
```

(a) Coprocessor model.

```
--- GPU kernel ---
14: gups(A, B, C):
15:   shmem_inc(A + B[GRID_ID], C[GRID_ID])

--- host code ---
16: gups(A, B, C) # on GPU, GRID_WIDTH=len(B)
```

(b) Message-per-lane model & Gravel.

```
--- GPU kernel ---
17: gups(A, B, C):
18:   # allocate data-structures in GPU's scratchpad
19:   int64_t ptrs[WG_SIZE]
20:   int dests[NODE_COUNT]
21:   int cnts[NODE_COUNT]
22:   # After sort: ptrs -> list of per-node Qs; dests
23:   # -> destination list and cnts -> list of
24:   # per-node Q sizes. dcnt = # of destinations.
25:   dcnt = sort(ptrs, dests, cnts, A, B, C)
26:   off = 0
27:   for (d = 0; d < dcnt; d++):
28:     sync_inc_list(&ptrs[off], dests[d], cnts[d])
29:     off += cnts[d]

--- host code ---
30: gups(A, B, C) # on GPU, GRID_WIDTH=len(B)
```

(c) Coalesced APIs.

Table 3-2. Lines of code for GUPS for each model.

	Coprocessor	*Msg-per-lane & Gravel*	*Coalesced APIs*
host	296	174	187
GPU	46	19	131
total	342	193	318

Table 3-3. Ranking different GPU networking models.

	coprocessor	**msg-per-lane**	**coalesced APIs**	**Gravel**
SIMT utilization	*	✓	*	✓
large messages	✓	*	*	✓
efficient sync	✓	✗	✓	✓
programmability	✗	✓	*	✓

* *Good for prior workloads studied; bad for small unpredictable messages.*

We use four criteria to summarize the benefits and limitations of each model: (1) *SIMT utilization*, described in Section 2.3; (2) *large messages*, meaning that messages are large enough to amortize network overhead; (3) *efficient synchronization*, meaning that work-items coordinate to use the NI efficiently; and (4) *programmability*, meaning that applications are simpler (e.g., fewer lines of code). Table 3-3 summarizes how each model ranks across these four criteria.

3.1.1 Coprocessor Model

In the coprocessor model, programmers write CPU code to handle network communication before and after each GPU kernel. GPUDirect RDMA [19] and CUDA-aware MPI [20] are two examples focused on GPU networking that follow this model. To implement an irregular application, a

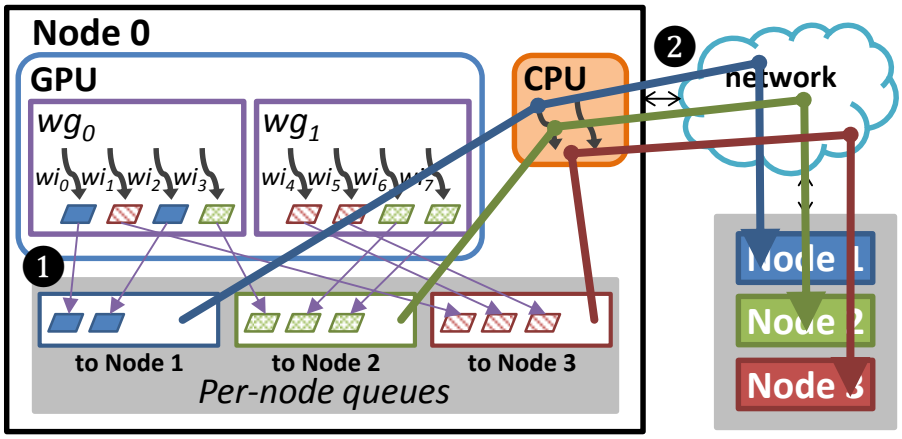


Figure 3-1. Using the coprocessor model to distribute GUPS.

programmer might manually organize messages into per-node queues (Figure 3-1), which exposes several low-level issues. Specifically, the programmer must avoid overflowing a queue, manually send and receive the queues, and overlap the sends/receives with GPU execution. The GPU code must efficiently insert messages into the per-node queues.

The pseudo-code (Table 3-1a) avoids overflowing a queue by chunking the updates (lines 6-7). Specifically, each chunk is sized to match the per-node queue size. This enables each queue to handle the worst case, where all work-items send messages to the same node. Chunking also helps to overlap communication (lines 8-11) and computation. On the GPU, work-groups use leader-level synchronization to efficiently reserve space in the queues (line 4). Note, work-group-level synchronization occurs once per destination (lines 2-3), which causes branch and memory divergence.

3.1.2 Operation-per-lane Model

In prior work, the message-per-lane model (Figure 3-2) burdens programmers with managing GPU-initiated system-level operations in a SIMT-efficient way (e.g., DCGN [23] and CUDA's dynamic parallelism [21]) or requires special hardware (e.g., GGAS [18] and NVSHMEM [22]) to route a wavefront's messages between the GPU's SIMT lanes and the NI. Here, we assume the

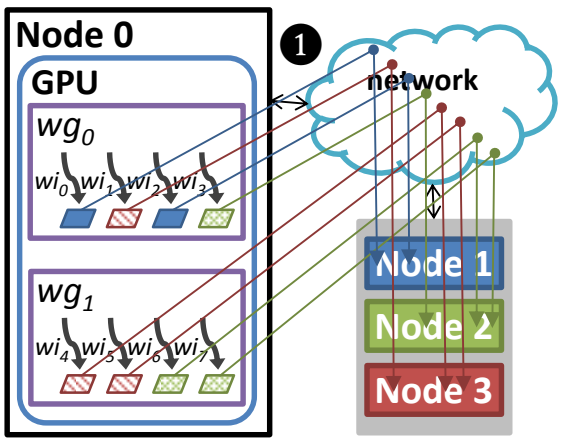


Figure 3-2. Using the operation-per-lane model to distribute GUPS.

latter approach, where SIMT issues are hidden from the programmer. We also note that aggregation buffers, discussed in Section 3.2, can achieve this effect without special hardware.

Table 3-1b shows pseudo-code for the message-per-lane model. After launching the GPU kernel (line 16), work-items update slices of A (line 15). Table 3-2 shows that this model (i.e., 193 lines) is more programmable than the coprocessor model (i.e., 342 lines). Unfortunately, the messages generated by work-items are too small for efficient network transmission.

3.1.3 Coalesced APIs

Coalesced APIs, shown in Figure 3-3, are designed to be executed by all work-items in a work-group at the same time and with identical arguments (e.g., destination, command, payload). GPUnet [24] and GPUrdma [25] are examples of prior work on GPU networking that provide coalesced APIs. GPUfs, which focuses on enabling GPUs to directly operate on files, also provides coalesced APIs [26]. At first glance, this model seems to degenerate to the message-per-lane model for small random messages. However, the pseudo-code in Table 3-1c shows that a tenacious programmer can use the GPU's scratchpad (lines 18-21) to sort a work-group's messages by destination (lines 22-25). A counting sort, where the keys are the destination IDs, works well [34].

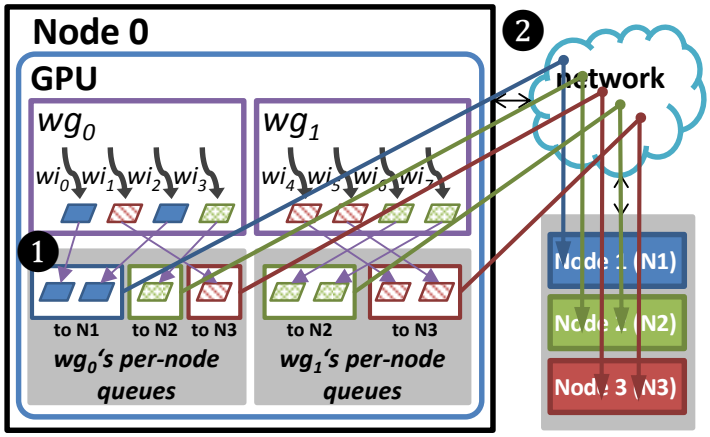


Figure 3-3. Using coalesced APIs to distribute GUPS.

The sort outputs a contiguous list of messages for each destination targeted by a work-group. The pseudo-code then uses a coalesced API, `sync_inc_list`, to send each list.

Our coalesced APIs version (318 lines) is 1.6x more code than the message-per-lane model (193 lines). One issue is the amount of scratchpad used (i.e., a work-group with 256 work-items uses 4 kB of scratchpad). A second problem is that aggregating across a work-group (instead of the entire GPU) generates small per-node queues. Finally, a third issue is that coalesced APIs are invoked for each destination, which degrades SIMT utilization.

3.2 Coordinating GPU Thread Access to the Aggregation Buffers

In the previous section, we showed that the operation-per-lane model provides the best programmability. We also observed that it can perform poorly when system-level coordination is required for each system-level operation executed from the GPU.

In this section, we explore using aggregation buffers to amortize the cost of system-level coordination. Specifically, our goal is to route system-level operations, initiated by work-items on the GPU according to the operation-per-lane model, to their respective aggregation buffers. Packing system-level operations into aggregation buffers limits system-level coordination to occur once per aggregation buffer instead of once per system-level operation.

The key challenge is to efficiently deposit system-level operations into their respective aggregation buffer. Depositing a system-level operation means writing all of its attributes into the buffer so that the operation can be consumed where the system-level resource resides. To efficiently deposit operations, we leverage leader-level synchronization (Section 3.2.1), which occurs at SIMT granularity (instead of work-item granularity). Thus, each aggregation buffer can be viewed as a producer/consumer queue where producers are SIMT groups that use leader-level synchronization to coordinate.

Using leader-level synchronization, we develop two schemes to route system-level operations from the GPU's data-parallel lanes to the correct aggregation buffer—SIMT-direct aggregation and indirect aggregation (Section 3.2.2). We also present a quantitative comparison of SIMT-direct aggregation and indirect aggregation (Section 3.2.3). Finally, we conclude this section with a discussion on use cases for aggregation buffers (Section 3.2.4).

3.2.1 Leader-level Synchronization

Leader-level synchronization is a crucial building block used to efficiently reserve space in an aggregation buffer for a SIMT group's system-level operations. This concept is demonstrated in Figure 3-4. In the figure, data-parallel threads executing in the same SIMT group coordinate to route their system-level operations to the host CPU through a shared-memory producer/consumer queue. Specifically, at time ❶, the threads in SIMT group 1 coordinate to elect a leader thread, t_3, which reserves four slots in the queue on behalf of its entire SIMT group. Importantly, the leader thread is chosen without invoking the memory system's expensive coherence mechanisms. For example, SIMT constructs (introduced in Chapter 2) such as data-parallel operations (e.g., reduction or prefix sum), scratchpad memory, work-group-level barriers, and lockstep execution can all be leveraged to elect a leader thread, instead of using shared-memory synchronization.

Referring back to the figure, after t_3 has reserved four slots, the threads in SIMT group 1, t_0-t_3, deposit their operations into the queue (time ❷). Specifically, depositing an operation entails

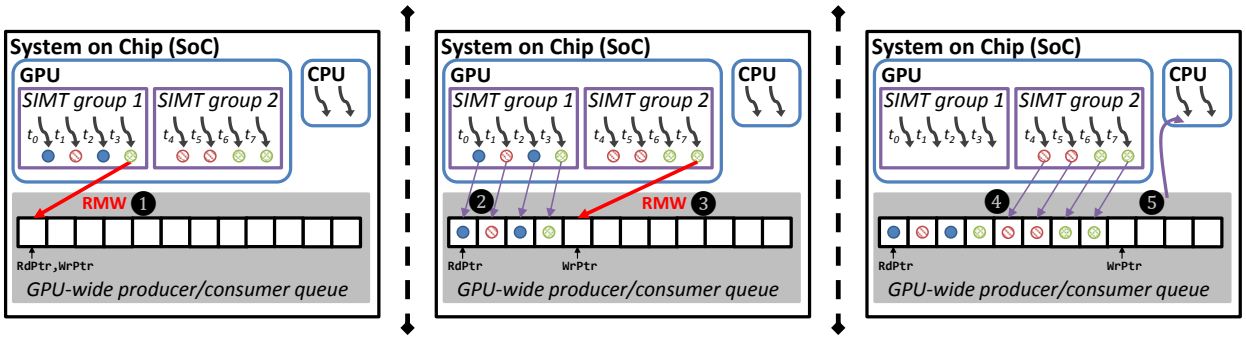


Figure 3-4. Leader-level synchronization.

writing all of its attributes (e.g., command, arguments) into the queue. Similarly, at time ❸ a leader thread, t_7, is elected to reserve four queue slots for the threads in SIMT group 2, t_4-t_7, which deposit their operations (time ❹) after the slots have been reserved. In this example, threads on the host CPU retrieve operations from the queue and execute them (time ❺). A subtle detail, not shown in the figure, is that a CPU thread dequeues several operations at once to amortize the high-cost of producer/consumer synchronization.

3.2.2 Aggregation Schemes

In this section, we explore how to use leader-level synchronization to route operations to their respective aggregation buffer. The discussion is organized around two questions, discussed below and summarized by Figure 3-5.

1. **How do we define a SIMT group?**

A SIMT group can be defined to be a wavefront or a work-group. A wavefront is convenient because it corresponds to the GPU's execution granularity, which means that work-items in a

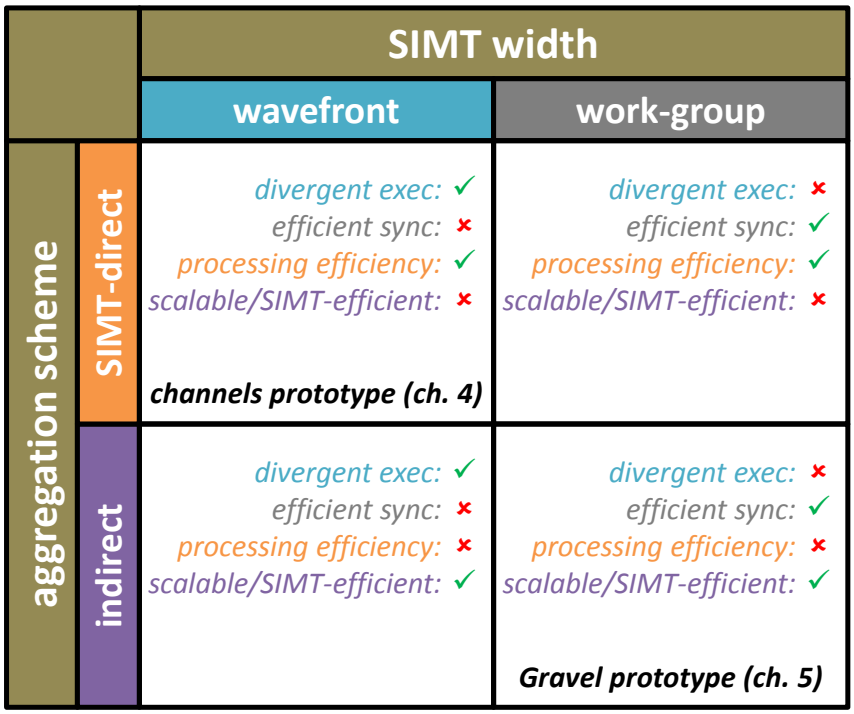


Figure 3-5. Aggregation taxonomy.

wavefront can synchronize in a well-defined way from divergent code (this concept is labeled *divergent exec* in Figure 3-5). In contrast, a work-group can comprise more than one wavefront and when those wavefronts have non-overlapping control flow it is not clear how to synchronize across them.

The advantage of a work-group is that it can provide more work-items to amortize the cost of leader-level synchronization. This concept is labeled *efficient sync* in Figure 3-5. In Chapter 4, we describe our channels prototype, which makes the SIMT group a wavefront. The channels prototype was our first effort to implement leader-level synchronization and making the SIMT group a wavefront seemed like the only plausible option. In Chapter 5, we describe Gravel, which makes a SIMT group a work-group. To enable leader-level synchronization from divergent control flow we explore a diverged work-group-level semantic, which guarantees that all of a work-group's wavefronts are present during leader-level synchronization.

2. **How do we route system-level operations originating from the same SIMT group to different aggregation buffers?**

We consider two ways to export operations to the correct aggregation buffer: SIMT-direct aggregation (prototyped in Chapter 4) and indirect aggregation (prototyped in Chapter 5). In

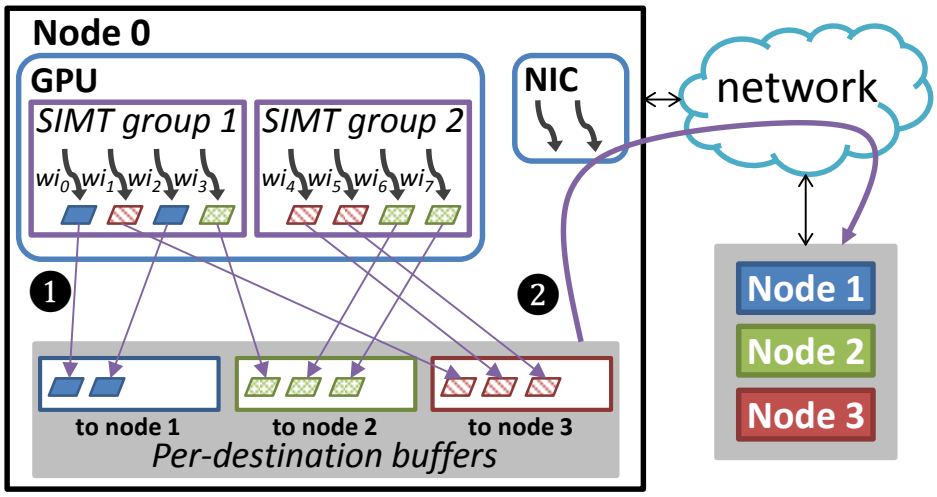


Figure 3-6. SIMT-driven aggregation.

SIMT-direct aggregation, which is shown being used to combine network messages in Figure 3-6, work-items write their operations into their respective aggregation buffer directly. Specifically, work-items within a work-group coordinate to identify which work-items can invoke leader-level synchronization together (time ❶). For example, in Figure 3-6 leader-level synchronization can be applied to wi_0 and wi_2, because they execute compatible operations and belong to the same SIMT group. In this example, threads associated with the NIC monitor the aggregation buffers (labeled per-node buffers in the figure) and send them through the network after they reach their maximum occupancy or exceed a timeout (time ❷).

The second aggregation scheme that we consider is indirect aggregation, which is shown being used to aggregate network messages in Figure 3-7. In the figure, messages initiated by work-items in the same SIMT group are collected in per-SIMT-group buffers (time ❶). The per-SIMT-group buffers are then transferred to a dedicated aggregator, which copies the messages into their respective aggregation buffers (time ❷). The aggregator can be implemented in hardware or as a scalar thread. In the example, the aggregator is realized through the scalar threads residing inside of the NIC. Finally, the aggregation buffers are sent through the network after they accumulate enough messages or exceed a timeout (time ❸).

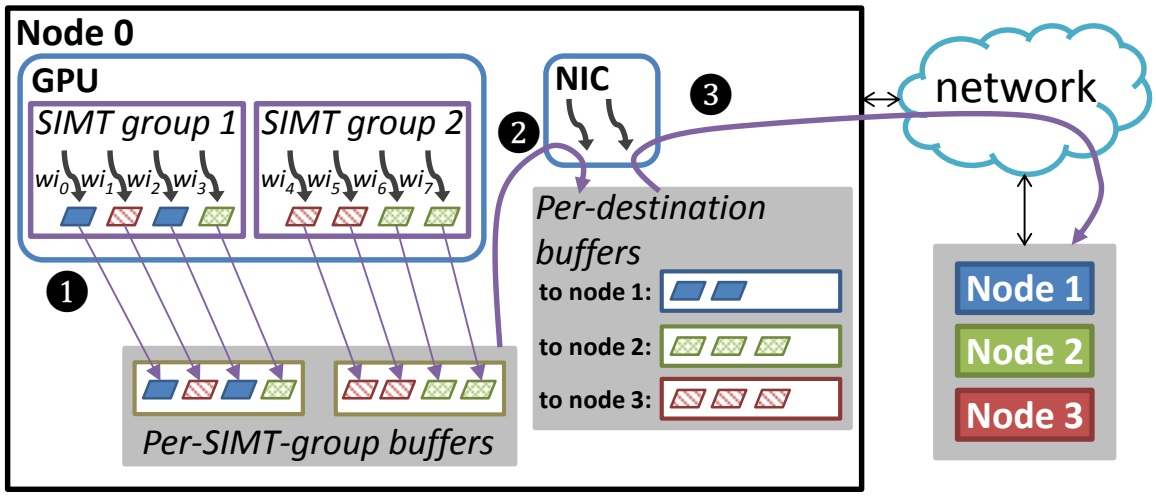


Figure 3-7. Indirect aggregation.

Algorithmically, SIMT-direct aggregation moves less bytes than CPU-side aggregation. This concept is labeled *processing efficiency* in Figure 3-5. Specifically, indirect aggregation has to process each message twice, whereas messages are processed once by SIMT-direct aggregation. Furthermore, indirect aggregation requires work-items to communicate each operation's attributes to the aggregator so that an operation can be directed to the correct aggregation buffer. SIMT-direct aggregation encodes this information once for each aggregation buffer.

In practice, SIMT-direct aggregation is hard to implement efficiently. In particular, it is hard to avoid branch divergence when operations originating from the same SIMT group target different aggregation buffers. It is also hard to minimize memory divergence because work-items write the aggregation buffers in an unpredictable manner. Finally, SIMT-direct aggregation invokes leader-level synchronization once per aggregation buffer targeted by a SIMT group. In contrast, indirect aggregation invokes leader-level synchronization once per work-group. These concepts are labeled *scalable/SIMT efficiency* in Figure 3-5.

3.2.3 Sensitivity Study

We wanted to evaluate GUPS with both SIMT-direct aggregation and indirect aggregation on clusters of different sizes to confirm the qualitative analysis in the previous section (larger clusters demand more aggregation buffers). Unfortunately, we were unable to access a large enough

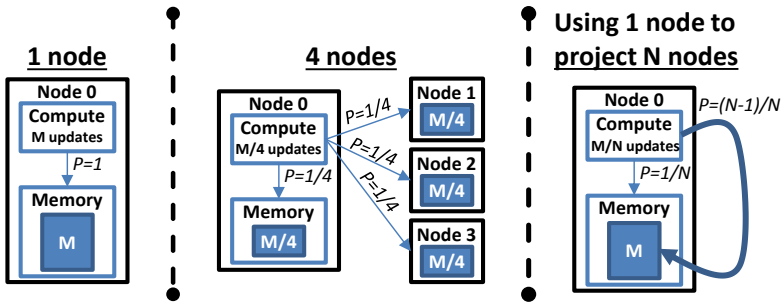


Figure 3-8. Using one node to project GUPS at N nodes.

cluster. Instead, we developed a GUPS-inspired micro-benchmark, depicted in Figure 3-8, to project the performance of GUPS on a cluster with N nodes using one node.

To understand the experiment, consider some concrete examples. First, the left-most scene in the figure shows a GUPS execution of M updates on one node. All updates occur on that node with 100% probability (i.e., P=1).

Now, consider executing the same M updates across four nodes (shown in the middle scene). Node 0 is responsible for 1/4th (i.e., M/4) of those updates. Furthermore, because the array is evenly distributed across the four nodes, 1/4th of node 0's M/4 updates execute locally and the rest are equally distributed across the other three nodes. The key observation is that the amount of traffic leaving a node (e.g., 3/4th of node 0's M/4 updates) is the same as the amount of traffic coming in (e.g., each other node sends node 0 1/4th of its M/4 updates).

With this understanding, we employ the experiment shown in the right-most scene of Figure 3-8. Specifically, given a GUPS computation of M updates across N nodes, we execute M/N

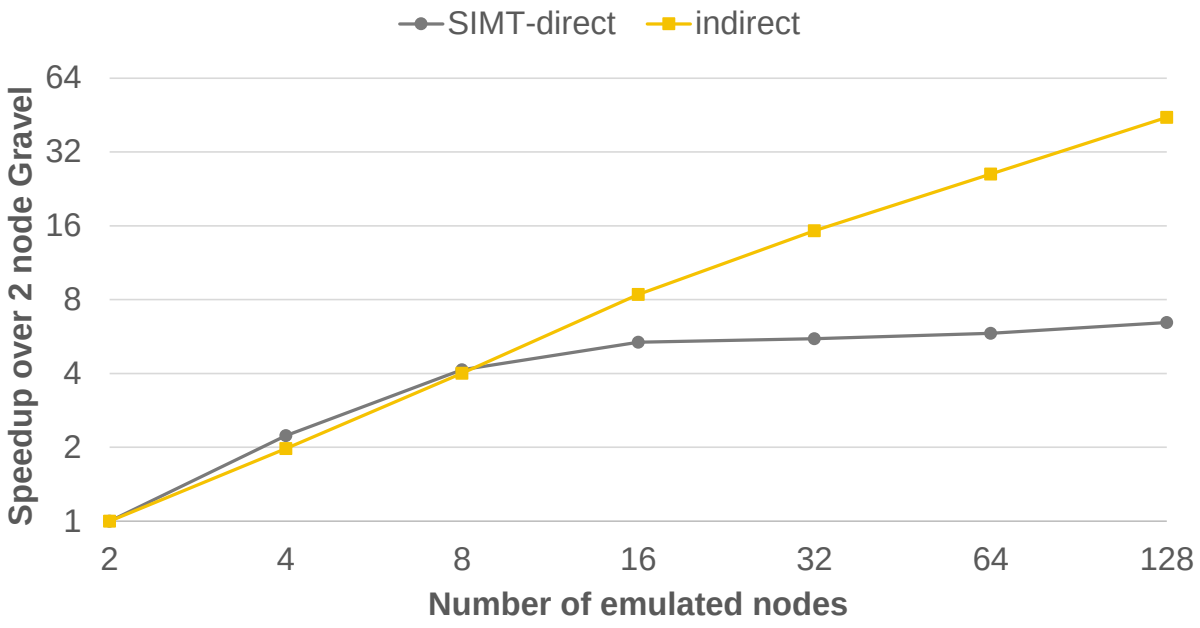


Figure 3-9. GUPS projected scalability.

updates on one node. We neglect the remaining M – M/N updates. We allocate the entire array on the one node (instead of distributing it). Finally, we loop all outgoing traffic back into the node.

Figure 3-9 shows how the two aggregation schemes perform at different cluster sizes. The results are normalized to SIMT-direct aggregation at two nodes. As predicted, SIMT-direct aggregation is more efficient on smaller clusters (despite the fact that it experiences branch and memory divergence). As the number of nodes in a cluster increases beyond eight nodes, indirect aggregation does better. This is because in SIMT-direct aggregation a SIMT group invokes leader-level synchronization once per aggregation buffer whereas indirect aggregation invokes leader-level synchronization once per SIMT group (irrespective of the number of aggregation buffers).

3.2.4 Aggregation Buffer Use Cases

We conclude this chapter with a discussion of different situations that can benefit from aggregation buffers. First, as previously discussed, aggregation buffers enable system-level operations such as dynamic memory allocation, file I/O, network I/O, printf, and system calls to be offloaded to a device outside of the GPU's data-parallel hardware. That device can be the host CPU, the NIC, or an embedded controller inside of the GPU.

In certain cases, aggregation buffers can be used to batch system-level operations. For example, tasks can be coalesced into larger data-parallel tasks that are more likely to saturate data-parallel hardware. This idea is explored in detail in Chapter 4. Similarly, aggregation buffers can be used to batch network messages. This idea is explored in detail in Chapter 5. One can also imagine batching systems calls to amortize the overhead of calling into the OS kernel.

A third use case for aggregation buffers is to improve the coupling of discrete cards with their host CPU. Specifically, discrete cards are typically connected over PCIe and companies often provide mechanisms for CPUs and discrete GPUs to access each other's memories. Thus,

aggregation buffers could be used to amortize PCIe overheads associated with a distant memory connected over PCIe.

Finally, aggregation buffers can also be used to mitigate branch divergence. Specifically, they could be used to dynamically regroup wavefronts based on the control flow of their underlying work-items. This idea is similar to dynamic micro-kernels, which were previously proposed by Steffen and Zambreno [35].

4. FINE-GRAIN TASK AGGREGATION AND COORDINATION ON GPUS

In this chapter, we explore a new tasking abstraction called channels—originally proposed by Gaster and Howes [28]. Specifically, channels are multi-producer/multi-consumer data queues that reside in virtual memory and act as a medium through which producers and consumers communicate in a data-flow manner. A given channel holds fine-grain data items—which we call channel elements (CEs)—that are processed by the same function. Constraining each channel to be processed by exactly one function facilitates efficient aggregation of work that then can be scheduled onto the GPU's data-parallel hardware.

Note, channels are aggregation buffers and enqueueing a CE onto a channel is a system-level operation. Furthermore, our goal is to enable programmers to inject a CE into a channel using operation-per-lane semantics. Channels are interesting to explore for two reasons. First, as we show later, they have potential to expand GPGPU programming by enabling new programming abstractions such as Cilk. Second, they epitomize the typical system-level operation, which means that many of the insights developed here can be applied to other system-level operations.

Furthermore, while Gaster and Howes defined channels, they did not propose an implementation, leaving designers to question their practicality. To this end, we propose and evaluate the first implementation of channels. Notably, the channel implementation proposed in this chapter is SIMT-direct (as described in Section 3.2.2). To obtain acceptable performance, our implementation is lock-free, non-blocking, and optimized for SIMT accesses.

The finer-grain parallelism enabled by channels requires more frequent and complex scheduling decisions. To manage this behavior, we leverage the GPU's CP, which typically is implemented as a small, in-order, programmable processor. We use this tightly integrated processor to monitor the channels, manage algorithmic dependencies among them, and dispatch

ready work to the GPU. Our analysis suggests that replacing the existing in-order processor with a modest out-of-order processor can mitigate the scheduling overheads imposed by dynamic aggregation.

Because no existing programs are written specifically for channels, we evaluate our implementation by mapping flow graph-based programs to channels. A flow graph is a data-driven graph representation of a parallel application. It is a popular abstraction used by many modern parallel programming languages, including Intel's Threading Building Blocks (TBB) [36]. Flow-graph nodes represent the program's computation, while messages flowing over directed edges represent communication and coordination. We use channels to aggregate individual messages into coarser-grain units that can be scheduled efficiently onto the GPU. Channel-flow graphs increase the diversity of applications that map well to GPUs by enabling higher-level programming languages with less rigid task abstractions than today's GPGPU languages.

We specifically explore mapping programs written in Cilk to channels. Cilk is a parallel extension to C/C++ for expressing recursive parallelism. We define a set of transformations to map a subset of Cilk to a channel-flow graph so that it can execute on a GPU. This presented two important challenges. First, GPUs do not provide a call stack, which CPUs normally use to handle recursion. Our solution is to map Cilk's task tree to "stacks of channels". Second, previous Cilk runtimes use depth-first recursion to bound memory usage. However, although breadth-first scheduling is more effective at populating a GPU's thousands of hardware thread contexts, it requires exponential memory resources [37]. To solve this problem, we propose a bounded breadth-first traversal, relying on a novel yield mechanism that allows wavefronts to release their execution resources. Through channels and wavefront yield, we implement four Cilk workloads and use them to demonstrate the scalability of Cilk in our simulated prototype.

4.1 Channel Definition and Implementation

Gaster and Howes suggested channels to improve on today's coarse-grain GPU task abstractions. In this section, we summarize their vision and propose the first channel implementation, which executes on forward-looking APUs. Our implementation is SIMT-direct in that work-items directly reserve space and enqueue their CEs into a channel. In contrast, the per-node buffers, which are discussed in Chapter 5 and are analogous to channels, are indirect.

4.1.1 Prior Work on Channels

A channel is a finite queue in virtual memory, through which fine-grain data (channel elements, or CEs) are produced and consumed in a data-flow manner. Channels resemble conventional task queues, but differ in three ways:

1. Data in a channel are processed by exactly one function permanently associated with that channel.
2. CEs are aggregated dynamically into structured, coarse-grain tasks that execute efficiently on GPUs.
3. Each channel has a "predicate" function for making dynamic scheduling decisions.

Figure 4-1 shows data moving through channels in an APU-like system that includes a CPU and a GPU connected to a coherent shared memory. The GPU's control processor is extended to

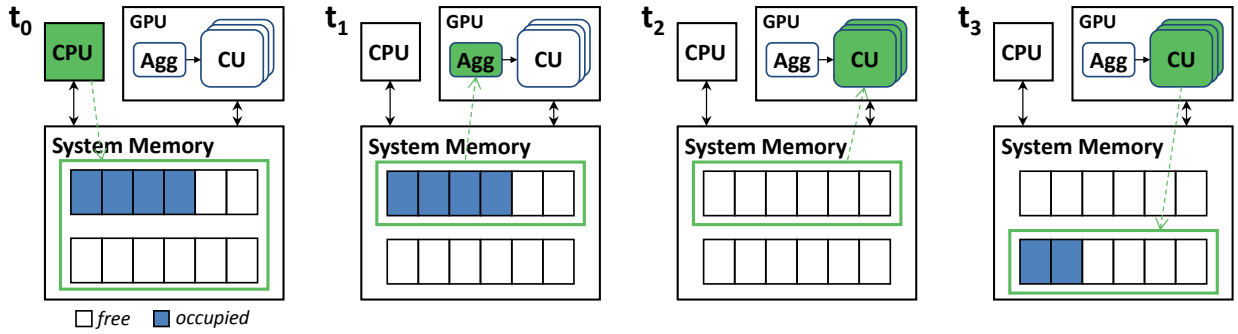


Figure 4-1. Channel flow.

monitor and manage channels. Because we are concerned primarily with this new capability, we call the control processor the aggregator (labeled *Agg* in Figure 4-1) for the remainder of this paper.

At time 0 (t_0) in Figure 4-1, the host initializes two channels and populates them with CEs. At time 1 (t_1), the aggregator, controlled through a user-defined scheduler, probes the channels; it detects enough CEs to justify a dispatch to GPU hardware. The GPU consumes the CEs at t_2 and produces new CEs in a different channel at t_3.

Restricting each channel to processing by exactly one function avoids burdening the aggregator with inspecting individual CEs. This constraint does not limit fine-grain task-parallelism because channels are mapped to shared virtual memory and therefore are visible to all producers.

The predicate is a Boolean function that assists the aggregator in making scheduling decisions. The simplest predicate is one that returns false unless enough CEs are available to populate all of a SIMD unit's lanes. This is what we assume for this paper.

4.1.2 Lock-free Channel Implementation

To realize finer-grain task abstractions on GPUs, we introduce a SIMT-direct multi-producer/multi-consumer queue that is lock-free, non-blocking, and array-based. Lock-free queues have a rich history in the context of CPUs. Early work considered array-based designs [38][39][40], but linked lists are preferred [41][42]. Linked lists are not well suited for GPUs because different work-items in a wavefront consuming adjacent CEs are susceptible to memory divergence, which occurs when the work-items access different cache blocks; if the requests had been to the same cache block, the GPU's coalescing hardware could have merged them. We find that our queue implementation accommodates the high levels of contention that are typical on a massively threaded GPU.

SIMT-direct Channel Implementation

Our array-based channel is implemented as three structures:

1. **Data array:** Buffer for produced CEs.
2. **Control array:** Buffer of data-array offsets, populated by producers and monitored by the aggregator.
3. **Done-count array:** Adjacent data-array elements can share a done-count element. The aggregator monitors the done-count array to free data-array elements in the order they were allocated.

The size of the done-count array is the size of the data array divided by the number of data-array elements that share a done count. The control array is twice the size of the data array. Array elements can be in one of five states:

1. **Available:** Vacant and available for reservation.
2. **Reserved:** Producer is filling, hidden from aggregator.
3. **Ready:** Visible to aggregator, set for consumption.
4. **Dispatched:** Consumer is processing.
5. **Done:** Waiting to be deallocated by aggregator.

Figure 4-2 illustrates two wavefronts, each four work-items wide, operating on a single channel in system memory. For space, the control array is the same size as the data array in the figure, but in practice it is twice the size of the data array. In the text that follows, producers operate on the tail end of an array and consumers operate on the head end.

At time 0 (t_0), the data array's `head` and `tail` pointers are initialized to the same element. Similarly, the control array's `head` and `tail` pointers are initialized to the same element. The control array maintains two tail pointers (`tail` and `reserveTail`) because producers cannot

instantaneously reserve space in the control array and write the data-array offset. All done counts are initialized to 0.

At t_1, each work-item in wavefront 0 reserves space for a CE. Four data-array elements are transitioned to the reserved state by updating the data array's `tail` pointer via compare-and-swap (CAS). At t_2, each work-item in wavefront 1 reserves space for a CE and at t_3 those work-items finish writing their data-array elements.

Data-array elements are made visible to the aggregator by writing their offsets into the control array. Specifically, at t_4, wavefront 1 updates `reserveTail` via CAS to reserve space in the control array for its data-array offsets. At t_5, the offsets are written and at t_6 the control array's `tail`, which is monitored by the aggregator, is updated to match `reserveTail`. The array elements related to wavefront 1 are now in the ready state. The design is non-blocking because wavefront 1 can make its CEs visible to the aggregator before wavefront 0 even though it reserved space after wavefront 0.

At t_7, the data-array elements generated by wavefront 1 are transitioned to the dispatched state when the aggregator points consumers at their respective control-array elements. Those control-array elements also transition to the dispatched state; they cannot be overwritten until their

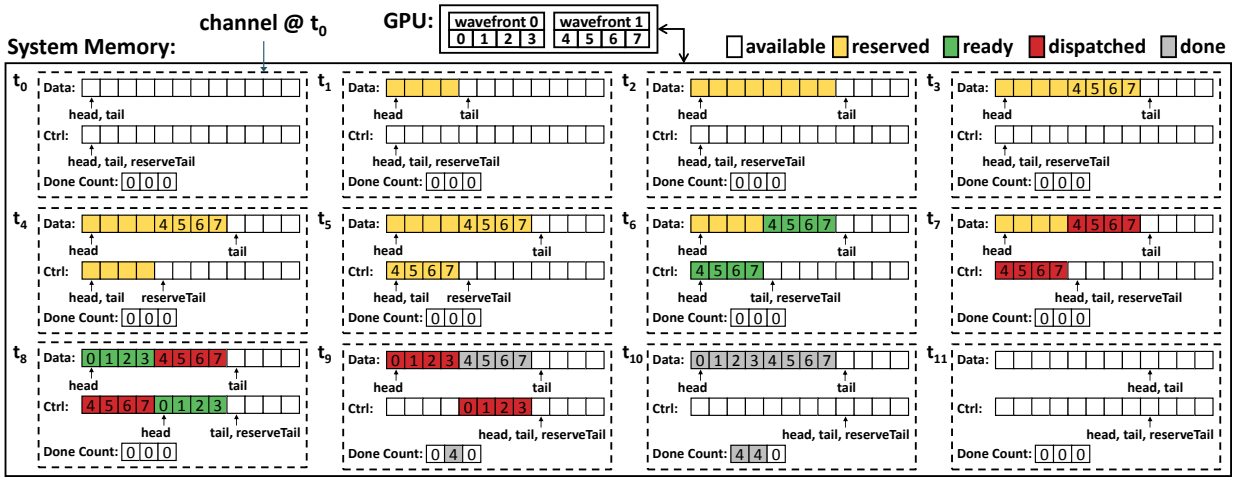


Figure 4-2. SIMT-direct, lock-free, non-blocking channel implementation.

corresponding data-array elements are deallocated because the control array is twice the size of the data array.

At t_8, wavefront 0 finishes writing its data-array elements and makes its CEs visible to the aggregator. At t_9, wavefront 0's CEs are dispatched. Also at t_9, the consumers of wavefront 1's CEs signal that they no longer need to reference the data-array elements by updating their respective done counts atomically; these data-array elements cannot be deallocated before wavefront 0's data-array elements. At t_{10}, the consumers of wavefront 0's CEs update their respective done counts. Finally, at t_{11}, the aggregator deallocates space.

Discussion and Optimization

The array-based channel maps well to the GPU's coalescing hardware. The CUs are responsible for allocation and consumption while the aggregator handles deallocation, which is off the critical path of execution. The aggregator manages the channels without inspecting their individual CEs.

```
 1: int gpuReserveNElements(int numEl, int *tail) {
 2:   int wfTail = 0;
 3:   // 1. Choose one work-item to operate on tail
 4:   bool update = most_sig_work_item();
 5:   // 2. Intra-wavefront prefix sum
 6:   int offset = prefix_sum(numEl);
 7:   int numElToRes = offset + numEl;
 8:   // 3. Intra-wavefront synchronization
 9:   join_wfbarrier();
10:   while(update) {
11:     int oldTail = *tail;
12:     int nextTail = oldTail + numElToRes;
13:     int curTail = CAS(tail, oldTail, nextTail);
14:     if(oldTail == curTail) {
15:       wfTail = oldTail;
16:       update = false;
17:     }
18:   }
19:   wait_at_wfbarrier();
20:   // 4. Broadcast tail to entire wavefront
21:   wfTail = reduction(wfTail);
22:   return (wfTail + offset);
23: }
```

Figure 4-3. GPU fetch-and-update.
(ignores wrapping/overflow)

Space is reserved in the data and control arrays through conditional fetch-and-update (via CAS). By leveraging intra-wavefront communication instructions [43], this operation can be amortized across a wavefront, greatly reducing memory traffic. Figure 4-3 depicts pseudo-code with these optimizations that updates a channel array's tail pointer.

4.2 Programming with Channels

This section proposes a low-level API to interface channels and describes utilizing channels through flow graphs.

4.2.1 Channel API

Table 4-1 shows the channel API. Producers call `talloc` to allocate CEs. An allocated CE is made visible to the aggregator via the `enq` function. A CE must be enqueued to the channel that was specified during its allocation. A consumer obtains work with the `deq` function; the specific channel and offset within that channel are managed by the aggregator. After data is consumed, the aggregator is signaled that deallocation can occur via the `tfree` API.

The `talloc` API enables minimum data movement between producers and consumers because the destination channel is written directly through the pointer that `talloc` returns. Figure 4-5, lines 3-19, demonstrate the API in Table 4-1.

4.2.2 Channel-flow Graphs

Flow graphs comprise a set of nodes that produce and consume messages through directed edges; the flow of messages is managed through conditions. Several popular parallel programming

Table 4-1. Channel API.

API Function	Description
`void *talloc(int id, int cnt)`	allocate `cnt` CEs in channel `id`.
`void enq(int id, void *ptr)`	place CEs at `ptr` in channel `id`.
`void *deq(int id, int off)`	get CE in channel `id` at `off`.
`void tfree(int id, int off)`	free CE in channel `id` at `off`.

languages and runtimes support flow graphs. For example, Intel's TBB provides a sophisticated flow-graph abstraction [36]. MapReduce [44] and StreamIt [45] provide more constrained flow-graph frameworks. GRAMPS, which had a strong influence on the original proposal for channels, explores scheduling flow graphs onto graphics pipelines [46].

Channels facilitate flow graphs with fine-grain messages. A channel-flow graph is specified as a directed graph composed of kernel nodes and channel nodes. A kernel node resembles a GPGPU kernel that consumes and produces data. Kernel nodes are analogous to function nodes in TBB. A channel node is a buffer that accumulates messages produced by kernel nodes and routes them to be consumed by other kernel nodes. Channel nodes are similar to queue nodes in TBB, but bounded.

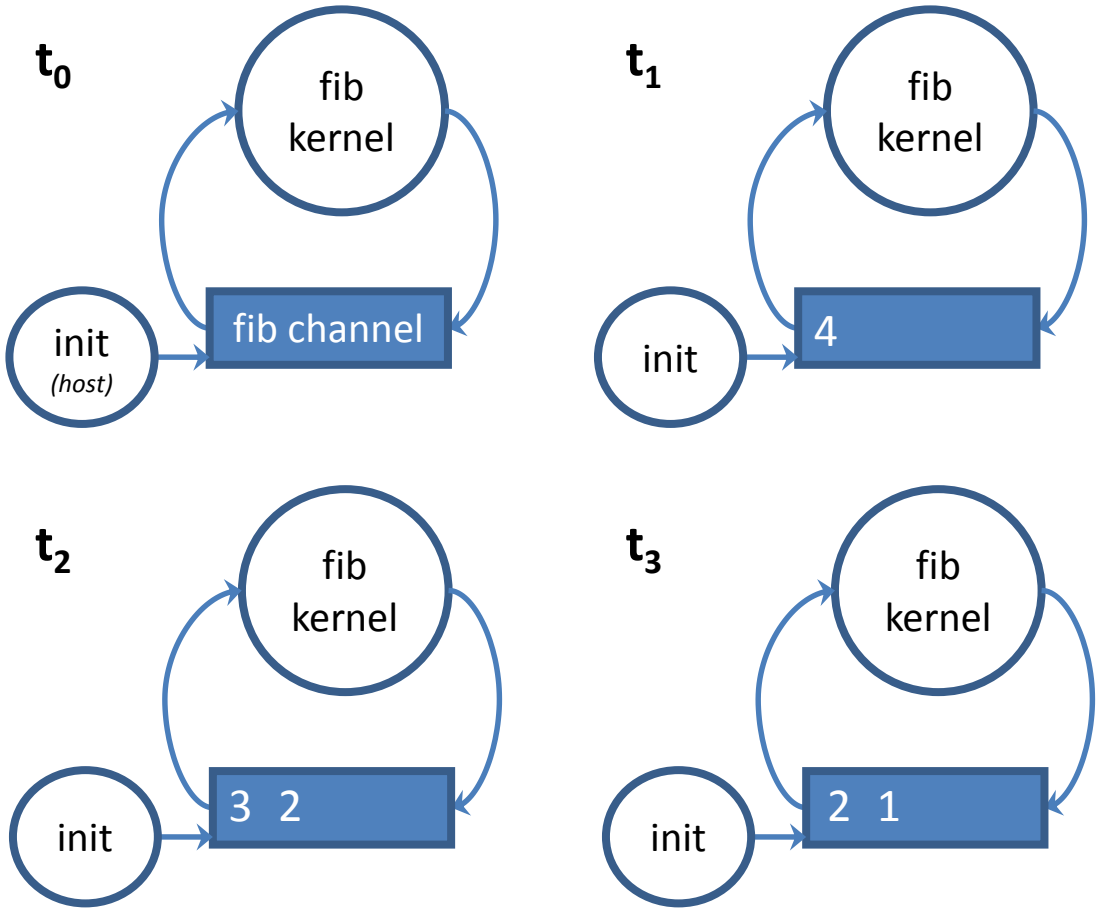


Figure 4-4. Channel-flow graph for naïve Fibonacci.

Figure 4-4 shows an example flow graph to compute the fourth Fibonacci number. At time 0 (t_0), the graph, made of one kernel node and one channel node, is initialized on the host; the CE is uniquely defined for that channel. At t_1, an `init` node (the host) puts source CEs in the channel node. At t_2, the kernel node consumes CEs from the channel node and produces new CEs. At t_3, the kernel node consumes the remaining CEs and the computation is done.

```
 1: #define LEN 32768
 2:
 3: typedef struct {
 4:   int val;
 5: } FibObj;
 6:
 7: void FibKernel(int srcID, int srcOff,
 8:                int destID, int *result) {
 9:   FibObj *src = (FibObj *)deq(srcID, srcOff);
10:   if(src->val <= 2) {
11:     atomic_add(result, 1);
12:   } else {
13:     FibObj *ob = (FibObj *)talloc(destID, 2);
14:     ob[0].val = src->val - 1;
15:     ob[1].val = src->val - 2;
16:     enq(destID, ob);
17:   }
18:   tfree(srcID, srcOff);
                                    19: }
20:
21: void main(int argc, char * argv[]) {
22:   int n = atoi(argv[1]);
23:   int res = 0;
24:
25:   Graph g;
26:   ChanNode *ch = g.ChanNode(sizeof(FibObj), LEN);
27:   KernelNode *kern = g.KernelNode(FibKernel);
28:   kern->setConstArg(2, sizeof(int), ch->chID);
29:   kern->setConstArg(3, sizeof(int *), &res);
30:   ch->connectToKernelNode(kern);
31:
32:   FibObj *ob = (FibObj *)ch->talloc(1);
33:   ob->val = n;
34:   ch->enq(ob);
35:
36:   g.execute();
37:   g.waitForDone();
38:   printf("fib(%d) = %d\n", n, res);
39: }
```

Figure 4-5. Fibonacci example.

```
1: int fib(int n) {
2:   if(n <= 2) return 1;
3:   else {
4:     int x = spawn fib(n - 1);
5:     int y = spawn fib(n - 2);
6:     sync;
7:     return (x + y);
8:   }
9: }
```

Figure 4-6. Fibonacci in Cilk.

A simple graph API was prototyped for this research. Figure 4-5 demonstrates how to build the channel-flow graph shown in Figure 4-4. A sophisticated flow-graph framework is beyond the scope of this work. The remainder of this chapter focuses on other aspects of our design.

4.3 Case Study: Mapping Cilk to Channels

Channels facilitate mapping higher-level abstractions to GPUs. As an example, we discuss translating a subset of the Cilk programming language to a channel representation.

4.3.1 Cilk Background

Cilk extends C/C++ for divide-and-conquer parallelism [47]. Cilk programs use the keyword `spawn` before a function to schedule it as a task. The keyword `sync` forces its caller to block until all of its spawned tasks are complete. Figure 4-6 demonstrates how these keywords are used to calculate the *n*th Fibonacci number. These two Cilk primitives form the basis of the language and are what we explore mapping to channels. Other primitives are left for future work.

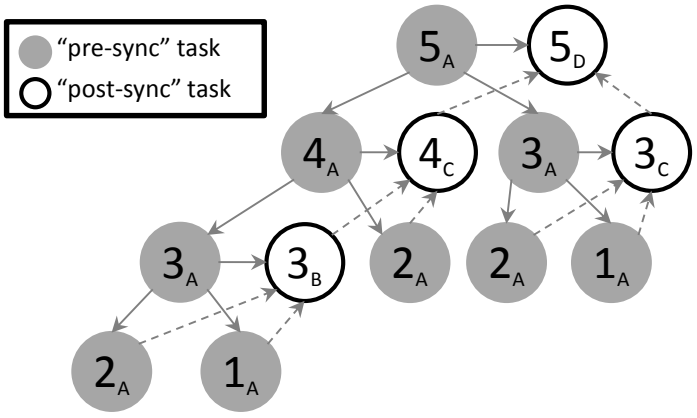


Figure 4-7. Cilk tree for Fibonacci.

4.3.2 Cilk as a Channel-flow Graph

One strategy to implement Cilk on channels is to divide kernels into sub-kernels that are scheduled respecting dependencies. Specifically, a sub-kernel is created whenever `sync` is encountered. Each `spawn` is translated into a `talloc`/`enq` sequence that reserves space in the correct channel, writes the task parameters, and schedules the work. Each `sync` is translated into a `talloc`/`enq` sequence that schedules work to a channel connected to the "post-sync" sub-kernel. It may be possible to automate these translations, but they are done manually for this research.

Figure 4-7 shows the Cilk tree to calculate the fifth Fibonacci number. Shaded circles are "pre-sync" tasks (lines 2-5 in Figure 4-6). White circles are "post-sync" tasks (line 7 in Figure 4-6). Solid lines depict task spawns and dotted lines are dependencies. Each circle is labeled with a letter specifying the order in which it can be scheduled.

Shaded circles, or pre-sync tasks, have no dependencies. They are labeled "A" and are scheduled first. White circles, or post-sync tasks, depend on shaded circles and other white circles. Dependencies on shaded circles are respected by scheduling white circles after all shaded circles are complete. White circles are labeled "B" or lexicographically larger. Dependencies among white circles are inferred conservatively from the level of recursion from which they derive. For example, the white circle representing the continuation for the fourth Fibonacci number and

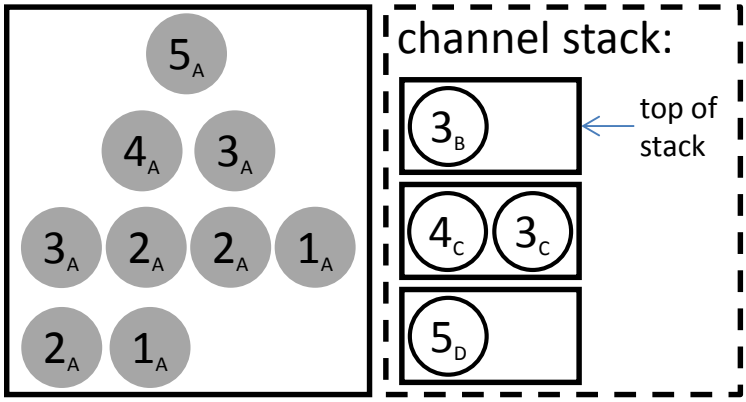


Figure 4-8. Managing dependencies with channels.

labeled “C” derives from the second level of the Cilk tree and depends on a continuation that derives from the third level.

Continuations that derive from deeper levels of the Cilk tree can be scheduled first. This is achieved by maintaining “stacks of channels” for continuations and scheduling each continuation at the correct offset within the stack. Virtual memory is allocated up front for channel stacks, similar to how CPU threads are allocated private stacks. Tasks determine the correct offset within the stack by accepting their recursion depth as a parameter. The scheduler drains the channel at the top of the stack before scheduling channels below it. This strategy is called levelization [48].

Figure 4-8 shows the tasks from Figure 4-7 organized into a main channel for pre-sync tasks and a stack of channels for post-spawn tasks. Figure 4-9 shows the channel-flow graph for the Cilk version of Fibonacci. Channel stack nodes (e.g., the dashed box in Figure 4-9) are added to the channel-flow-graph framework. Instead of atomically updating a global result, as is done by the flow graph in Figure 4-4, each thread updates a private result in the channel stack. Intermediate results are merged into a final result by a second continuation kernel node.

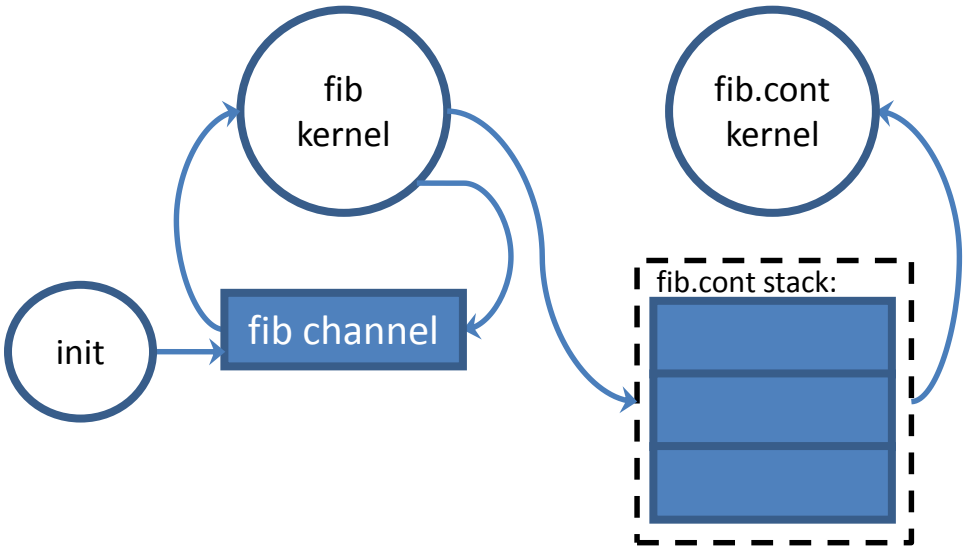


Figure 4-9. Channel-flow graph for Cilk version of Fibonacci.

Finally, it should be noted that the translations described for the Cilk version of Fibonacci generalize to other Cilk programs because they all have one logical recursion tree.

4.3.3 Bounding Cilk's Memory Footprint

For CPUs, Cilk runtimes use a work-first scheduling policy to bound the memory footprint to the depth of the Cilk tree. In work-first scheduling, threads traverse the Cilk tree in a depth-first manner by scheduling the continuation for a task that calls `spawn` and executing the spawned task [47]. This does not generate work fast enough for GPUs.

The scheduling policy described in Section 4.3.2 is called help-first. It generates work quickly by doing a breadth-first traversal of the Cilk tree, but consumes exponential memory relative to a workload's input size [37]. To make this policy feasible, the memory footprint must be bounded. This is possible if hardware supports yielding a CTX.

If a hardware context yields its execution resources when it is unable to obtain space in a channel, the scheduler can drain the channels by prioritizing work deeper in the recursion. When a base-case task is scheduled, it executes without spawning new tasks, freeing space in its channel. When a task near the base case executes, it spawns work deeper in the recursion. Because base-

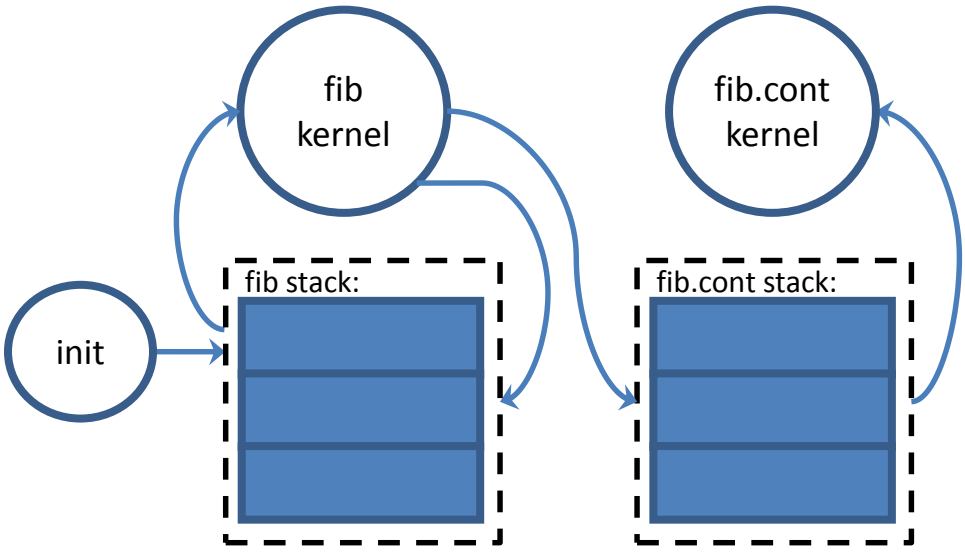


Figure 4-10. Bounding memory for the Cilk version of Fibonacci.

case tasks are guaranteed to free space, forward progress is guaranteed for the recursion prior to the base case. Inductively, forward progress is guaranteed for all channels.

The scheduler can differentiate work from different recursion levels if both pre- and post-sync tasks are organized into channel stacks, as shown in Figure 4-10. An alternative approach is a hybrid scheduler that uses help-first scheduling to generate work and then switches to work-first scheduling to bound memory [49]. Future work will compare a help-first only scheduler to a hybrid scheduler.

4.4 Wavefront Yield

To facilitate Cilk and similar recursive models, we propose that future GPUs provide a "wavefront yield" instruction. Our yield implementation, depicted in Figure 4-11, relies on the aggregator to manage yielded wavefronts. After a wavefront executes yield (❶), the GPU saves all of its state to memory (❷) including registers, program counters, execution masks, and NDRange identifiers. LDS is not saved because it is associated with the work-group and explicitly managed by the

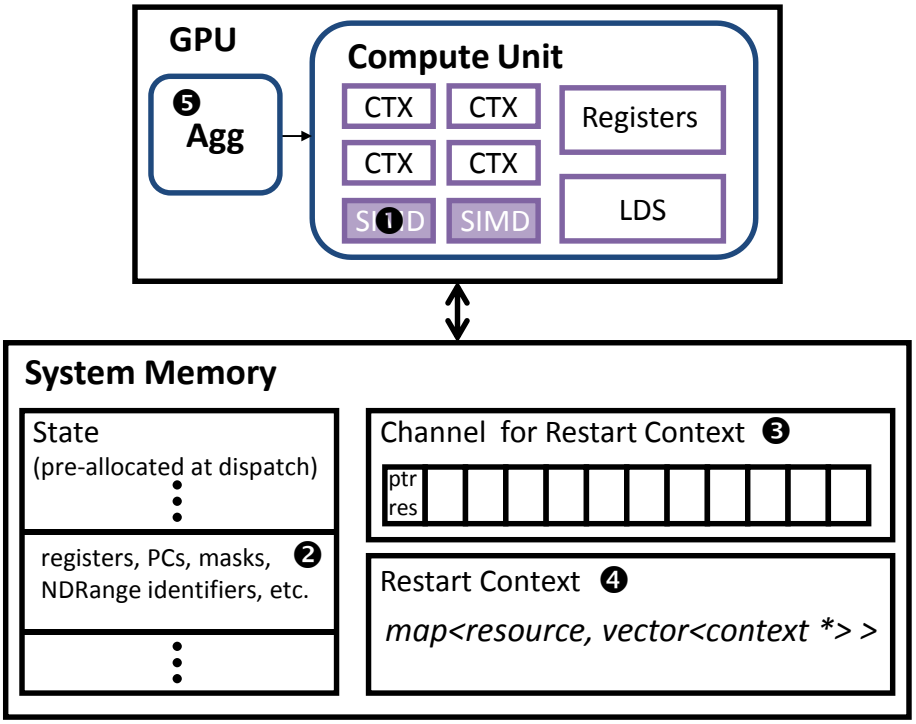


Figure 4-11. Wavefront yield sequence.

programmer; a restarting wavefront must be assigned to the same CU on which it was previously executing. Memory space for yield is allocated for each CTX before dispatch and deallocated as wavefronts complete. This is the same strategy used for spill memory in HSA.

In addition to the wavefront's state, a restart context, used to restart the wavefront, is saved to a data structure in memory (❸). This data structure can be a finite size because the aggregator will consume whatever is inserted into it; in our implementation, we use a channel. The restart context comprises a pointer to the wavefront's saved state and the resource that the wavefront blocked on. The aggregator retrieves the restart context and inserts it into a software-defined data structure that tracks blocked wavefronts (❹). The aggregator then schedules a new wavefront to occupy the yielded context (❺). The aggregator monitors resources and restarts wavefronts as appropriate.

4.5 Methodology and Workloads

We prototyped our channel implementation in the simulated system depicted in Figure 4-12. We used gem5 [50] enhanced with a proprietary GPU model. The GPU's control processor is implemented as a programmable core that serves as the aggregator. It is enhanced with private L1 caches that feed into the GPU's unified L2 cache. Each CU has a private L1 data cache that also feeds into the GPU's L2 cache. All CUs are serviced by a single L1 instruction cache connected to the GPU's L2 cache. More details can be found in Table 4-2.

To isolate the features required for channels, all caches are kept coherent through a read-for-ownership MOESI directory protocol [51] similar to the GPU coherence protocol proposed by Hechtman *et al*. [52]. Future work will evaluate channels with write-combining caches [53].

We implemented wavefront yield as described in Section 4.4. CTXs require, at a minimum, 856 bytes for program counters, execution masks, NDRange identifiers, etc. Additional bytes are required for registers. There are three kinds of registers: 64 x 4 byte (s), 64 x 8 byte (d), and 64 x 1-bit (c). The number of registers varies across kernels. We save all registers (live and dead). A more sophisticated implementation would avoid saving dead registers. The numbers of registers for our workloads are shown in Table 4-3. In the worst case (Queens), 9,072 bytes are saved/restored.

4.5.1 Workloads

We wrote four Cilk workloads derived manually from Cilk source according to the transformations discussed in Section 4.3. They are characterized in Table 4-3.

1. **Fibonacci:** Compute the *n*th Fibonacci number. Partial results are stored in continuation channels and merged by a continuation kernel.

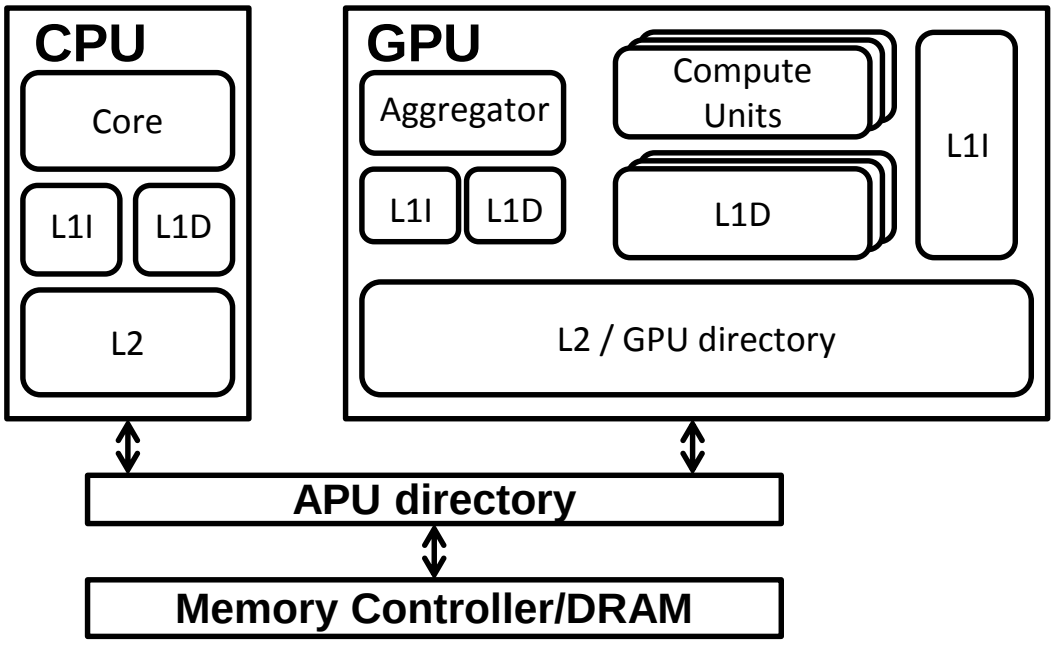


Figure 4-12. Simulated system.

2. **Queens:** Count the number of solutions to the NxN queens puzzle. In our implementation, derived from code distributed with the MIT Cilk runtime [47], the base case is a 4x4 sub-section of the chessboard.
3. **Sort:** Recursively split an array into four smaller sub-arrays until reaching a base case (64 elements), sort all of the base-case sub-arrays, and merge them. This workload also was derived from a version distributed with the MIT Cilk runtime [47].
4. **Strassen:** Repeatedly divide a matrix into four sub-matrices down to a base case (16x16 elements), multiply each pair of base-case matrices, and combine the results through atomic addition [54].

Table 4-2. Simulation configuration.
†See Section 4.6.3

Compute Unit	
Clock	1GHz, 4 SIMD units
Wavefronts (#/scheduler)	40 (each 64 lanes)/round-robin
Data cache	16kB, 64B line, 16-way, 4 cycles, delivers one line every cycle
Instr. cache (1 for all CUs)	32kB, 64B line, 8-way, 2 cycles
Aggregator	
Clock	2GHz, 2-way out-of-order core†
Data cache	16kB, 64B line, 16-way, 4 cycles
Instr. cache	32kB, 64B line, 8-way, 2 cycles
Memory Hierarchy	
GPU L2/directory	1MB, 64B line, 16-way, 16 cycles
DRAM	1GB, 30ns, 20GB/s
Coherence protocol	MOESI directory
Host CPU (not active in region of interest)	
Clock	1GHz, gem5 TimingSimpleCPU
L1D, L1I, L2 (size/assoc/latency)	64B lines across all caches (64kB/2/2), (32kB/2/2), (2MB/2/2)
Channel	
Done count	64

4.5.2 Scheduler

A scheduling algorithm, which executes on the aggregator, was written for the Cilk workloads. It respects Cilk's dependencies by tracking the current level in the recursion, as described in Section 4.3.2. It also checks for wavefronts that have yielded and restarts them as resources (i.e., channels deeper in the recursion) become available. Because levelization enforces dependencies, the GPU can block on the scheduler. We explore workload sensitivity to the aggregator in Section 4.6.3.

4.6 Results

We find that three of our four Cilk workloads scale with the GPU architecture, and show details in Figure 4-13. The average task size (average cycles/wavefront) for each workload, shown in Table 4-3, and cache behavior, depicted in Figure 4-14, help explain the trends.

First, we examine the workload that does not scale up to eight CUs: Fibonacci. Given the small amount of work in its kernel nodes, we would not expect Fibonacci to scale. Even so, it is useful for measuring the overheads of the channel APIs because it almost exclusively moves CEs through channels. A consequence of Fibonacci's small task size is that it incurs more GPU stalls waiting on the aggregator than workloads with larger task sizes. Larger task sizes allow the aggregator to make progress while the GPU is doing compute. At eight CUs, Fibonacci's cache-

Table 4-3. Workloads.
†Section 4.6.2 ††Measured from a one-wavefront execution (channel width=64, input=largest with no yields)

Workload	Data set	Kernel nodes	Registers/kernel	Channel width†	# of wavefronts	Average cycles/wavefront††
Fibonacci	24	2	16s/8d/2c, 3s/4d/1c	32,768	2,192	7,046
Queens	13x13	2	16s/8d/3c, 5s/5d/1c	16,384	1,114	35,407
Sort	1,000,000	4	16s/8d/2c (all 4 kernels)	32,768	4,238	30,673
Strassen	512x512	1	16s/6d/8c	8,192	587	259,299

to-cache transfers degrade performance; these occur because consumer threads execute on different CUs than their respective producer.

The other three Cilk workloads scale well from one CU to eight CUs, with speedups ranging from 2.6x for Sort to 4.3x for Strassen. This is because the workloads perform non-trivial amounts of processing on each CE, which is reflected in the average task size. Strassen's wavefronts are approximately 37 times larger than Fibonacci's. In contrast, Sort's wavefronts are a little more than four times larger than Fibonacci's, indicating that relatively small tasks can be coordinated through channels to take advantage of the GPU.

While few memory accesses hit in the L1 cache, many hit in the shared L2, facilitating efficient communication between producers and consumers. L2 cache misses degrade scalability because main memory bandwidth is much lower than cache bandwidth. As illustrated in Figure 4-13, the aggregator overhead is constant with respect to the number of CUs, so we would not expect it to be a bottleneck for larger inputs.

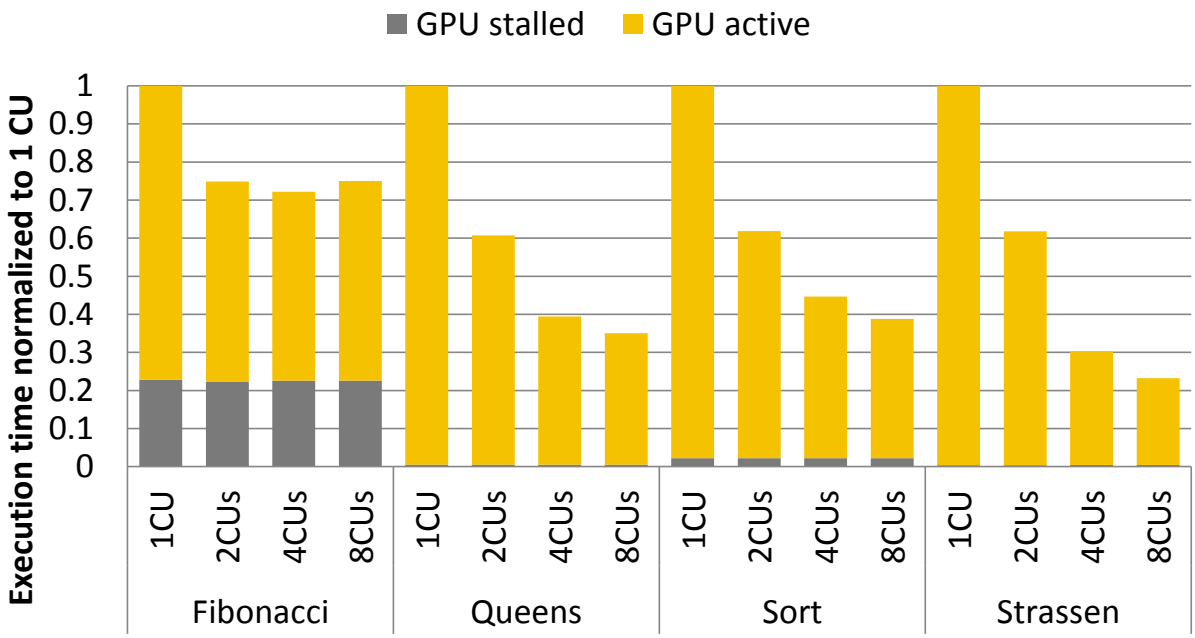


Figure 4-13. Scalability of Cilk workloads.

Table 4-4. GPU Cilk vs. conventional GPGPU workloads.

	LOC reduction	Dispatch rate	Speedup
Strassen	42%	13x	1.06
Queens	36%	12.5x	0.98

To help put these results in context, we compare channel workloads to non-channel workloads when possible. Specifically, we compare Strassen to matrix multiply from the AMD SDK [55] and Queens to a version of the algorithm distributed with GPGPU-Sim [56]. We would expect channels to be slower than conventional GPGPU code because their fine-grain nature leads to more tasks, which imposes extra coordination overhead; both channel codes trigger more than 10 times the number of dispatches than their non-channel counterparts. Surprisingly, we find that channels are on par with conventional GPGPU code because they facilitate more efficient algorithms. Specifically, Strassen has a lower theoretical complexity than AMD SDK's matrix multiply. Meanwhile, for Queens the GPGPU-Sim version pays large overheads to flatten a recursive algorithm that is expressed naturally through channels. Both Strassen and Queens have fewer lines of code (LOC) than the non-channel versions. These results are summarized in Table 4-4.

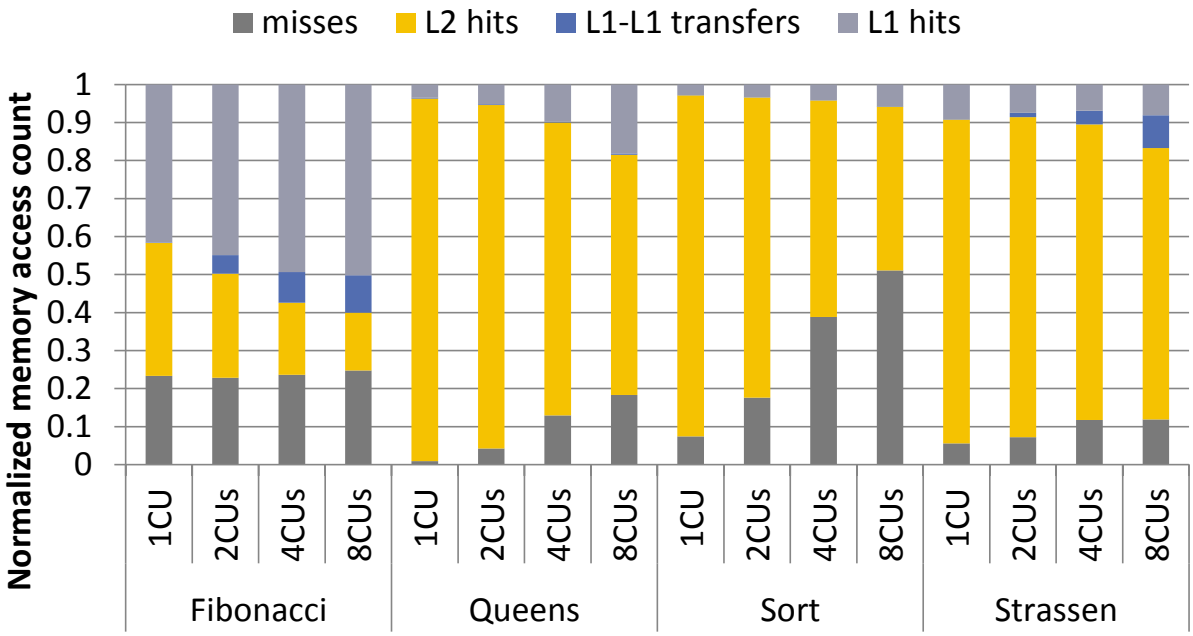


Figure 4-14. CU cache behavior.

4.6.1 Array-based Design

Figure 4-15, which compares the baseline channel to a "GPU-efficient channel" that has the intra-wavefront optimizations suggested in Section 4.1.2 (i.e., Figure 4-3), shows the effectiveness of amortizing synchronization across the wavefront. By reducing the number of `CAS` operations (where `talloc` and `enq` spend most of their time) by up to 64x, this optimization reduces the run-time drastically for all workloads.

We compared the GPU-efficient channel to a version that is padded such that no two CEs share a cache line. Padding emulates a linked list, which is not likely to organize CEs consumed by adjacent work-items in the same cache line. In all cases, the padded channel performs worse, but the degradation is less than expected because the CEs are organized as an array of structures instead of a structure of arrays. Our indirect aggregation design, described in Chapter 5, does not have this issue.

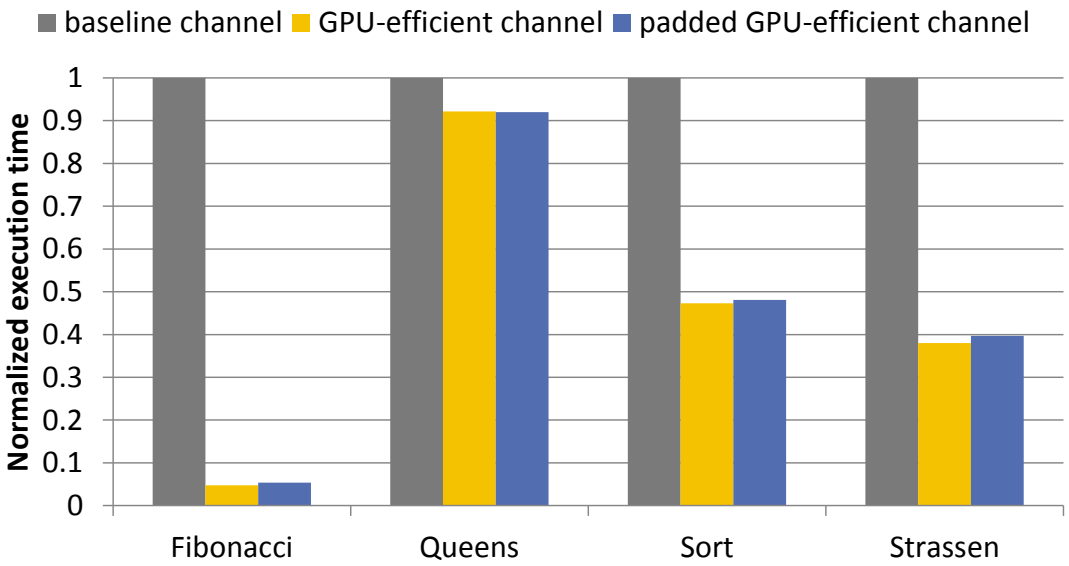


Figure 4-15. GPU-efficient array quantified.

4.6.2 Channel Granularity

Done Count

Channel space is deallocated in order at the granularity of the done count. A done count of 64 limits deallocation to less than 3% of total stall time on average.

Channel Width

Figure 4-16 shows how channel width can affect performance. Narrow channels are unable to supply enough CEs to utilize the GPU adequately. Meanwhile, larger channels degrade the performance of Strassen because wavefronts are not able to use the L2 cache as effectively. We configured each workload with the channel width that resulted in peak performance (shown in Table 4-3). Better cache-management policies, like cache-conscious thread scheduling [57], may eliminate the cache thrashing caused by wider channels.

Wavefront Yield

Figure 4-16 also shows the frequency and impact of yields. Saving and restoring CTXs generally has little impact on GPU active time because yields are relatively infrequent. However, at smaller

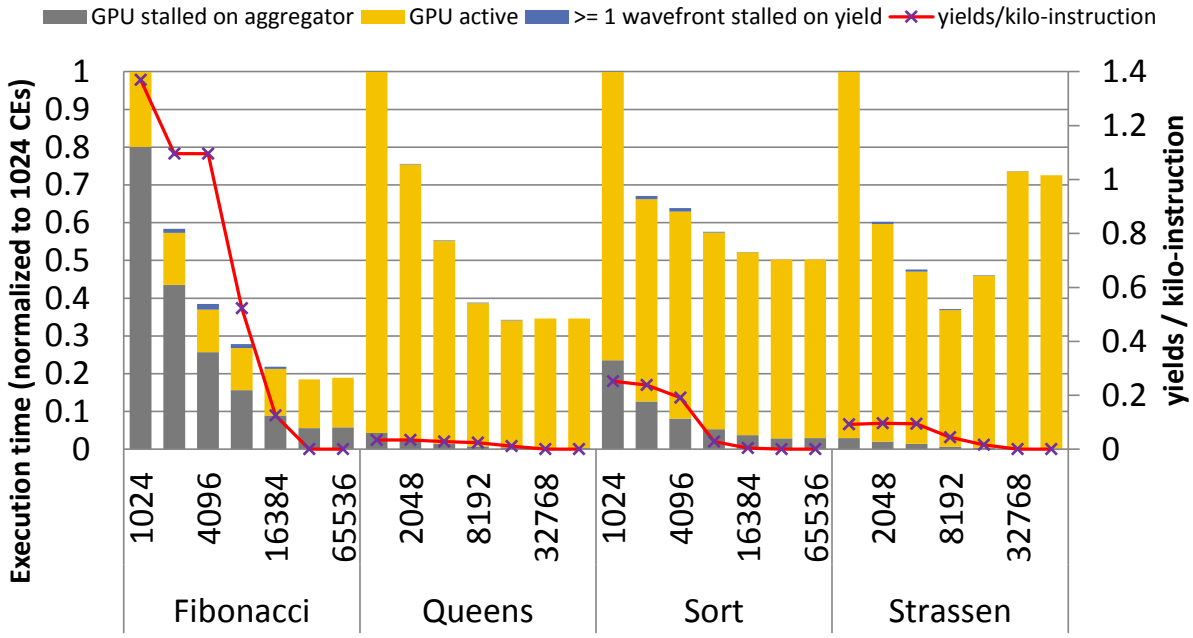


Figure 4-16. Channel width (CEs).

channel widths, frequent yields increase GPU stall time because the aggregator manages yields instead of dispatching new work.

4.6.3 Aggregator Sensitivity Study

We performed a sensitivity study to determine how complex the aggregator needs to be. The first design that we considered is a primitive core, called *simple*, which is not pipelined and executes one instruction at a time; this extremely slow design increases pressure on the aggregator. We also considered a complex out-of-order (OoO) core, called *4-way OoO*, to capture the other extreme. Finally, we looked at two intermediate designs: *2-way OoO* and *2-way light OoO*. *2-way OoO* resembles a low-power CPU on the market today. *2-way light OoO* is derived by drastically slimming *2-way OoO* and provides insight into how an even simpler core might perform. Table 4-5 summarizes our findings. *4-way OoO* provides little benefit relative to *2-way OoO*. *2-way light OoO* reduces the performance gap between *simple* and *2-way OoO*, but the aggregator overhead can still be as high as 35%. Hence, *2-way OoO* strikes a good balance between performance and core complexity and was used to generate the results reported in previous sections.

Table 4-5. Time (%) GPU (8 CUs) is blocked on aggregator.

	Description	Fibonacci	Queens	Sort	Strassen
Simple	*no pipelining, one instruction at a time*	41.5	2.9	8.9	3.4
2-way light OoO	*physical registers: 64, IQ size: 2, ROB size: 8, ld/st queue size: 8/8*	35.1	2.0	7.2	2.5
2-way OoO	*physical registers: 64, IQ size: 32, ROB size: 64, ld/st queue size: 32/32*	30.1	1.6	5.8	1.8
4-way OoO	*physical registers: 128, IQ size: 64 ROB size: 128, ld/st queue size: 64/64*	29.8	1.5	5.6	1.9

4.6.4 Divergence and Channels

Figure 4-17 depicts branch divergence. Fibonacci and Queens have many wavefronts with base-case and non-base-case threads, leading to high divergence. Strassen has little divergence because it distributes work very evenly. Sort, which spends most of its time in the base case, suffers severe divergence. This is because the base-case code was obtained from a CPU version that uses branches liberally.

4.7 Summary

Channels aggregate fine-grain work into coarser-grain tasks that run efficiently on GPUs. This section summarizes our work on channels, discusses its implications, and anticipates future work.

We proposed the first channel implementation. While our design scales to eight compute units, there are several improvements that future work should consider. Our implementation is a flat queue, but a hierarchical design may scale even better. We also used a read-for-ownership coherence protocol in our evaluation, but future work should quantify the effects of write-

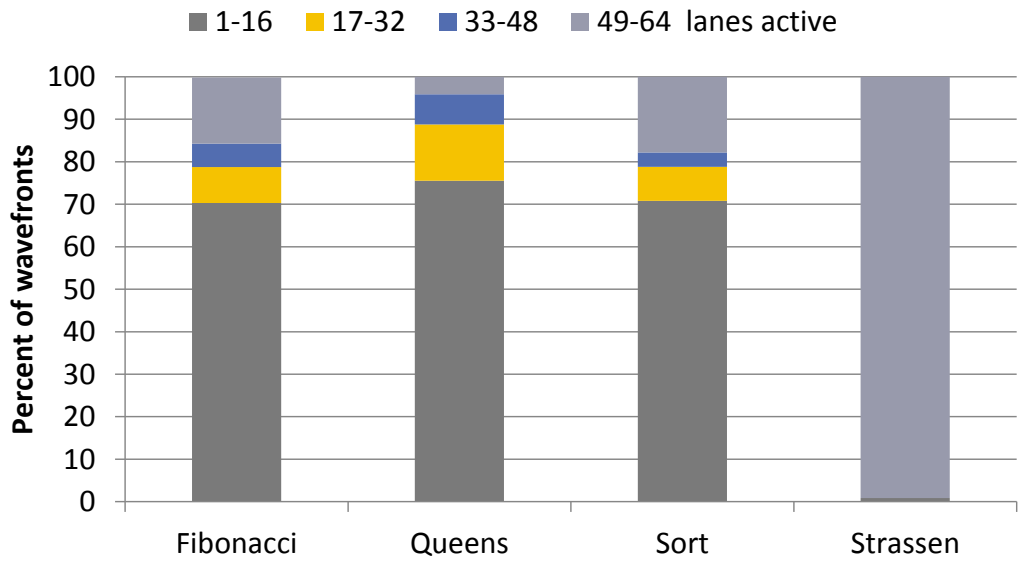


Figure 4-17. Branch divergence.

combining caches. Finally, future designs should optimize the layout of CEs in memory for SIMT hardware.

We described a set of transformations to map Cilk to a channel-flow graph. Future work should investigate mapping other high-level languages to GPUs through channels. Our implementation of Cilk on top of channels hard-codes both Cilk's dependencies and the subset of channels from which to schedule in the aggregator's firmware. Future work should explore general abstractions for managing the channels and their dependencies. For example, it may be possible to apply the concept of guarded actions to channels [58].

We used the GPU's control processor, which we called the aggregator, to manage channels and restart yielded wavefronts. We found that its architecture had a crucial impact on the performance of channels. While the out-of-order design that we used worked well for our workloads, a more efficient design might achieve similar results. Future work should explore control processor architectures that enable other novel GPGPU programming models and abstractions.

5. GRAVEL: FINE-GRAIN GPU-INITIATED NETWORK MESSAGES

Despite the fact that GPUs are becoming increasingly prominent in distributed systems, it is surprisingly difficult to route a network message between a work-item on the GPU and the network. This is because accessing the network is done through system-level operations. Specifically, network operations themselves are system-level operations that must be coordinated through the network interface (NI).

Thus, the goal of this chapter is to apply the operation-per-lane model to GPU-initiated network operations. To achieve this goal, we introduce Gravel—an indirect aggregation layer implemented entirely in software. In Gravel, GPU-initiated network messages are routed through a GPU-efficient producer/consumer queue to an aggregator, which combines messages being sent to the same destination.

Gravel amortizes synchronization across adjacent work-items in a work-group. Note, this is different than our channels prototype, which amortizes synchronization across a wavefront (not a work-group). As we show later in this chapter, work-groups are better at amortizing the overhead of producer/consumer synchronization because they have more work-items than a wavefront. But work-group-level synchronization can be ambiguous inside of a branch because GPUs execute wavefronts, not work-groups. To solve this challenge, Gravel leverages a *diverged work-group-level semantic* to enable the GPU to access the network from divergent code.

Notably, Gravel works on real hardware. This is in contrast to our channels prototype described in Chapter 4, which was confined to a simulator. To make Gravel work on current GPUs, the aggregator is implemented with CPU threads and software predication is used to achieve the diverged work-group-level semantic. We believe that future GPUs can support these features more

efficiently in hardware. For example, we suggest and evaluate two alternatives to software predication—a work-group-wide reconvergence stack and fine-grain barriers.

A second constraint is that Gravel is confined to integrated GPUs. This is because we leverage fine-grain shared virtual memory (SVM) [32], which is a feature where the CPU and GPU synchronize through atomic memory operations. Several integrated GPUs, such as AMD's HSA-compatible GPUs [59] and Intel's Graphics Gen9 [31], support this feature. In contrast, discrete GPUs currently lack this capability.

We evaluate our prototype Gravel implementation on a cluster of eight AMD APUs. Compared to one node, Gravel achieves a 5.3x speedup on average across six irregular applications. Furthermore, we show that Gravel is more productive and usually more performant than the coprocessor model and coalesced APIs.

5.1 Gravel Overview

In Gravel (Figure 5-1), GPU-initiated messages are routed through a GPU-efficient producer/consumer queue to an aggregator, which repacks the messages into aggregation buffers, which we call per-node queues in this chapter. The aggregator sends a per-node queue to the network interface (NI) after it becomes full or exceeds a timeout. The producer/consumer queue

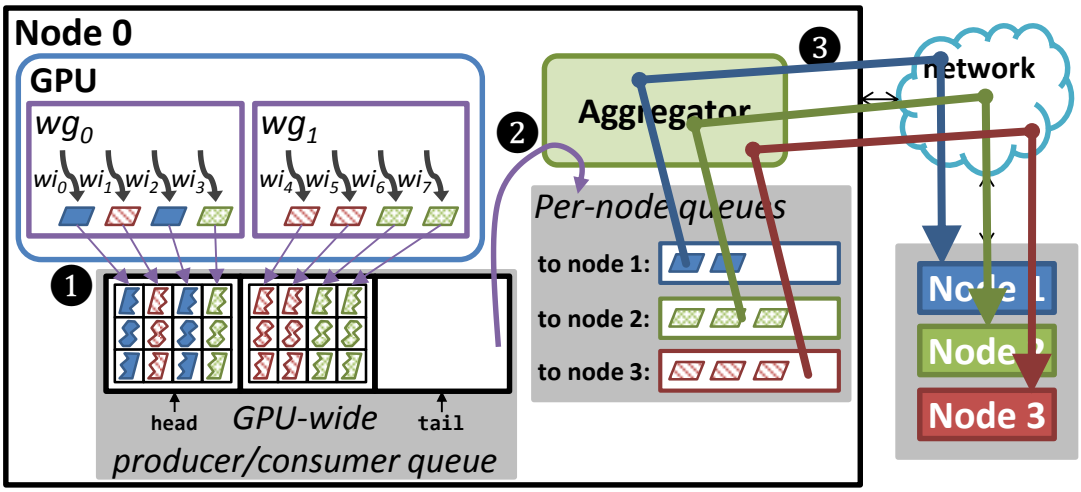


Figure 5-1. GPU-initiated network message flow in Gravel.

interface (Section 5.2.2) hides low-level issues like avoiding deadlock between work-items in a wavefront or optimizing SIMT utilization. Thus, Gravel adheres to the operation-per-lane model, but Gravel performs well for two reasons.

First, like the coprocessor model, Gravel's aggregator generates large messages to amortize network overhead. Second, Gravel amortizes synchronization across work-groups, which is similar to coalesced APIs—but Gravel does not require work-items to operate in a work-group synchronous fashion. Instead, we leverage a diverged work-group-level semantic to asynchronously offload messages to the NI (Section 5.3). Another alternative is to offload messages at wavefont granularity, which is done in prior work like GGAS [18] and is also the approach taken in our prototype of channels in Chapter 4. We find that offloading messages at work-group granularity is 3.2x faster (Section 5.2.1). A caveat is that it can be difficult to offload messages at work-group granularity from divergent code, as we discuss later in this chapter.

One last subtle point is that Gravel's indirect strategy scales better than SIMT-direct aggregation. Specifically, as the number of destinations (and per-node queues) increase, SIMT-direct aggregation suffers low SIMT utilization because work-items in the same work-group write different queues. In contrast, Gravel ensures that the GPU always writes messages to a single queue (i.e., the producer/consumer queue).

5.2 Gravel's Producer/consumer Queue

We now describe Gravel's producer/consumer queue, which acts as the GPU's interface to Gravel's aggregator. The queue differs from CPU queues in two important ways. First, it handles SIMT correctness and performance issues that occur when exporting messages from the GPU's data-parallel hardware to the NI. Second, the queue limits the frequency of shared-memory synchronization, which is required to coordinate work-items initiating messages in parallel.

First, Section 5.2.1 explains how work-group-level synchronization enables the GPU to export messages at work-group granularity. Next, Section 5.2.2 details the producer-consumer synchronization algorithm used to order work-items and aggregator threads accessing the queue. Finally, Section 5.2.3 quantifies the queue's performance.

5.2.1 Work-group-level Synchronization

The GPU interacts with the aggregator through in-memory queues. For example, to send a message, a work-item reserves space in a queue, writes the message (e.g., command, payload) into the queue, and notifies the NI that the message is ready to be sent. Thus, producer/consumer synchronization is required to reserve space and again to notify the NI.

Ignoring (for now) the case where the queue is full, a work-item can reserve space by using a `fetch-add` to atomically increment the queue's write index. In this approach, depicted in Figure 5-2 (scenes *a* and *c*), shared-memory synchronization occurs at work-item granularity. An

```
1: work_item_level_reserve(Q):
2:   return fetch_add(Q.WrIdx, 1);
```

```
sample run:
ret: [2,3,4,5]
```

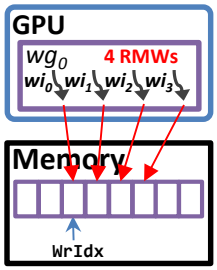
(a) work-item-level synchronization pseudo-code

```
 3: work_group_level_reserve(Q):
 4:   lid = LANE_ID; # wi's WG offset
 5:   max = reduce_max(lid);
 6:   MyOff = prefix_sum(1);
 7:   Qi = 0;
 8:   if lid == max:
 9:     Qi=fetch_add(Q.WrIdx,MyOff+1);
10:   return reduce_sum(Qi)+MyOff;
```

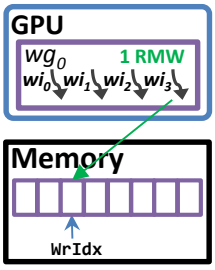
```
sample run:
lid:  [0,1,2,3]
max:  [3,3,3,3]
MyOff:[0,1,2,3]
Qi:   [0,0,0,0]

Qi:   [0,0,0,2]
ret:  [2,3,4,5]
```

(b) work-group-level synchronization pseudo-code



(c) work-item-level behavior



(d) work-group-level behavior

Figure 5-2. Work-item vs. work-group-level synchronization.

alternative, shown in Figure 5-2 (scenes *b* and *d*), is to leverage SIMT execution so that a leader work-item synchronizes globally on behalf of its work-group. Figure 5-2b shows that this can be achieved using a few work-group-level operations. Specifically, the leader work-item is chosen to be the work-item with the largest lane ID using a `reduce-max` operation (lines 4-5). Then, a prefix-sum operation is used to determine each work-item's local offset (line 6); inactive work-items can cause the local offset to differ from the lane ID. Next, the leader work-item reserves a slot for each work-item (lines 7-9). Finally, the leader work-item broadcasts the work-group's queue offset, which is added to each work-item's local offset (line 10).

Figure 5-3 shows how work-group size impacts the throughput of Gravel's producer/consumer queue (Section 5.2.2) for 32-byte messages; details about the processor are in Section 5.4. Larger work-groups achieve greater throughput by amortizing atomic operations across more work-items. For example, a work-group with four wavefronts achieves more than 3x throughput than a work-group with a single wavefront by reducing the number of atomic

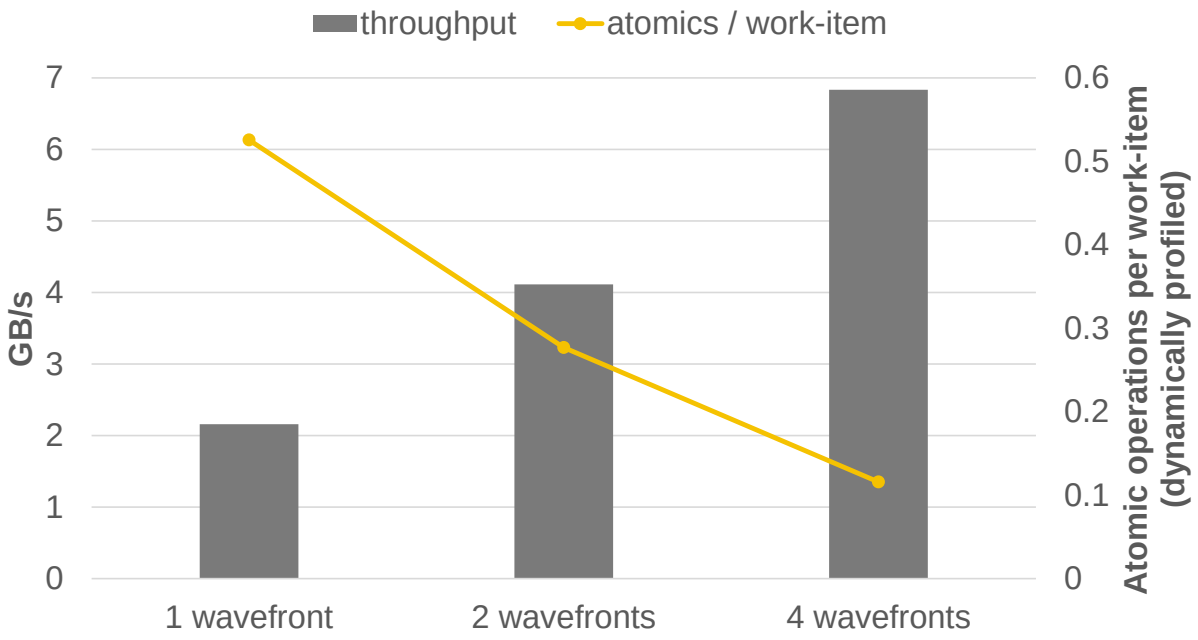


Figure 5-3. Producer/consumer throughput vs. work-group size.

operations by almost 80%. We also measured the throughput of Gravel's producer/consumer queue implemented with work-item-level synchronization and found that it is two orders of magnitude slower (0.06 GB/s).

One issue is that work-group-level synchronization requires all of the work-items in a work-group to participate. As a result, Gravel requires explicit software predication to leverage work-group-level synchronization from divergent code. Section 5.3 discusses this issue in detail and explores diverged work-group-level operations an alternative for future GPUs.

5.2.2 Producer/consumer Behavior

The producer/consumer queue's design and operation is illustrated in Figure 5-4. Each queue slot is arranged as a two dimensional array, where each column holds a work-item's message. This organization enables messages to be written in a non-divergent manner. In our implementation, the first row is used to store the command (e.g., `PUT`, atomic increment), the second row stores the destination, and subsequent rows encode arguments (e.g., address, value).

In addition to the payload, each queue slot has variables to synchronize producers (i.e., work-items) and consumers (i.e., aggregator threads) and avoid overflowing the queue. To obtain an offset into the queue, `fetch-add` is used to increment `WriteIdx` (by producers) and `ReadIdx` (by consumers). Three situations require synchronization. The first occurs when two or more producers alias to the same array slot. A ticket lock, `WriteTick`, is used to synchronize producers. The second situation occurs when two or more consumers alias to the same array slot. A second ticket lock, `ReadTick`, is used to synchronize consumers. Finally, a full/empty bit, `F`, is used to arbitrate between a producer that has the write ticket and a consumer that has the read ticket.

Figure 5-4, which focuses on the messages initiated by wg_0, demonstrates the queue's operation. Initially (time ❶), the queue, which has three slots, is empty. At time ❷, wi_3 obtains a write ticket of 0 after performing a `fetch-add` operation on `WriteTick`. Because the write ticket equals the current ticket, `N`, and the full bit, `F`, is clear, wi_3's work-group owns the slot. All four work-items (i.e., wi_0-wi_3) write their messages into the slot and wi_3 sets the full bit, `F`, at time ❸. At time ❹, an aggregator thread, t_0, takes ownership of the slot because the full bit, `F`, is set and its read ticket equals the slot's current ticket, `N`. Finally, after the aggregator has consumed the messages, it clears the full bit, `F`, and increments the current ticket, `N`, to release the slot (time ❺).

5.2.3 Producer/consumer Queue Analysis

Figure 5-5 shows the throughput of Gravel's producer/consumer queue at different message sizes; work-groups have four wavefronts. The left side of the figure corresponds to small messages (e.g.,

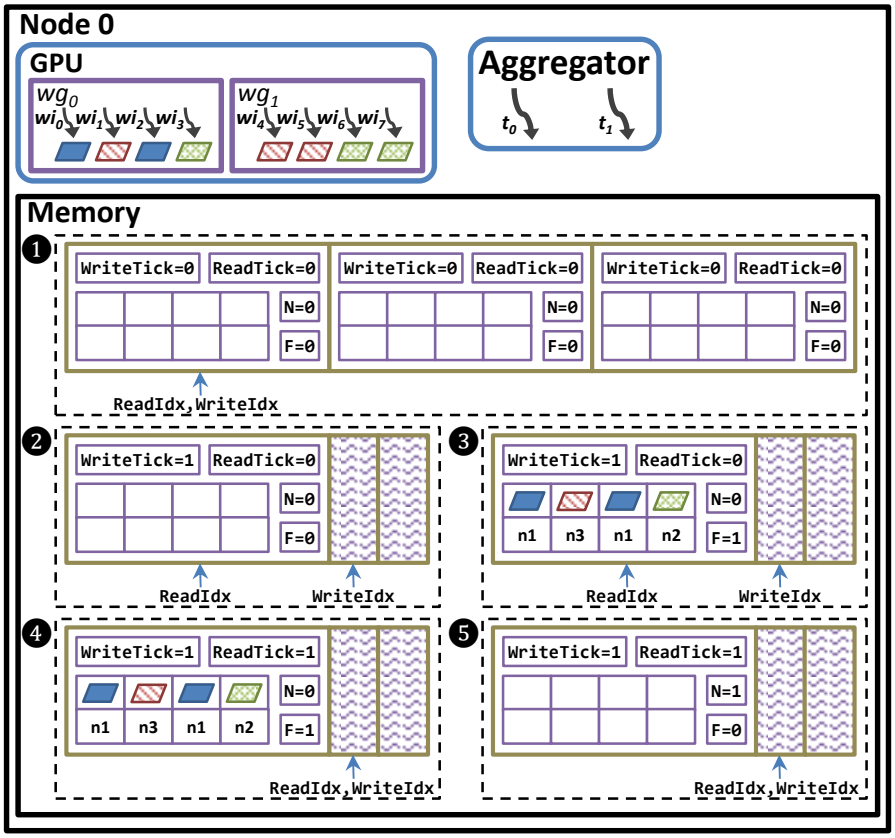


Figure 5-4. Gravel's producer/consumer behavior.

smaller than a cache line), which incur large overhead. The right side corresponds to larger messages that can be managed using traditional synchronization approaches. The plot demonstrates that Gravel's producer/consumer queue achieves high throughput for small messages. For example, 32-byte messages are processed at 7 GB/s, which matches the network bandwidth in our system (Section 5.4).

To put Gravel's performance into perspective, the plot shows two additional producer/consumer queues, where all producers and consumers are CPU threads. The first is a simple single-producer/single-consumer (SPSC) queue [60]. The second is a multi-producer/multi-consumer (MPMC) queue, which uses the same synchronization algorithm as Gravel. The only difference is that each queue slot is organized to be written by a single CPU thread instead of a GPU work-group.

Two factors enable Gravel to offload small messages faster than the CPU-only queues. The first is work-group-level synchronization, which amortizes producer/consumer synchronization across a work-group—up to 256 messages in our system. In contrast, the other queues require

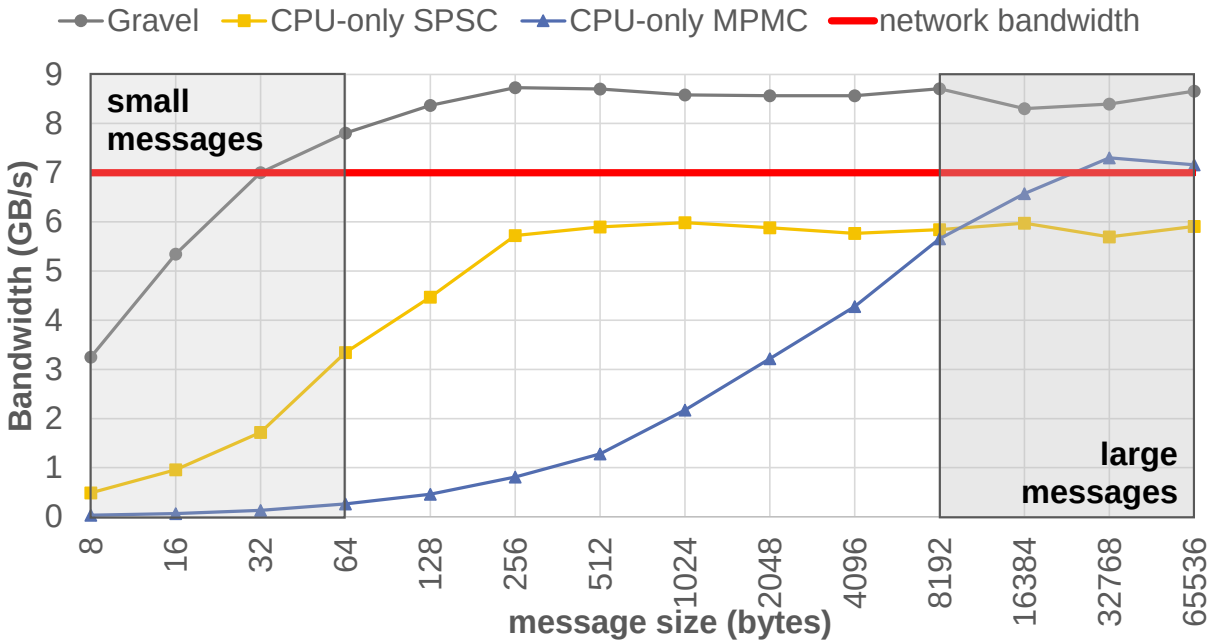


Figure 5-5. Producer/consumer queue throughput.

producer/consumer synchronization for each message. The second factor is the payload organization, which allows the work-items in a work-group to write messages into the same cache lines. This is possible because work-items in the same work-group execute on the same CU. Conversely, extra bytes are appended to the payload in the CPU-only designs to avoid false sharing and this padding adds significant overhead for small messages. For example, in the SPSC queue, three cache lines are read/written to send an eight-byte message—a padded read index, a padded write index, and the padded payload. Things are worse for the MPMC queue. In contrast, Gravel's queue incurs a half-byte of overhead to send the same eight-byte message.

The performance of large messages, which is not the focus of this paper, is explained by how each queue uses the evaluated CPU, which is four-way threaded. The MPMC queue is configured with two producer threads and two consumer threads. Gravel's queue uses all four CPU threads as consumers. Thus, in the limit, Gravel is limited by the throughput of its four consumer threads, the MPMC approaches the throughput of two threads, and the SPSC approaches the throughput of a single thread.

5.3 Diverged Work-group-level Semantic

Earlier, we described work-group-level synchronization (Section 5.2.1) and showed that it helps to amortize synchronization (Figure 5-3). We also noted that software predication is required to leverage work-group-level synchronization from divergent code because work-group-level operations must occur within converged control flow [32].

In this section, we first provide an example that requires network access from diverged control flow, then show how software predication enables the example to work on current GPUs (Section 5.3.1). Next, we define useful behavior for work-group-level operations that occur in diverged

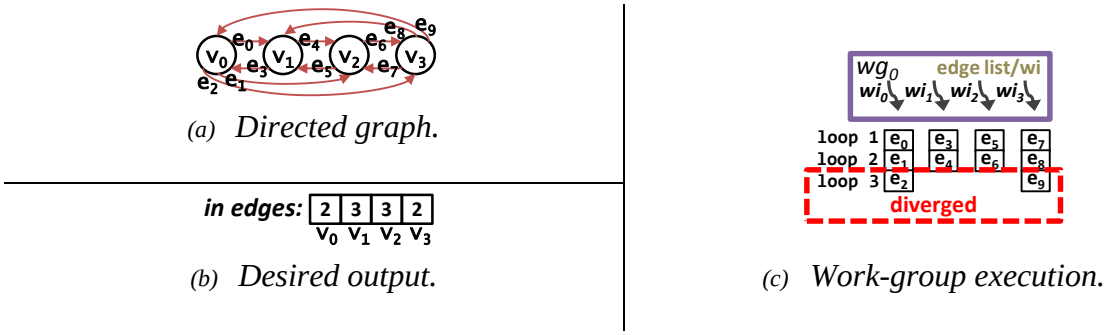
control flow (Section 5.3.2). Finally, we describe how future GPUs can provide this behavior (Section 5.3.3).

5.3.1 Software Predication

To understand how the current behavior of work-group-level operations limits Gravel's networking capability (and the operation-per-lane model more generally), consider the example in Figure 5-6, which counts the number of incoming edges for each vertex in a directed graph. For instance, in Figure 5-6a, v_0 has two incoming edges—one from v_1 and a second from v_3. In general, this problem can be solved by traversing each vertex's outgoing edge list and incrementing a counter once for each neighbor encountered. Figure 5-6b shows the final counters for the graph in Figure 5-6a.

Figure 5-6c shows one way to distribute this problem using Gravel. Each GPU work-item traverses a vertex's outgoing edge list. Table 5-1a shows pseudo-code; each work-item loops through its edge list and uses a network operation, `shmem_inc`, to update a distributed array of counters. Figure 5-6c shows that all of the work-items are active during the first two loops. In the third loop, wi_1 and wi_2 become inactive, which prevents wi_0 and wi_3 from leveraging work-group-level synchronization to access the network.

Table 5-1b shows how software predication solves this problem. In the code, inactive work-items keep executing with their work-group. Specifically, before entering the loop, work-items



(a) Directed graph.

(b) Desired output.

(c) Work-group execution.

Figure 5-6. Using WG-level operations in diverged control flow.

coordinate to determine the number of loop iterations to execute (line 5). Inside the loop, work-items determine whether they are active (line 7) and if so they construct a network message (lines 9-11). Finally, the network API is extended with an extra argument to differentiate active and inactive work-items (line 12).

Software predication enables work-group-level synchronization in divergent code, but it requires a non-trivial code transformation and introduces software overhead. Next, we articulate software predication's behavior and then consider alternatives to achieve that behavior.

5.3.2 Defining Useful Behavior

This section proposes that work-group-level operations occurring in diverged code execute across the *active* work-items in a work-group. Specifically, a work-item is active if it is predicated on when its work-group executes a basic block. Note, our proposal requires a way for the GPU to view the application's control flow at work-group granularity (instead of wavefront granularity)

Table 5-1. Diverged work-group-level operation pseudo-code.

```
 1: count_in_edges(edge_list, visitors):
 2:   for each edge in edge_list:
 3:     shmem_inc(&visitors[edge.idx], edge.node)
```

(a) Ideal pseudo-code to count each vertex's in edges.

```
 4: count_in_edges(edge_list, visitors):
 5:   loop_cnt = reduce_max(edge_list.size)
 6:   for i in range(loop_cnt):
 7:     active = i < edge_list.size
 8:     idx = 0, node = 0
 9:     if active:
10:       idx = edge_list[i].idx
11:       node = edge_list[i].node
12:     shmem_inc(&visitors[idx], node, active)
```

(b) Pseudo-code modified to use software predication.

```
13: count_in_edges(edge_list, visitors):
14:   if LANE_ID == 0:
15:     initfbar fb # create fine-grain barrier object
16:   joinfbar fb # start with all work-items
17:   for each edge in edge_list:
18:     shmem_inc(&visitors[edge.idx], edge.node, fb)
19:     if edge + 1 == edge_list.end:
20:       leavefbar fb
```

(c) Pseudo-code modified to use fine-grain barriers.

so that it is clear which work-items participate in a given work-group-level operation. Below, we describe two useful diverged work-group-level operations.

1. **Reduction:** Active work-items submit a value. Inactive work-items submit a *non-interfering value* (e.g., `0` for reduce-to-sum, `INT_MAX` for reduce-to-minimum). The reduction of these values is returned to active work-items.
2. **Prefix Sum:** Active work-items submit a value. Inactive work-items submit the non-interfering value `0`. The prefix sum of these values is returned to active work-items.

More generally, non-interfering values are used to implement data-parallel operations. For example, a work-group-level sort might be defined such that inactive work-items submit `INT_MAX`, which will be placed at the end of the sorted list, where it can be ignored by the active work-items.

5.3.3 Supporting Diverged Work-group-level Operations

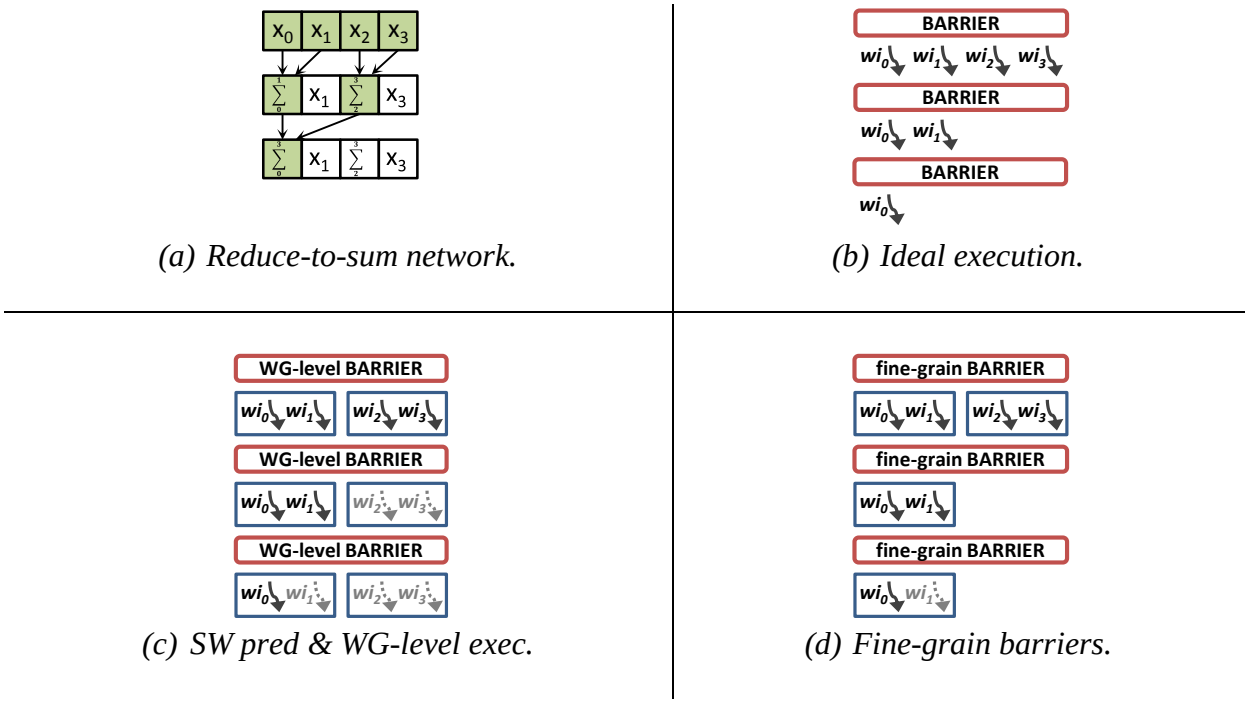
Work-group-level operations use a work-group's work-items to index an array and route the respective elements through a data-parallel network. For example, Figure 5-7a shows a reduce-to-sum network with four elements and Figure 5-7b shows an ideal execution with four work-items. Note that all work-items must be present to submit their values. Subsequent levels of the network, which are executed by the work-items that submitted values, are separated by a barrier.

In a diverged work-group-level operation the GPU must determine which wavefronts have active work-items and wait for those wavefronts to arrive. This is non-trivial because wavefronts in the same work-group progress through the control flow graph at different rates. Next, we discuss three ways to determine the wavefronts with active work-items.

First, it may be possible to automate the code translation for software predication. One issue with software predication is that it can cause a completely inactive wavefront to continue executing, as depicted in Figure 5-7c because it builds off of work-group-level operations. Another

approach is to build GPUs that track control flow at work-group granularity instead of wavefront granularity. For example, thread block compaction, proposed to mitigate branch divergence, suggests a work-group-level reconvergence stack [16]. Compared to software predication, this approach does not add software overhead, but it does allow inactive wavefronts, as depicted in Figure 5-7c because it essentially expands the GPU's execution granularity to the width of a work-group.

Finally, fine-grain barriers (`fbar`), introduced by HSA, can be used to identify active work-items [59]. An `fbar` enables barrier synchronization across a subset of a work-group's work-items. Specifically, HSA provides primitives to create/destroy an `fbar`, register/unregister work-items with an `fbar`, and synchronize the registered work-items. However, HSA's current `fbar` instruction is not able to distinguish work-items in a wavefront, which is required by our proposal. Thus, we make the case that future GPUs should allow an arbitrary set of work-items to be registered/unregistered with an `fbar`. This would allow a compiler to instrument control flow



(a) Reduce-to-sum network. *(b) Ideal execution.* *(c) SW pred & WG-level exec.* *(d) Fine-grain barriers.*

Figure 5-7. Diverged reduce-to-sum operation.

containing work-group-level operations with `fbar` operations. This idea is demonstrated in Table 5-1c. Unlike the other solutions, this approach does not cause completely inactive wavefronts to continue executing (Figure 5-7d).

A second aspect to implementing diverged work-group-level operations is submitting non-interfering values for inactive work-items. Each inactive work-item causes an execution unit on the GPU to become idle. Hence, those idle execution resources can be temporarily predicated on to submit the non-interfering values.

5.4 Methodology and Workloads

We prototyped Gravel on an eight-node cluster. Each node has an AMD APU with four CPU threads and an HSA-enabled integrated GPU. The nodes are connected by a 56 Gb InfiniBand link. More details can be found in Table 5-2.

Gravel's aggregator is realized by using the integrated CPU to consume GPU-initiated messages and repack them into per-node queues. We use MPI to send/receive the queues and allocate three queues per node (over allocation helps hide network latency). Each per-node queue is 64 kB, which we found is large enough to obtain most of the benefit of large messages on our system and does not consume an excessive amount of memory.

To obtain the necessary thread support, all network requests are funneled through a dedicated network thread [61]. Upon receiving a per-node queue, the network thread iterates through each message and resolves it as a local memory operation (e.g., load, store). The aggregator performs best with one CPU thread because there are several background threads in the system (i.e., Gravel's network thread, an HSA background thread, and an MPI progress thread).

Currently, Gravel supports the following non-blocking network operations: `PUT`, atomic increment, and a primitive active message API. `PUT` and atomic increment operate on a partitioned global address space (PGAS). Atomic operations (i.e., atomic increment and active messages) are serialized by routing them through Gravel's network thread. Thus, some operations that can execute locally are still routed through the NI. On our system, this approach is faster than using concurrent read-modify-write operations. Furthermore, it simplifies writing active messages.

Six applications are evaluated with the inputs in Table 5-3. Our implementations of the graph algorithms (i.e., PR, SSSP, and color) are based on GasCL, which is a single-node graph processing system for GPUs [62]. The following text summarizes each application.

- **Giga-updates-per Second (GUPS):** Described in Section 3.1 [33].
- **PageRank (PR):** Ranks web pages by iteratively sending each vertex's rank through its links.

Table 5-2. Node architecture.

Processor (AMD A10-7850K)	*CPU*: 2 cores (4 threads); 3.7 GHz; 16 kB L1D; 2 MB L2
	GPU: 8 CUs; 720 MHz; 16 kB L1D; 2 MB L2
Memory	32 GB; DDR3-1600; 2 channels
NIC	56 Gb/s InfiniBand card
Software	Ubuntu 14.04; Open MPI 1.10.1; GCC 4.9.3; HSA runtime 1.0.3
Gravel's configuration	24 per-node queues (each 64 kB; 125 µs timeout); 1 MB producer/consumer queue; 1 aggregator thread

Table 5-3. Application inputs.

benchmark(s)	**inputs**
GUPS	~180 million updates
PR-1; SSSP-1; color-1	hugebubbles-00020 [63] (~21 million vertices, ~64 million edges)
PR-2; SSSP-2; color-2	cage15 [63] (~5 million vertices, ~99 million edges)
kmeans	8 clusters, 16 million points
mer	human-chr14 [64] (3.6 GB)

- **Single-source/shortest-path (SSSP):** Calculates the shortest distance from a source vertex to every other vertex.
- **Graph coloring (color):** Labels each vertex in a graph such that no two neighbors have the same color.
- **Kmeans clustering (kmeans):** Iteratively groups a set of Cartesian coordinates into a fixed number of clusters.
- **Meraculous graph construction (mer):** Two phases of a genome sequencing pipeline [65]. The first phase constructs and populates a distributed hash table and the second phase traverses the hash table. We accelerated and measured the first phase. The second phase has significant branch divergence and is left for future work.

5.5 Results and Analysis

In this section, we analyze Gravel's scalability (Section 5.5.1) and then compare Gravel to prior models for enabling system-level operations on GPUs (Section 5.5.2).

5.5.1 Scalability Analysis

Gravel's scalability is depicted in Figure 5-8. Two factors that impact scalability are the frequency of *remote* data access (i.e., an access through the network), and the cost of a remote access relative to a local access. For each input, Table 5-4 summarizes the frequency of remote data access and the average message size, which influences the cost of a remote access.

Recall that our implementation serializes atomic operations (i.e., `fetch-add` and active messages) by routing all of them—including local operations—through the NI. Thus, the throughput for atomics is similar for local and remote access. GUPS, kmeans, and mer, use atomics exclusively. Thus, even though these applications are dominated by remote accesses, as shown in Table 5-4, they approach the ideal speedup of 8x.

PR and color use non-atomic operations (i.e., `PUT` operations) exclusively. A local `PUT` is executed by the GPU directly as a store. Thus, for PR and color, local operations achieve more concurrency than remote operations because they execute across the GPU's massively parallel architecture. In contrast, remote operations are executed by CPU threads (i.e., Gravel's network thread) across the seven receiving machines. We experimented with helper threads at the receiver to recover some of the lost concurrency, but the CPU is already saturated. Thus, we observe little benefit from helper threads.

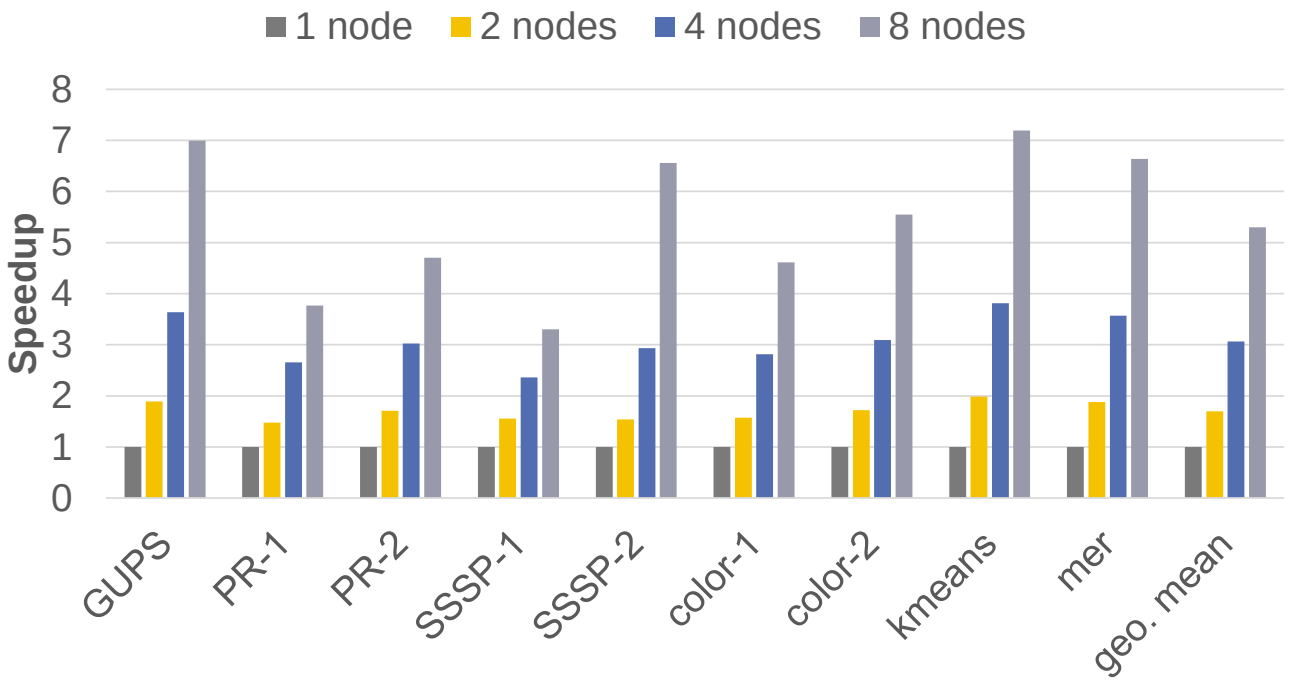


Figure 5-8. Gravel's scalability.

Table 5-4. Network statistics for Gravel at eight nodes.

	Remote access frequency	*Average message size (bytes)*
GUPS	87.5%	65,440
PR-1	37.7%	64,611
PR-2	16.5%	15,700
SSSP-1	30.0%	1,563
SSSP-2	16.2%	57,916
color-1	36.7%	27,258
color-2	16.5%	9,463
kmeans	87.5%	5,656
mer	87.5%	64,822

Finally, SSSP uses atomic operations (i.e., active messages) and `PUT` operations. Specifically, SSSP-2 approaches the ideal speedup because remote access is infrequent and Gravel is able to combine remote accesses into large messages (i.e., ~58 kB as shown in Table 5-4). In contrast, remote access occurs more frequently in SSSP-1 and the cost of those accesses is higher because Gravel's aggregator is not effective for this input (i.e., messages are ~1.6 kB on average). As a result, SSSP-1 does not scale as well as other inputs.

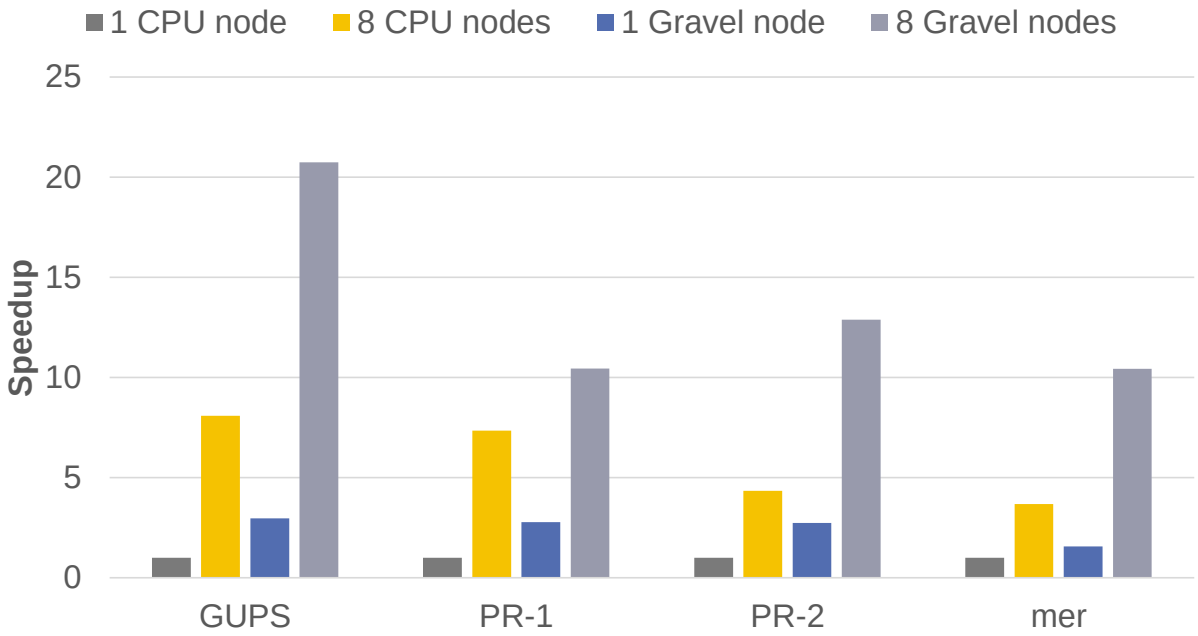


Figure 5-9. Gravel vs. CPU-based distributed systems.

To put these results into perspective, we compared Gravel to CPU-based distributed systems, which fail to leverage the GPU. Specifically, Figure 5-9 shows how Gravel compares to Grappa [66] for GUPS and PR and to UPC [65] for mer. Notice that Gravel is significantly faster on one node, where aggregation and networking are irrelevant. Fundamentally, the GPU's massively parallel architecture is better suited to the underlying data-parallel behavior of these workloads and this advantage translates to eight nodes, where Gravel continues to outperform CPU-based systems.

Finally, Figure 5-10 shows how the per-node queue size, which determines the maximum size of a network message, effects GUPS. In general, larger queues provide better multi-node performance, but the benefit diminishes beyond 32 kB. Thus, to obtain good performance without using an excessive amount of memory, we use 64 kB per-node queues.

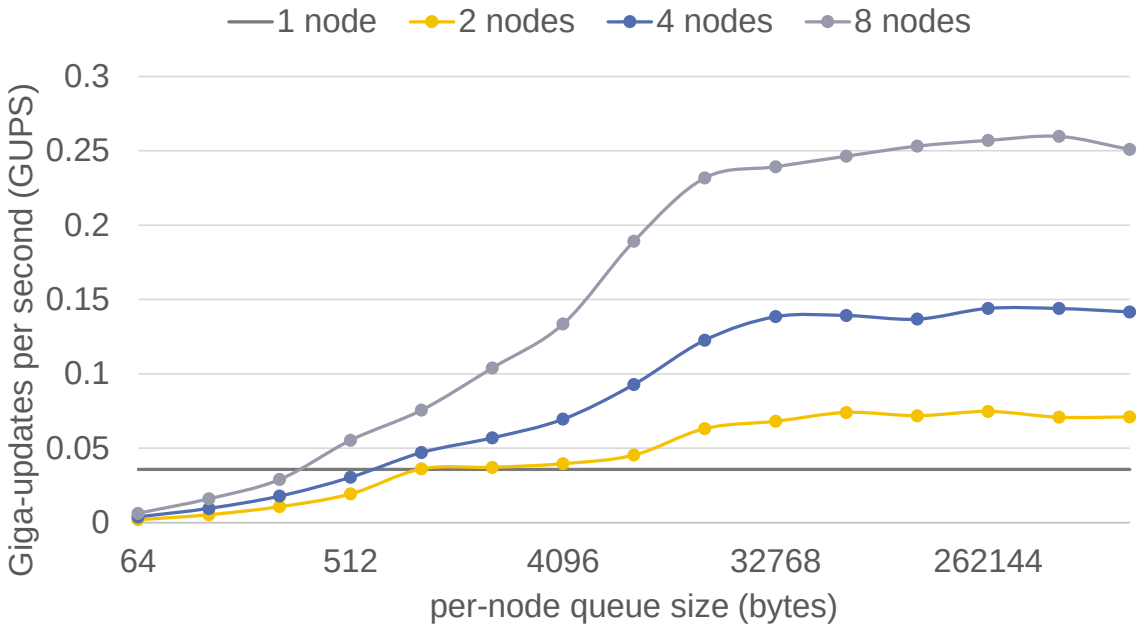


Figure 5-10. Gravel's aggregation sensitivity.

5.5.2 Style Comparison

We wrote versions of each application for each GPU system-level operation model using the methodology described in Section 3.1. Figure 5-11 demonstrates that Gravel performs equal to or better than other models in all cases. Note, in this section, we treat prior implementations of the operations-per-lane model, which forego aggregation, as a separate data point from Gravel to emphasize the performance impact of aggregation.

The first set of bars, labeled *coprocessor*, were generated by configuring the coprocessor model to use the same amount of buffering as Gravel. Recall, the number of work-items executing concurrently is limited to avoid overflowing a per-node queue. Such small per-node queues, which are sufficient for Gravel, limit the amount of parallelism on the GPU, causing this version of the coprocessor model to perform worse than Gravel in all cases. This effect is pronounced for PR and color, where work-items access the network many times.

In the second set of bars, labeled *coprocessor + extra buffering*, we allocate 1 MB for each per-node queue, which is an order of magnitude more space for per-node queues than Gravel. While this enables GUPS and SSSP-2 to perform as well as Gravel, most applications still perform worse. This is because Gravel is more effective at overlapping communication and computation.

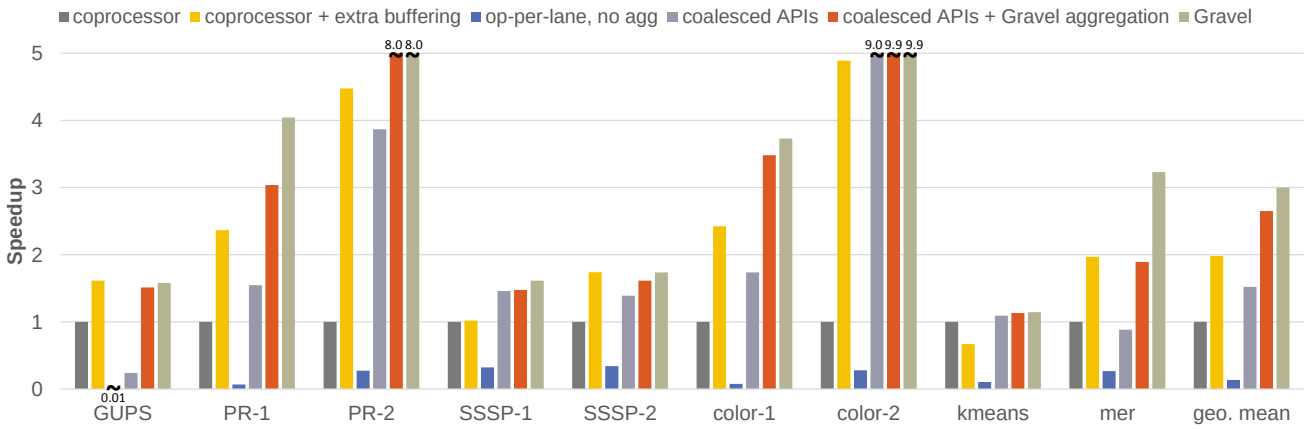


Figure 5-11. Style comparison at eight nodes.

Specifically, in Gravel, per-node queues are sent through the network as soon as they become full or exceed a timeout, while the coprocessor model delays sending a message until the GPU kernel completes. This effect is pronounced for kmeans, which actually performs worse in the coprocessor model with larger per-node queues.

The third set of bars, labeled *op-per-lane, no agg*, bypasses the aggregator. Figure 5-11 shows that sending small messages directly degrades performance. Similarly, the fourth set of bars, labeled *coalesced APIs*, shows that combining messages across a work-group is not sufficient in most cases. Other than SSSP-1, SSSP-2, and kmeans, coalesced APIs leads to messages that are too small.

Finally, the second-to-last set of bars, labeled *coalesced APIs + GPU-wide aggregation*, routes messages initiated in the coalesced APIs model through Gravel's aggregator, which helps to decouple the impact of the small messages from the overhead of sorting on the GPU. Figure 5-11 shows that Gravel's aggregator helps coalesced APIs to perform nearly as well as Gravel. One interesting case is mer, which uses more scratchpad than other benchmarks. This scratchpad usage, in combination with the amount of scratchpad used by the coalesced APIs model, limits the number of work-items that execute on the GPU concurrently.

5.6 GPU Networking on Future GPUs

In this section, we explore how future GPUs can provide better support for small messages. Specifically, Section 5.6.1 suggests replacing Gravel's CPU-based aggregator with dedicated hardware and Section 5.6.2 evaluates alternatives to software predication.

5.6.1 Hardware Aggregator

Currently, Gravel leverages the integrated CPU to aggregate messages, which enables us to use current hardware—but this approach is inefficient. Specifically, we found that, even at eight nodes,

the CPU's out-of-order, multi-GHz core spends 65% of its time polling for GPU-initiated messages. Furthermore, these hardware threads cannot be used for other tasks.

Dedicated hardware could do aggregation in a more energy- and latency-efficient manner. A hardware aggregator could be fixed-function logic, but a small programmable core would provide more flexibility. For example, modern GPUs and NICs both incorporate control processors that could be used for aggregation. Placing the aggregator in the GPU would allow it be used for other purposes (e.g., task aggregation, memory allocation) and enable data-parallel optimization (e.g., a GPU-wide memory coalescer).

5.6.2 Diverged Work-group-level Operation Analysis

Gravel uses software predication to achieve the diverged work-group-level semantic on current hardware, but this approach introduces software overhead that could be avoided in a GPU with native support. To test this hypothesis we emulate the behavior of the alternatives proposed in Section 5.3.3.

To emulate a GPU that tracks control flow at work-group granularity we perform synchronization at wavefront granularity. Specifically, code without predication (e.g., Table 5-1a) works on current GPUs when the work-group size is limited to one wavefront. Using this methodology, we observed a 1.28x speedup over software predication for a modified version of GUPS, called GUPS-mod, where each work-item performs a random number of updates and 95% of work-items perform no updates (otherwise, the benchmark is too memory bound to observe interesting performance effects).

Next, we evaluate fine-grain barriers by emulating the desired behavior in software. In our implementation, a software-based `fbar` comprises three scratchpad variables: `MemberCnt`, `ArriveCnt`, and `sense`. Work-items join (leave) a barrier by atomically incrementing

(decrementing) `MemberCnt`. When a work-item arrives at a barrier it atomically increments (decrements) `ArriveCnt` if `sense` is positive (negative) and waits for `sense` to invert. The last work-item to arrive inverts `sense`. Finally, work-items in one wavefront can use an `fbar` before work-items in a second wavefront leave. Thus, a second `fbar` is used to coordinate work-items leaving the primary `fbar`. We found that our software-based `fbar` operations (e.g., Table 5-1c) provide a 1.06x speedup over software predication for GUPS-mod. Unfortunately, implementing an `fbar` in software incurs significant overhead. Thus, this result should be viewed as a lower bound.

5.7 Summary

Gravel enables GPUs to initiate small network messages and we showed that it is more programmable and performant than prior GPU networking models. Our main contributions were to: (1) show that data-parallel hardware can be exploited to amortize synchronization; (2) use this insight to efficiently offload GPU-initiated network messages to Gravel's aggregator; and (3) explore diverged work-group-level semantics to offload messages from divergent code. Furthermore, we believe that these ideas are applicable beyond networking.

To prototype Gravel, we leveraged an integrated GPU, which allowed us to use the CPU for aggregation. Fundamentally, Gravel could work with discrete GPUs as well, with the GPU writing to a producer/consumer queue in main memory over PCIe, and using PCIe atomic operations to synchronize. Current PCIe implementations may limit performance, but future versions of PCIe or more advanced interfaces such as CCIX [67] should improve this situation.

6. RELATED WORK

In this chapter, we survey four categories of related work. First, in Section 6.1, we examine prior work on multi-producer/multi-consumer queues, which encompasses the underlying producer/consumer data structures and algorithms that we have developed in this thesis to enable the GPU to execute system-level operations according to the operation-per-lane model as envisioned in Chapter 3. Notably, we observe that a lot of the prior work has focused on linked-list-based queues, but the GPU's memory coalescing hardware makes it better suited to operate on array-based queues. We then delve into related work on dynamic aggregation of fine-grain work on data-parallel hardware in Section 6.2, which more directly relates to our prototypes of the aggregation layer described in Chapters 4 and 5.

Next, Section 6.3 examines prior work on GPU tasking, which helps to put our work on implementing and utilizing channels in Chapter 4 into perspective. Finally, we conclude this chapter with a summary of the related work on GPU networking and I/O, which relates our work in Chapter 5 on Gravel.

6.1 Multi-producer/multi-consumer Queues

Prior work on lock-free, multi-producer/multi-consumer queues is skewed towards CPUs; it includes linked list- and array-based designs. Linked lists often are preferred because they are not fixed-length and are easier to manage [41][42]. Unfortunately, linked lists are a poor fit for the GPU's memory-coalescing hardware.

Array-based queues often require special atomic operations, limit the size of an element to a machine word, and usually are not as scalable [38][39]. Gottlieb *et al*. described an array-based algorithm without these limitations, but their design is blocking [40]. Channels use conventional CAS, encapsulate user-defined data (of any size), are non-blocking, and scale well on GPUs.

6.2 Dynamic Aggregation for Data-parallel Hardware

GRAMPS, which inspired Gaster and Howes to propose channels (explored in Chapter 4), maps flow graphs to graphics pipelines and provides packet queues to aggregate fine-grain work into data-parallel tasks [46]. Channels and Gravel apply these concepts to more general computation. Our work gives a fresh perspective on how to implement aggregation queues and use them to realize higher-level languages on GPUs.

Dynamic micro-kernels allow programmers to regroup threads using the keyword `spawn` [35]. To support this semantic, a fully associative look-up table (LUT), indexed on the program counter of the branch destination, is proposed. While micro-kernels target mitigating branch divergence, they could be used for dynamic work aggregation. Compared to channels and Gravel, one limitation is that the number of tasks is limited to the number of entries in the LUT.

Stream compaction uses global scan and scatter operations to regroup pixels by their consumption kernels [68]. Channels avoid regrouping by limiting each channel to one consumption function. Gravel leverages a scalar thread to achieve the desired regrouping.

The Softshell GPU task runtime uses persistent GPU work-groups to schedule and aggregate work from a monolithic task queue [69]. Channels instantiate a separate queue for each consumption function and leverage the GPU's control processor to manage those queues. Gravel uses scalar threads on the integrated CPU in place of Softshell's persistent work-groups.

Dynamic thread block launch [70] introduces special hardware to enable GPU-wide task aggregation. In contrast, our work leverages existing mechanisms. Specifically, our channels prototype leverages the GPU's control processor and Gravel make's use of shared-memory synchronization.

Gravel's diverged work-group-level semantic is similar to HSA's `fbar` object in that it enables a subset of data-parallel lanes to coordinate [59]. It also resembles unconditional operations as described by Hillis and Steele [71], which provide the ability to temporarily activate inactive data-parallel lanes so that they can participate in a data-parallel computation.

6.3 GPU-directed Dynamic Tasking

Aila and Laine proposed a scheme that they call persistent threads, which bypasses the GPU scheduler and places the scheduling burden directly on the programmer [72]. Exactly enough threads are launched to fill the machine and poll a global work queue. In contrast, channels fill the machine in a data-flow manner and only launch consumers that will dequeue the same work, which encourages higher SIMT utilization.

Tzeng *et al*. also explored task queues within the confines of today's GPUs [73]. Their approach was to operate on a queue at wavefront granularity. They allocated a queue per SIMD unit and achieved load-balance through work stealing/sharing. Channels use dynamic aggregation to provide a more conventional task abstraction.

HSA supports dependencies among kernels [30]. Similarly, dynamic parallelism in CUDA enables coarse-grain work coordination [29]. These approaches require programmers to reason about parallelism at a coarse granularity. We found that specifying dependencies at a coarse granularity, while scheduling work at a fine granularity, worked well for Cilk.

Fung *et al*. proposed that a wavefront's state be checkpointed to global memory for the purposes of recovering from a failed transaction [74]. We propose a wavefront yield instruction to facilitate Cilk on GPUs. While similar, we go a step further by allowing the wavefront to relinquish its execution resources. In contrast, CUDA and HSA only support context switches at kernel granularity.

6.4 GPU Networking and I/O

GPUs typically lack native support for work-items to initiate I/O. DCGN [23], GPUfs [26], and MemcachedGPU [75], all rely on the CPU to orchestrate I/O. GPUDirect RDMA optimizes network I/O by allowing the CPU to initiate direct data transfers between a discrete Nvidia GPU and the NIC [19] .

GPUnet builds off of GPUDirect RDMA to provide a coalesced socket API for GPUs [24]. Specifically, work-items read and write sockets by sending a request, over PCIe, to a CPU with GPUDirect capabilities. Compared to Gravel, GPUnet does not combine messages and its coalesced APIs are synchronous, as described earlier. In contrast, Gravel enables asynchronous network access, but routes all GPU-initiated messages (i.e., both control and data) through the CPU. Compared to discrete GPUs, routing messages through an integrated CPU incurs less overhead, but future work should aim to incorporate GPUnet's ability to bypass the CPU for data and Gravel's small message support.

NVSHMEM is similar to Gravel in that it enables PGAS-style distributed memory for GPUs [22]. NVSHMEM leverages hardware (i.e., NVLink), which limits it to GPUs on the same PCIe bus. In contrast, Gravel leverages software (i.e., producer/consumer synchronization), which makes it more portable, but forces messages through the CPU. Furthermore, Gravel does GPU-wide aggregation. NVSHMEM probably experiences some natural aggregation (e.g., memory coalescing), but it does not seem to incorporate explicit aggregation. We hypothesize that NVSHMEM would benefit from Gravel-style aggregation on a lower-performance interconnect and when communication is more random (e.g., GUPS).

In particular, GGAS [18] and GPUrdma [25] enable GPU-initiated messages without CPU involvement. Specifically, the GPU driver is modified to expose the network interface controller's

(NIC) doorbell register and GPU code is written to interact directly with the NIC. This contribution is orthogonal to Gravel's, which focuses on providing an efficient and programmable interface to the network. For example, GGAS interacts with the network at wavefront granularity, which we showed is not efficient and GPUrdma uses coalesced APIs.

Several CPU-based systems, including Grappa [66], GraphLab [76], and GMT [77], focused on maximizing small network message performance through aggregation. Gravel, which draws inspiration from these CPU-based systems, employs a GPU-compatible aggregation scheme to enable the GPU to efficiently initiate small messages.

7. CONCLUSIONS AND FUTURE WORK

In this chapter, we first summarize the main contributions of this work (Section 7.1) and then we discuss how future work can build on these contributions (Section 7.2).

7.1 Summary of Contributions

At the highest level, we made two contributions. First, we described how to efficiently coordinate and synchronize system-level operations being executed by GPU threads across the entire GPU. Second, we explored two classes of system-level operations in details: task spawning and networking.

To efficiently coordinate and synchronize system-level operations, we identified leader-level synchronization as a key building block. In short, leader-level synchronization invokes synchronization once per SIMT group, where a SIMT group can be a wavefront or a work-group, rather than once per thread. We then identified two ways to use leader-level synchronization to regroup system-level operations. In the first, called SIMT-direct aggregation, a SIMT group invokes leader-level synchronization once for each of its sub-groups of system-level operations. In the second, called indirect aggregation, a SIMT group offloads its system-level operations to a dedicated aggregator that is better suited to regroup system-level operations.

Using the implementation concepts above, we investigated channels, which is an abstraction for regrouping small tasks into larger tasks that can saturate the GPU's underlying data-parallel hardware. Specifically, we first used SIMT-driven aggregation to implement channels. We then used channels themselves to organize Cilk programs so that they can execute efficiently on GPUs.

We also used the implementation concepts outlined above to combine small network messages being sent to the same destination into larger messages that are more suitable for network transmission. To prototype this idea, we built a software runtime called Gravel and used it to

distributed several applications dominated by small and unpredictable messages across a cluster of eight AMD APUs connected by InfiniBand. We found that Gravel's GPU-initiated network message capability enables these applications to run faster than prior approaches.

7.2 Future Work

There are several promising ways to extend the ideas that we have presented. First, with respect to system-level coordination, there are at least three worthwhile directions to pursue. The first is to improve SIMT-direct aggregation so that it can be invoked once per SIMT group instead of once per aggregation buffer. Similarly, indirect aggregation can be improved by placing a dedicated hardware aggregator next to the GPU. Finally, a third implementation issue is to extend these ideas to work with discrete GPUs.

Next, we discuss extending our research on fine-grain task aggregation and coordination (Chapter 4). There are many directions to take this work. The most obvious is to provide more generalized mechanisms for programming the control processor to schedule channel-flow graphs. It would also be useful to investigate other programming languages that can be mapped to GPUs with channel-flow graphs. Finally, these ideas should be tested and refined on larger GPUs.

Finally, there are several ways to extend our work on Gravel: fine-grain GPU-initiated network messages (Chapter 5). First, we focused on non-blocking operations (e.g., `PUT`, atomic increment). Thus, it would be useful to study blocking operations. The primary difficulty with blocking operations is that work-items consume GPU execution resources until they complete their GPU kernel. Thus, executing a blocking operation from a GPU kernel could be quite expensive. Nonetheless, blocking operations are an important primitive that should be supported by Gravel. Another worthwhile research direction would be to extend Gravel to support discrete GPUs. One challenge is that leader-level synchronization might not produce enough messages to amortize

PCIe overhead, but placing a dedicated hardware aggregator inside of the GPU can help to resolve this issue. Finally, Gravel may not scale to very large systems (e.g., tens of thousands of nodes). One idea that we would like to explore to address this issue is to perform hierarchical aggregation, where a per-destination buffer is mapped to multiple destinations (to reduce the number of per-destination buffers in the system). Note that this approach requires extra messages to be sent by the disaggregator.

A. More Details on Gravel's Implementation

This appendix covers some of Gravel's lower-level implementation details. Specifically, the first section (A.1) shows how network messages flow from the GPU to the network. The message flow is presented as a software pipeline, where each stage is connected by a producer/consumer queue. The second section (A.2) explains the operation of the various producer/consumer queues used in Gravel. The section is designed as progression. Specifically, it begins by explaining how Gravel's CPU-only single-producer/single-consumer (SPSC) and multi-producer/multi-consumer (MPMC) queues operate and concludes by showing how Gravel's GPU-to-CPU producer/consumer queue (often referred to as Gravel's producer/consumer queue) is derived from these simpler queues.

A.1 GPU-initiated Network Message Flow

It is useful to view Gravel's message aggregation approach as a three-stage software pipeline, shown in Figure A-1. Stage 1 consolidates small network messages, initiated by work-items on the GPU, into a common data structure called a *work-group-level package*. In stage 2, CPU threads consume work-group-level packages and route their network messages into *per-node buffers*—arrays of packed messages being sent to the same node. Finally, in stage 3, a dedicated network thread sends per-node buffers through the network. The network thread is also responsible for receiving, disaggregating, and processing per-node buffers.

Producer/consumer queues are used to exchange work-group-level packages and per-node buffers between the pipeline stages. Specifically, the interface between stage 1 and stage 2 is Gravel's producer/consumer queue where the producers are work-groups on the GPU, the consumers are threads on the CPU and the queue elements are work-group-level packages. Similarly, conventional CPU-only producer/consumer queues are used to exchange per-node

buffers between stage 2 and stage 3. The following sub-sections describe the three pipeline stages and their producer/consumer queue interfaces in more detail.

A.1.1 Pipeline Stage 1: Message Consolidation

The role of stage 1 is to consolidate network messages, initiated by work-items across the GPU, into a single data structure, called a work-group-level package. The primary challenge is to do this without excessive global synchronization. Stage 1 exploits work-groups to amortize global synchronization across a large number of messages. Note that stage 1 consolidates all messages, regardless of their destination. This means that they must be processed a second time (in stage 2), to group them by their destination. Fortunately, once messages are in a work-group-level package, they can be processed without synchronization. Furthermore, processing work-group-level packages in stage 2 is overlapped with consolidating messages into work-group-level packages in stage 1.

Figure A-1 shows how stage 1 consolidates messages into work-group-level packages that can be processed by stage 2. Specifically, work-items in a work-group jointly reserve a work-group-level package (described in Section A.2.3) and populate it with their network messages (time ❶). On a CPU, synchronizing every time a network message is initiated would trigger unacceptable coherence overheads. Two phenomena explain why GPU work-items incur modest

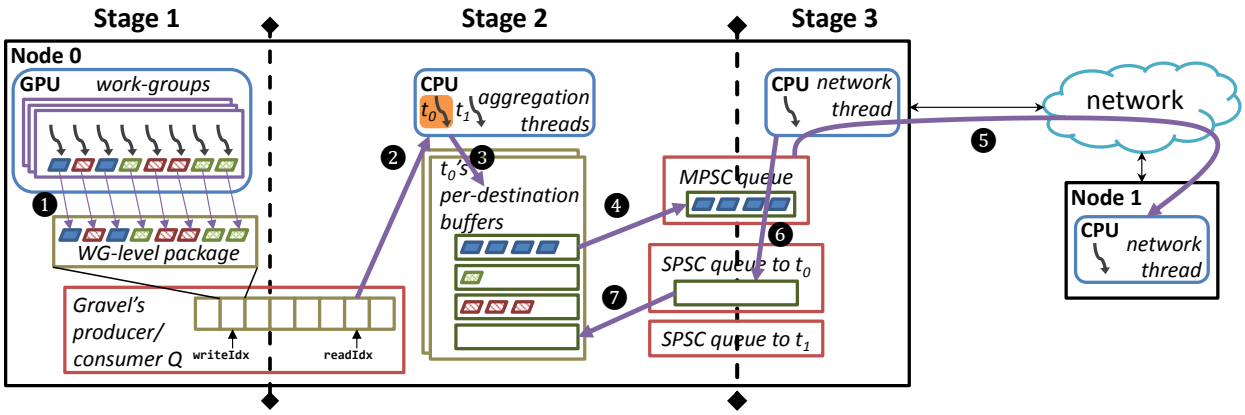


Figure A-1. Gravel's message transit pipeline.

synchronization overheads when updating a work-group-level package. First, global synchronization is amortized across the entire work-group, which comprises a number of work-items (e.g., up to 256 work-items on current AMD GPUs). Second, the work-group-level package is organized such that work-items in the same work-group write their operations to the same cache lines at the same time. The work-group-level package is passed to stage 2, through Gravel's producer/consumer queue, after all of the work-items in the work-group have populated it with their network messages.

A.1.2 Pipeline Stage 2: Message Aggregation

The primary goal of stage 2 is to extract messages from a work-group-level package (time ❷ in Figure A-1) and repack them into per-node buffers (time ❸), which are arrays packed with messages being sent to the same machine node. This is achieved by *aggregator threads* on the CPU. CPU threads are better suited to the control-intensive nature of message binning than GPU work-items, which are susceptible to costly branch and memory divergence. To avoid excessive synchronization, different CPU aggregation threads process different work-group-level packages in parallel. Furthermore, each aggregation thread maintains its own private set of per-node buffers to further reduce synchronization. Figure A-2 demonstrates how three messages, targeted at node 1, are packed to minimize per-message overheads. Per-node buffers are sent to the network for processing after reaching their maximum occupancy or exceeding a timeout (time ❹).

Package Header	**Message Header 1**			**Message Header 2**	
Command: PACKED	Command: PUT	0x1000,	0x4008,	Command: INCREMENT	0x2000
Message Headers: 2	Type: INT	5	42	Type: INT	
	Messages: 2			Messages: 1	

Operations: PUT(0x1000,5,node 1);PUT(0x4008,42,node 1);INC(0x2000,node 1)

Figure A-2. Per-node buffer format example.

A.1.3 Pipeline Stage 3: Network Interface

To send and receive per-node buffers over the network, we follow prior work [66][77] and use MPI. Each node maintains a polling *network thread* on the CPU that receives per-node buffers and processes their messages. Popular MPI implementations, like OpenMPI, currently have limited thread support [61]. In our system, network requests are initiated by different consumers (e.g., `MPI_Send`), the network thread (e.g., `MPI_Irecv`), and host CPU threads (e.g., `MPI_Barrier`). To obtain the required level of thread support, all network requests are funneled through a multi-producer/single-consumer (MPSC) queue to stage 3 of the pipeline, and processed by the network thread (time ❺). The MPSC implementation is the MPMC queue described in Section A.2.2 instantiated with one consumer thread. To avoid deadlock, the network thread converts blocking operations, like `MPI_Barrier`, to multiple non-blocking operations [78].

Per-node buffers cannot be reused until the network thread marks them as available. Our solution is to instantiate an SPSC queue between each aggregator thread and the network thread. The network thread recycles a per-node buffer by pushing its pointer into the per-node buffer's SPSC queue (time ❻ and time ❼). The SPSC queues are implemented as described in Section A.2.1.

A.2 Gravel's Producer/consumer Queue Design

This section discusses the design of Gravel's various producer/consumer queues, which are used to connect the pipeline stages described in the previous section (A.1). Specifically, Gravel uses there queues: an SPSC queue, an MPMC queue, and a GPU-to-CPU producer/consumer queue, which is referred to as "Gravel's producer/consumer."

All three queues are array-based. Compared to a linked-list-based queue, arrays are more amenable to the GPU's memory coalescer and help to avoid expensive dynamic memory

allocation. Each array slot encodes a payload and its synchronization variables. In all cases, a write index (`wrIdx`) and a read index (`rdIdx`) are used to order producers and consumers, respectively. To avoid false sharing, the read and write indices each occupy their own cache line. Array slots are managed through four producer/consumer actions depicted in Figure A-3 (a):

- `reserve()`: producer obtains queue slot for payload
- `publish()`: producer marks queue slot as populated
- `peek()`: consumer checks if queue slot is full
- `release()`: consumer marks queue slot as available

The sections that follow describe how the producer/consumer actions are handled for the three queue variations.

A.2.1 Single-producer/Single-consumer Queue

The SPSC queue uses its read and write indices to synchronize the producer and consumer, which is sufficient because the producer and consumer never conflict if they check that the queue is available (i.e., not empty or full) [79]. Thus, an array slot in the SPSC queue is an opaque set of

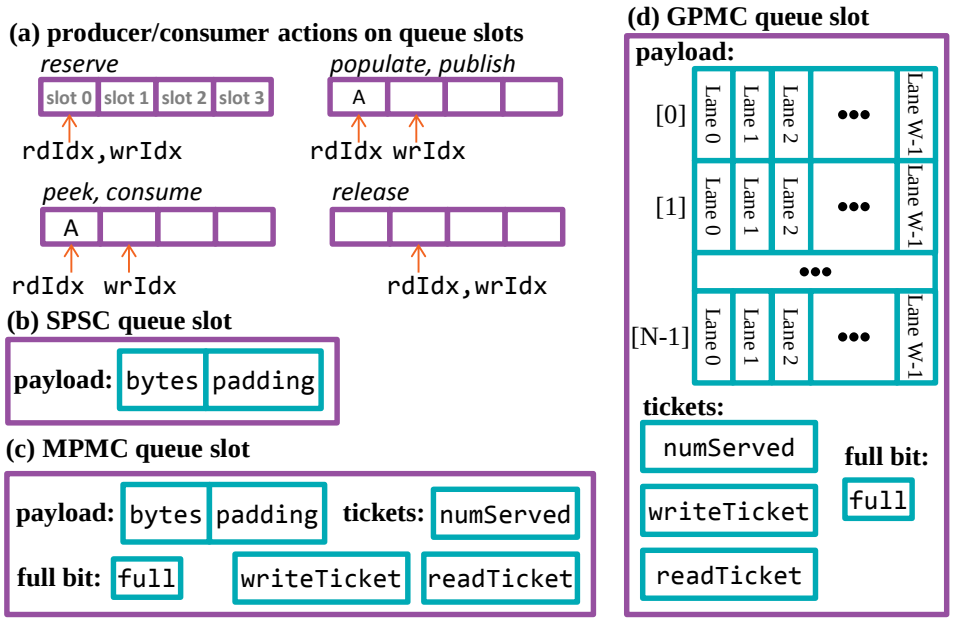


Figure A-3. Producer/consumer queues.

bytes used to encode the payload. This is depicted in in Figure A-3 (b). To avoid false sharing, extra bytes are allocated to pad the payload.

Pseudo-code for the producer/consumer actions can be found in Table A-1's SPSC column. To `reserve` a slot, the producer checks that the queue is not full (line 1) and then retrieves a pointer to the payload at `wrIdx` (line 2). If the queue is full, the producer waits for the consumer to create space. After the producer obtains a slot, it populates the payload. To `publish` its slot, the producer increments `wrIdx` (lines 3-4).

The consumer `peeks` for items to consume. If the queue is not empty (lines 5-6), then the consumer retrieves the payload at `rdIdx` (lines 7-8). To `release` the array slot, the consumer increments `rdIdx` (lines 9-10).

A.2.2 Multi-producer/Multi-consumer Queue

The structure of the MPMC queue is derived directly from the SPSC queue. Notably, the global read and write indices are not sufficient to order producers and consumers. Instead, the following

Table A-1. Pseudo-code for the producer/consumer actions.

	SPSC	MPMC	Gravel
reserve()	1:while (wrIdx + 1 == rdIdx) ; 2:return array[wrIdx].payload;	1:I = wrIdx.fetch_add(1) % 2: array.size; 3:T = 4:array[I].wrTicket.fetch_add(1); 5:while (T != array[I].numServed) ; 6:while (array[I].full) ; 7:return array[I].payload;	1:L = get_local_id(0); 2:leader = reduce_max(L); 3:I = 0; 4:if (L == leader) 5: *MPMC::reserve() lines **1-4*** 6:I = reduce_add(I); 7:return 8: &array[I].payload[0][L];
publish()	3:wrIdx = (wrIdx + 1) % 4: array.size;	8:array[I].full = true;	9:L = get_local_id(0); 10:leader = reduce_max(L); 11:if (L == leader) 12: array[myIdx].full = true;
peek()	5:if (wrIdx == rdIdx) 6: return NULL; 7:return 8: array[rdIdx].payload;	9:I = rdIdx.fetch_add(1) % 10: array.size; 11:T = 12:array[I].rdTicket.fetch_add(1); 13:while (T != array[I].numServed) ; 14:while (!array[I].full) ; 15:return array[I].payload;	/* same as MPMC::peek() */
release()	9:rdIdx = (rdIdx + 1) % 10: array.size	16:array[I].full = false; 17:array[I].numServed.fetch_add(1);	/* same as MPMC::release() */

invariant drives the producer/consumer actions: *at most, one agent (i.e., a producer or a consumer) can access an array slot at a time*.

Three conflicts warrant enforcing the invariant. The first occurs when two or more producers alias to the same array slot. To maintain the invariant, exactly one producer is chosen to proceed. Other producers must wait or try again. The second conflict arises when two or more consumers alias to the same array slot. This scenario is handled like the first—one consumer is chosen to proceed. After resolving the first two conflicts, a third occurs between the chosen producer and the chosen consumer. The conflict is resolved by choosing at most one to proceed.

Figure A-3 (c) depicts an array slot in the MPMC queue. In addition to the padded payload, there are variables to enforce the invariant as described in the prior paragraph. Tickets (i.e., `numServed`, `writeTicket`, and `readTicket`) are used to order producers (conflict 1) and consumers (conflict 2). A full bit (i.e., `full`) is used to order a producer and consumer that both have a valid ticket (conflict 3). Table A-1's MPMC column has pseudo-code for the producer/consumer actions. Initially, all array slots are empty, their full bits are clear, both tickets (i.e., `writeTicket` and `readTicket`) are zero, and the number of tickets served (i.e., `numServed`) is zero.

To `reserve` a slot, a producer does a `fetch-add` on `wrIdx`, which returns an offset, `I`, into the array (lines 1-2). The producer then obtains a unique write ticket, `T`, by doing a fetch-and-add on the slot's `writeTicket` (lines 3-4). Conflict 1 is resolved by waiting for the producer's write ticket, `T`, to equal `numServed` (line 5). Conflict 3 is resolved by waiting for the slot's full bit to be clear (line 6). Finally, a pointer to the slot's payload is returned (line 7) and populated. To `publish` the slot, the producer sets the slot's full bit[1] (line 8).

[1] The full bit at offset `I` is located from the payload pointer.

The `peek` action is similar to the reserve action. It involves obtaining a queue slot and waiting for that slot to be populated[2]. Consumers obtain a ticket (used to resolve conflict 2) by performing a `fetch-add` on the slot's `rdTicket` (lines 11-12). Conflict 3 is resolved by waiting for the slot's full bit to be set (line 14). To `release` its slot, the consumer clears its slot's full bit (line 16) and increments `numServed` (line 17), which allows other producers and consumers to make progress.

A.2.3 Gravel's Producer/consumer Queue

Gravel's producer/consumer queue follows directly from the MPMC queue. Because producers are work-groups—groups of work-items executing on the same CU—Gravel's producer/consumer queue differs in two ways. First, to facilitate optimal memory coalescing, the payload is organized as a column-major array—shown in Figure A-3 (d). Thus, words written by adjacent work-items are adjacent in memory. Because adjacent work-items write adjacent locations in memory, the payload does not need to be padded.

The second difference is that work-groups choose a leader work-item to synchronize with other producers and consumers. This leader uses the MPMC actions to adhere to the MPMC invariant (at most one agent can access an array slot at a time) and to obtain a payload pointer. After obtaining a payload pointer, the leader broadcasts the pointer to the other work-items in its work-group.

Table A-1's Gravel column has pseudo-code for the producer actions. To `reserve` a slot, the producer work-group elects a leader work-item by comparing each work-item's unique lane ID, `L` (line 1). Specifically, each work-item participates in a data-parallel reduction to determine the maximum lane ID (line 2). If a work-item's lane ID equals the maximum lane ID then that work-item is the leader (line 4). The leader lane mimics the MPMC `reserve` action (line 5). Finally,

[2] The pseudo-code resolves conflicts 2 and 3 by blocking the consumer. A reentrant call to a non-blocking version must retrieve the consumer's offset.

the leader lane uses a reduction to broadcast the work-group's offset to other work-items in the work-group.

To `publish` the slot, the work-group elects a leader work-item (lines 9-11) to set the full bit (line 12). A data parallel reduction is collective across the work-group, which means that all work-items are guaranteed to reach it before the leader sets the full bit. The `peek` and `release` actions, which are executed by consumer threads on the integrated CPU, are identical to the MPMC `peek` and `release` actions.

BIBLIOGRAPHY

[1] G. E. Moore, "Cramming More Components onto Integrated Circuits," *in Proceedings of the IEEE Journal of Electronics*, vol. 38, no. 8, pp. 114–117, Apr. 1965.

[2] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. LeBlanc, "Design of Ion-implanted MOSFET's with Very Small Physical Dimensions," *in Proceedings of the IEEE Journal of Solid-State Circuits*, vol. 9, no.5, Oct. 1974.

[3] H. Sutter, "The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software," *in Dr. Dobb's Journal*, March. 2005. [Online]. Available: http://www.gotw.ca/publications/concurrency-ddj.htm.

[4] M. H. Lipasti and J. P. Shen, "Superspeculative Microarchitecture for Beyond AD 2000," *in Proceedings of the Journal of IEEE Computer*, vol. 30, no. 9, pp. 59–66, Aug. 1997.

[5] "The Green500 List." [Online]. http://www.green500.org/

[6] "Amazon Elastic Compute Cloud: User Guide for Linux Instances." [Online]. http://docs.aws.amazon.com/AWSEC2/ latest/UserGuide/using_cluster_computing.html.

[7] T. Geller, "Supercomputing's Exaflop Target," *in Communications of the ACM*, vol. 54, no. 8, pp. 16–18, Aug. 2011.

[8] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mane, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viegas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems," *Google preliminary whitepaper*, Mar. 2015.

[9] D. Yu, K. Yao, and Y. Zhang, "The Computational Network Toolkit [Best of the Web]," *in IEEE Signal Processing Magazine*, vol. 32, no. 6, Nov. 2015.

[10] R. Collobert, K. Kavukcuoglu, and C. Farabet, "Torch7: A Matlab-like Environment for Machine Learning," *in BigLearn NIPS Workshop*, Jan. 2011.

[11] "Install GraphLab Create with GPU Acceleration," [Online]. https://turi.com/download/install-graphlab-create-gpu.html.

[12] S. Keckler, "Life After Dennard and How I Learned to Love the Picojoule," *Keynote for the International Symposium on Microarchitecture (MICRO)*, Dec. 2011.

[13] D. R. Hower, B. A. Hechtman, B. M. Beckmann, B. R. Gaster, M. D. Hill, S. K. Reinhardt, and D. A. Wood, "Heterogeneous-race-free Memory Models," *in Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 427–440, Mar. 2014.

[14] B. R. Gaster, D. R. Hower, and L. Howes, "HRF-Relaxed: Adapting HRF to the complexities of industrial heterogeneous memory models," *in Proceedings of the Journal of ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 12, no. 1, pp. 1–26, Apr. 2015.

[15] W. W. L. Fung, I. Sham, G. Yuan, and T. M. Aamodt, "Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow," *in Proceedings of the International Symposium on Microarchitecture (MICRO)*, pp. 407–418, Dec. 2007.

[16] W. Fung and T. Aamodt, "Thread Block Compaction for Efficient SIMT Control Flow," *in Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*, pp. 25–36, Feb. 2011.

[17] N. Chatterjee, M. O'Connor, G. H. Loh, N. Jayasena, and R. Balasubramonian, "Managing DRAM Latency Divergence in Irregular GPGPU Applications," *in Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pp. 128–139, Nov. 2014.

[18] L. Oden and H. Fröning, "GGAS: Global GPU Address Spaces for Efficient Communication in Heterogeneous Clusters," *in Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER)*, Sep. 2013.

[19] S. Potluri, K. Hamidouche, A. Venkatesh, D. Bureddy, and D. K. Panda, "Efficient Inter-node MPI Communication using GPUDirect RDMA for InfiniBand Clusters with NVIDIA GPUs," *in Proceedings of the International Conference on Parallel Processing (ICPP)*, Oct. 2013.

[20] H. Wang, S. Potluri, M. Luo, A. Singh, S. Sur, and D. Panda, "MVAPICH2-GPU: Optimized GPU to GPU Communication for InfiniBand Clusters," *in Proceedings of the International Supercomputing Conference (ISC)*, May 2011.

[21] "CUDA C Programming Guide: Appendix C. CUDA Dynamic Parallelism." [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#cuda-dynamic-parallelism.

[22] S. Potluri, N. Luehr, and N. Sakharnykh, "Simplifying Multi-GPU Communication with NVSHMEM," *presented at the GPU Technology Conference*, Apr. 2016. [Online]. Available: http://on-demand.gputechconf.com/gtc/2016/presentation/s6378-nathan-luehr-simplyfing-multi-gpu-communication-nvshmem.pdf.

[23] J. Stuart and J. Owens, "Message Passing on Data-Parallel Architectures," *in Proceedings of the IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, May 2009.

[24] S. Kim, S. Huh, Y. Hu, X. Zhang, E. Witchel, A. Wated, and M. Silberstein, "GPUnet: Networking Abstractions for GPU Programs," *in Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 201–216, Oct. 2014.

[25] F. Daoud, A. Watad, and M. Silberstein, “GPUrdma: GPU-side library for high performance networking from GPU kernels,” *in Proceedings of the International Workshop on Runtime and Operating Systems for Supercomputers (ROSS)*, June 2016.

[26] M. Silberstein, B. Ford, I. Keidar, and E. Witchel, “GPUfs: Integrating File Systems with GPUs,” *in Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 485–498, Mar. 2013.

[27] J. Kim and C. Batten, “Accelerating Irregular Algorithms on GPGPUs Using Fine-Grain Hardware Worklists,” *in Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 75–87, Dec. 2014.

[28] B. R. Gaster and L. Howes, “Can GPGPU Programming Be Liberated from the Data-Parallel Bottleneck?” *in IEEE Computer*, vol. 45, no. 8, pp. 42–52, Aug. 2012.

[29] “CUDA C Programming Guide.” [Online]. Available: http://docs.nvidia.com/cuda/cuda-c-programming-guide/.

[30] G. Kyriazis, “Heterogeneous System Architecture: A Technical Review,” *AMD whitepaper*, Aug. 2012.

[31] S. Junkins, “The Compute Architecture of Intel® Processor Graphics Gen9,” *Intel whitepaper v1.0*, Aug. 2016.

[32] “OpenCL 2.0 Reference Pages.” [Online]. http://www.khronos.org/registry/cl/sdk/2.0/docs/man/xhtml/.

[33] “HPC Challenge Benchmark: RandomAccess.” [Online]. Available: http://icl.cs.utk.edu/projectsfiles/hpcc/RandomAccess/.

[34] Wikipedia, “Counting sort.” [Online]. https://en.wikipedia.org/wiki/Counting_sort.

[35] M. Steffen and J. Zambreno, “Improving SIMT Efficiency of Global Rendering Algorithms with Architectural Support for Dynamic Micro-Kernels,” *in Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 237–248, Dec. 2010.

[36] “Intel Threading Building Blocks.” [Online]. Available: http://www.threadingbuildingblocks.org/.

[37] S. Min, C. Iancu, and K. Yelick, “Hierarchical Work Stealing on Manycore Clusters,” *in Proceedings of the Conference on Partitioned Global Address Space Programming Models (PGAS)*, Oct. 2011.

[38] J. Valois, “Implementing Lock-Free Queues,” *in Proceedings of the International Conference on Parallel and Distributed Computing Systems*, pp. 64–69, Oct. 1994.

[39] C. Gong and J. M. Wing, “A Library of Concurrent Objects and Their Proofs of Correctness,” *Carnegie Mellon University (CMU) technical report*, July 1990.

[40] A. Gottlieb, B. D. Lubachevsky, and L. Rudolph, "Basic Techniques for the Efficient Coordination of Very Large Numbers of Cooperating Sequential Processors," *in Proceedings of the ACM Journal of Transactions on Programming Languages and Systems (TOPLAS)*, vol. 5, no. 2, pp. 164–189, Apr. 1983.

[41] M. M. Michael and M. L. Scott, "Nonblocking Algorithms and Preemption-Safe Locking on Multiprogrammed Shared Memory Multiprocessors," *in Proceedings of the Journal of Parallel and Distributed Computing*, vol. 51, no. 1, pp. 1–26, May 1998.

[42] E. Ladan-mozes and N. Shavit, "An Optimistic Approach to Lock-Free FIFO Queues," *in Proceedings of the International Symposium on Distributed Computing (DISC)*, pp. 117–131, Oct. 2004.

[43] M. Harris, "Optimizing Parallel Reduction in CUDA," *NVIDIA presentation*, 2007. [Online]. Available: http://developer.download.nvidia.com/assets/cuda/files/reduction.pdf.

[44] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *in Proceedings of the ACM Symposium on Operating System Design and Implementation (OSDI)*, Dec. 2004.

[45] W. Thies, M. Karczmarek, and S. P. Amarasinghe, "StreamIt: A Language for Streaming Applications," *in Proceedings of the International Conference on Compiler Construction (CC)*, pp. 179–196, Apr. 2002.

[46] J. Sugerman, K. Fatahalian, S. Boulos, K. Akeley, and P. Hanrahan, "GRAMPS: A Programming Model for Graphics Pipelines," *in Proceedings of the ACM Journal of Transactions on Graphics (TOG)*, vol. 28, no. 1, pp. 4:1–4:11, Jan. 2009.

[47] M. Frigo, C. E. Leiserson, and K. H. Randall, "The Implementation of the Cilk-5 Multithreaded Language," *in Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, pp. 212–223, June 1998.

[48] G. Diamos and S. Yalamanchili, "Speculative Execution on Multi-GPU Systems," *in Proceedings of the IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, Apr. 2010.

[49] Y. Guo, R. Barik, R. Raman, and V. Sarkar, "Work-First and Help-First Scheduling Policies for Async-Finish Task Parallelism," *in Proceedings of the IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, pp. 1–12, May 2009.

[50] 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," *in Proceedings of the SIGARCH Journal of Computer Architecture News (CAN)*, vol. 39, no. 2, pp. 1–7, May. 2011.

[51] P. Conway and B. Hughes, "The AMD Opteron Northbridge Architecture," *in Proceedings of the Journal of IEEE Micro*, vol. 27, no. 2, pp. 10–21, Mar. 2007.

[52] B. A. Hechtman and D. J. Sorin, "Exploring Memory Consistency for Massively-Threaded Throughput-Oriented Processors," *in Proceedings of the International Symposium on Computer Architecture (ISCA)*, pp. 201–212, June 2013.

[53] B. A. Hechtman, S. Che, D. R. Hower, Y. Tian, B. M. Beckmann, M. D. Hill, S. K. Reinhardt, and D. A. Wood, "QuickRelease: A Throughput-oriented Approach to Release Consistency on GPUs," *in Proceedings of the IEEE International Symposium On High Performance Computer Architecture (HPCA)*, Feb. 2014.

[54] V. Strassen, "The Asymptotic Spectrum of Tensors and the Exponent of Matrix Multiplication," *in Proceedings of the Annual Symposium on Foundations of Computer Science*, pp. 49–54, Oct. 1986.

[55] AMD Corporation, "AMD Accelerated Parallel Processing SDK." [Online]. Available: http://developer.amd.com/tools-and-sdks/.

[56] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt, "Analyzing CUDA Workloads Using a Detailed GPU Simulator," *in Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 163–174, Apr. 2009.

[57] T. G. Rogers, M. O'Connor, and T. M. Aamodt, "Cache-Conscious Thread Scheduling for Massively Multithreaded Processors," *in Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 72–83, Dec 2012.

[58] A. Parashar, M. Pellauer, M. Adler, B. Ahsan, N. Crago, D. Lustig, V. Pavlov, A. Zhai, M. Gambhir, A. Jaleel, R. Allmon, R. Rayess, S. Maresh, and J. Emer, "Triggered Instructions: A Control Paradigm for Spatially-Programmed Architectures," *in Proceedings of the International Symposium on Computer Architecture (ISCA)*, pp. 142–153, June 2013.

[59] "HSA Programmer's Reference Manual: HSAIL Virtual ISA and Programming Model, Compiler Writer's Guide, and Object Format (BRIG) Version 1.0.1," *HSA foundation technical document*, July 2015. [Online]. Available: http://www.hsafoundation.com/?ddownload=4945.

[60] H. Levy, "Single Producer Consumer on a bounded array problem," *University of Washington course notes*, 2003. [Online]. https://courses.cs.washington.edu/courses/cse451/03wi/section/prodcons.htm.

[61] "OpenMPI FAQ." [Online]. https://www.open-mpi.org/faq/?category=supported-systems#thread-support.

[62] S. Che, "GasCL: A Vertex-Centric Graph Model for GPUs," *in Proceedings of the IEEE High Performance Extreme Computing Conference (HPEC)*, Sep. 2014.

[63] "University of Florida Sparse Matrix Collection." [Online]. http://www.cise.ufl.edu/research/sparse/matrices/.

[64] NERSC, "Meraculous data." [Online]. http://portal.nersc.gov/project/m888/apex/Meraculous_data/.

[65] E. Georganas, A. Buluç, J. Chapman, L. Oliker, D. Rokhsar, and K. Yelick, "Parallel De Bruijn Graph Construction and Traversal for De Novo Genome Assembly," *in Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pp. 437–448, Nov. 2014.

[66] J. Nelson, B. Holt, B. Myers, P. Briggs, L. Ceze, S. Kahan, and M. Oskin, "Latency-tolerant Software Distributed Shared Memory," *in Proceedings of the USENIX Annual Technical Conference (ATC)*, pp. 291–305, July 2015.

[67] CCIX Consortium, "Cache Coherent Interconnect for Accelerators (CCIX)." [Online]. http://www.ccixconsortium.com.

[68] J. Hoberock, V. Lu, Y. Jia, and J. C. Hart, "Stream Compaction for Deferred Shading," *in Proceedings of the Conference on High Performance Graphics (HPG)*, pp. 173–180, Aug. 2009.

[69] M. Steinberger, B. Kainz, B. Kerbl, S. Hauswiesner, M. Kenzel, and D. Schmalstieg, "Softshell: Dynamic Scheduling on GPUs," *in Proceedings of the ACM Journal of Transactions on Graphics (TOG)*, vol. 31, no. 6, pp. 161:1–161:11, Nov. 2012.

[70] J. Wang, N. Rubin, A. Sidelnik, and S. Yalamanchili, "Dynamic Thread Block Launch: a Lightweight Execution Mechanism to Support Irregular Applications on GPUs," *in Proceedings of the International Symposium on Computer Architecture (ISCA)*, pp. 528–540, June 2015.

[71] W. D. Hillis and G. L. Steele, Jr., "Data Parallel Algorithms," *in Communications of the ACM – Special issue on parallelism*, vol. 29, no. 12, pp. 1170–1183, Dec. 1986.

[72] T. Aila and S. Laine, "Understanding the Efficiency of Ray Traversal on GPUs," *in Proceedings of the Conference on High Performance Graphics (HPG)*, pp. 145–149, Aug. 2009.

[73] S. Tzeng, A. Patney, and J. D. Owens, "Task Management for Irregular-Parallel Workloads on the GPU," *in Proceedings of the Conference on High Performance Graphics (HPG)*, pp. 29–37, June 2010.

[74] W. W. L. Fung, I. Singh, A. Brownsword, and T. M. Aamodt, "Hardware Transactional Memory for GPU Architectures," *in Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 296–307, Dec. 2011.

[75] T. H. Hetherington, M. O'Connor, and T. M. Aamodt, "MemcachedGPU: Scaling-up Scale-out Key-value," *in Proceedings of the ACM Symposium on Cloud Computing (SoCC)*, pp. 43–57, Aug. 2015.

[76] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Power-Graph: Distributed Graph-Parallel Computation on Natural Graphs," *in Proceedings of the USENIX Conference on Operating Systems Design and Implementation (OSDI)*, pp. 17–30, Oct. 2012.

[77] A. Morari, A. Tumeo, D. Chavarria-Miranda, O. Villa, and M. Valero, "Scaling Irregular Applications through Data Aggregation and Software Multithreading," *in Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 1126–1135, May 2014.

[78] W. Gropp and R. Thakur, "Thread Safety in an MPI Implementation: Requirements and Analysis," *in the Journal of Parallel Computing*, vol. 33, no. 9, pp. 595–604, Sep. 2007.

[79] H. Sutter, "Writing Lock-Free Code: A Corrected Queue," *in Dr. Dobb's Journal*, Sep. 2008. [Online]. Available: http://www.ddj.com/high-performancecomputing/210604448.