The Dissertation Committee for Michael Wayne LeBeane
certifies that this is the approved version of the following dissertation:

# Optimizing Communication for Clusters of GPUs

Committee:

Lizy K. John, Supervisor

Steven K. Reinhardt

Mauricio Breternitz Jr.

Mattan Erez

Mohit Tiwari

# Optimizing Communication for Clusters of GPUs

**by**

**Michael Wayne LeBeane**

**DISSERTATION**

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

**DOCTOR OF PHILOSOPHY**

THE UNIVERSITY OF TEXAS AT AUSTIN

August 2018

Dedicated to Preeti.

# Acknowledgments

First, I would like to thank my advisor, Professor Lizy K. John, for her guidance and patience throughout my time in graduate school. Professor John taught me the fundamentals of architecture during my early coursework, and in my later years taught me about the art of writing papers and communicating my work to others. She has always been available to discuss research topics and has spent countless hours reviewing and improving my work.

I would like to thank my PhD committee for their feedback at my qualifying exam and defense. Their constructive criticism greatly improved the quality of this dissertation.

I was fortunate enough to meet a number of extraordinarily kind and brilliant UT graduate students during my time in graduate school. While there are too many names to mention, I would like to specifically acknowledge my lab mates from LCA. Jee Ho Ryoo, Reena Panda, Jiajun Wang, Wooseok Lee, Shuang Song, and Xinnian Zheng have always encouraged me and pushed me to graduate, even when things got hard. I don't know if I would have made it without their unwavering friendship and support.

My friends, colleagues, and mentors at AMD Research were instrumental contributors to the work contained in this dissertation. I would like to thank Khaled Hamidouche and Brad Benton for teaching me more about

high-performance networking runtimes than I ever wanted to know. I would like to thank Brandon Potter, Eric Van Tassell, Sooraj Puthoor, Tony Gutierrez, and Brad Beckmann for helping me solve frustrating simulator bugs for many years now, and for building much of the infrastructure used in this work. I would like to thank Mauricio Breternitz for his mentor-ship, kindness, and availability for spurious coffee breaks. Finally, I would especially like to thank Steve Reinhardt, my unofficial co-advisor, for adopting me while I was interning at AMD, for encouraging me to stay on and work on this topic for my PhD, and for patiently reviewing my terrible code.

I would like to thank my friends and colleagues at Intel's Storage Technology Group, who gave me my first real exposure to engineering research which lead me to the decision to pursue my PhD. I would especially like to thank Annie Foong for being my first real technical mentor and for keeping in touch with me throughout graduate school.

Most of all, I am forever grateful to my Mom, Dad, and sister Rachel for putting up with me for all these years, and for continuing to love and support me. I never would have made it without them.

# Optimizing Communication for Clusters of GPUs

Michael Wayne LeBeane, Ph.D.
The University of Texas at Austin, 2018

Supervisor: Lizy K. John

GPUs are frequently used to accelerate data-parallel workloads across a wide variety of application domains. While GPUs offer a large amount of computational throughput within a single node, the largest problems require a cluster of such devices communicating with different compute nodes across a network. These clusters can range in size from a small handful of machines constructed from commodity parts, to several thousand machines built from specialized components.

Despite widespread deployment of GPUs across clusters both big and small, communication between GPUs in networks of computers remains unwieldy. Networks of GPUs are currently programmed in a clunky coprocessor style, requiring coordination with a host CPU and driver stack to communicate with other systems. These intra-node bottlenecks for initiating communication operations are often much greater than the cost of sending data over a high-performance network.

This dissertation explores new techniques to more tightly integrate GPUs with network adapters to allow efficient communication between GPUs

across the network. It evaluates both hardware and software changes to NICs and GPUs to enable end-to-end, user-space communication between networks of GPUs, avoiding critical path CPU interference.

First, Extended Task Queuing (XTQ) is proposed to provide the ability to launch remote kernels without intervention of a host CPU at the target. Inspired by classic work on active messaging, XTQ uses NIC architectural modifications to support remote kernel launch without the participation of the remote CPU. Bypassing the remote CPU reduces remote kernel launch latencies and allows a more decentralized, cluster-wide work dispatch system.

Next, intra-kernel communication is optimized through the Command Processor Networking (ComP-Net) framework. ComP-Net uses a little-known feature of modern GPUs: embedded, programmable microprocessors that are typically referred to as Command Processors (CPs). GPU communication latency is decreased by running the network software stack on the CP instead of the host CPU. ComP-Net implements a runtime and programming interface that allows the GPU compute units to take advantage of the unique capabilities of a networking CP. Challenges related to the GPU's relaxed memory model and L2 cache thrashing are addressed to reduce the latency of network communication through the CP.

Finally, GPU Triggered Networking (GPU-TN) is proposed as an alternative intra-kernel networking scheme that enables a GPU to directly trigger network operations from within a GPU kernel without the involvement of any CPU on the critical path. Inspired by Portals 4 triggered operations, GPU

Triggered Networking implements a NIC hardware mechanism by which the GPU can directly trigger the network adapter to send messages. In this approach, the host CPU is responsible for creating the network command packet on behalf of the GPU and registering it with the NIC. When the GPU is ready to send a message, it "triggers" the NIC using a memory-mapped store operation. A small amount of additional hardware in the NIC collects these writes from the GPU and initiates the pending network operation when a threshold condition has been met. These optimizations allow for fine-grained remote communication without ending a kernel.

# Table of Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

With the impending end of Moore's Law and Dennard scaling [28], the computing industry has turned to accelerators to continue pushing the performance and power trends of the last 50 years. Chief among the currently proposed accelerator architectures are GPUs. While traditionally used solely for the acceleration of graphics workloads, GPUs have been re-purposed to accelerate a variety of data-parallel applications from domains such as computational finance, oil and gas, data science, climate modeling, and machine learning [74].

Individually, GPUs can provide up to 15.7 teraflops of performance [76], and the biggest, most expensive single node machines with many GPUs can provide upwards of 240 teraflops [75]. However, to solve the largest and most difficult problems, GPU-enabled nodes must be connected over a high-performance computer network. Indeed, there are many supercomputers and data centers that employ large networks of GPUs. For example, 98 of the top 500 fastest supercomputers and 30 of the top 100 machines on the Green 500 list utilize GPUs to reach unparalleled performance per watt on data-parallel workloads [97].

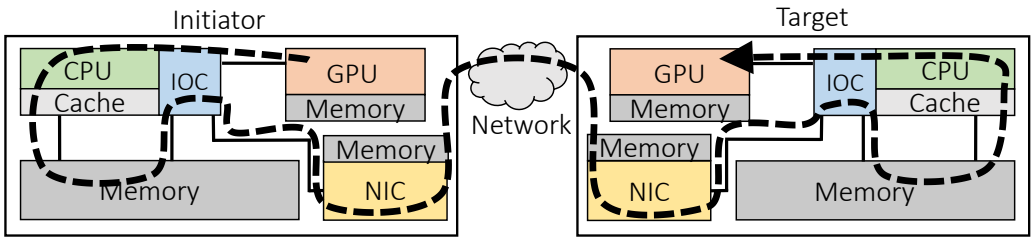



Figure 1.1: Control plane for GPU-to-GPU, inter-node communication (IOC stands for Input/Output Controller).

## 1.1 Problem Description

Despite widespread deployment across clusters both large and small, networks of GPUs are currently programmed in a cumbersome coprocessor style. In order for a GPU to transfer a piece of data from its local memory to the memory of a remote GPU, it must go through an exorbitant number of hops and control flow transfers involving most of the components on a node. As a small example, in Figure 1.1, a GPU on the initiator node wishes to transfer data to a GPU on a target node. In a conventional heterogeneous system, the initiator GPU has to end the currently executing kernel, notify the driver on the host CPU, and call into a networking stack which uses a high-performance Network Interface Controllers (NICs) sitting on an I/O bus to transfer the data to the target. At the target, a reverse sequence of steps will occur, and a new kernel has to be launched, as stale copies of the data can exist in non-coherent caches which are only flushed at kernel boundaries. While emerging technologies such as Nvidia's GPUDirect RDMA [64] and AMD's ROCn RDMA [9] enable NICs to transfer data directly to a GPU's on-board memory, the control plane still involves the tortured route described

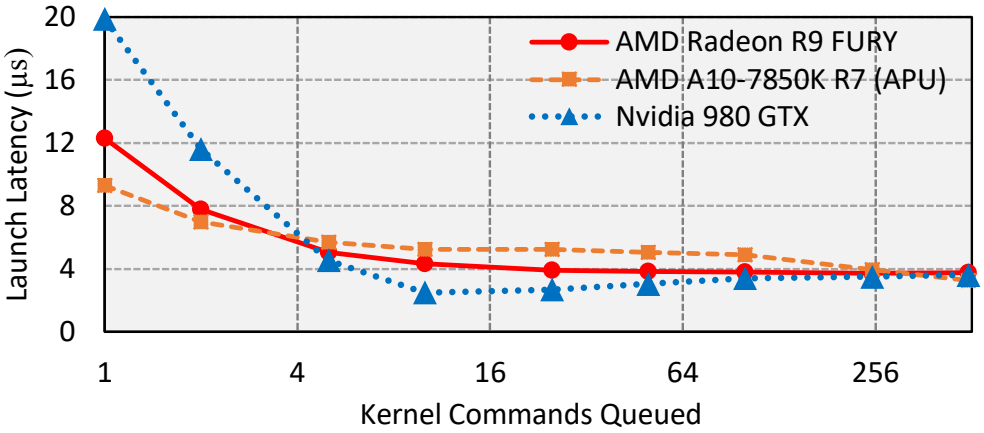



Figure 1.2: Study of kernel launch latencies on modern GPUs.

previously.

These overheads can be significant on modern hardware. To provide some context, Figure 1.2 explores the most expensive part of the above transfer: the overhead of starting and stopping the kernel and communicating this information with the host CPU. This experiment evaluates launch/completion latencies on GPUs from multiple vendors and different form factors. In this experiment, the GPUs' hardware scheduling logic is presented with a variable length sequence of empty kernels. Depending on the size of the kernel stream presented to the scheduler and the details of the target hardware, the launch latencies can vary from 3$\mu$s-20$\mu$s. Recall that the simple example discussed previously involved a kernel launch overhead through the CPU on both the initiator and the target.

Even in the best case, the large dispatch overheads discourage fine-grained or frequent messaging using kernel boundary solutions, and require

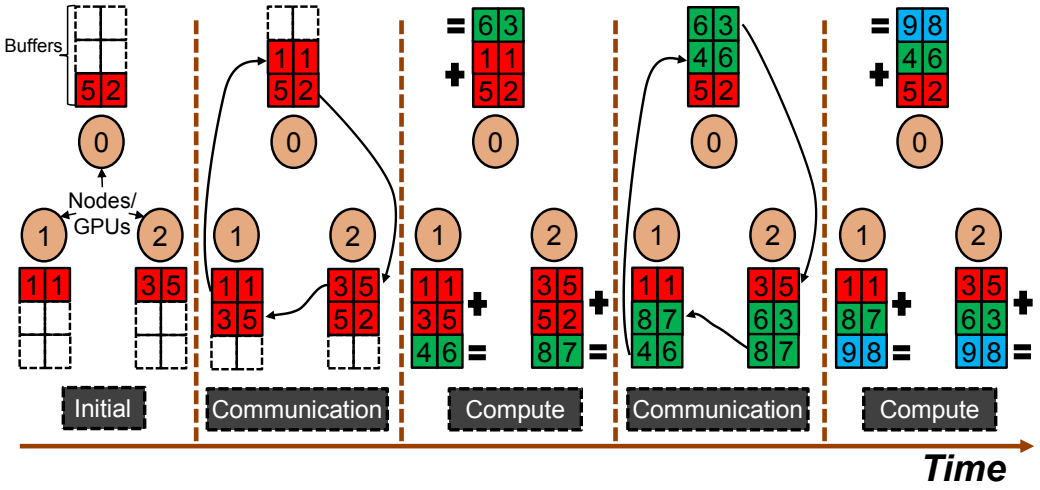



Figure 1.3: Allreduce algorithm on three nodes organized as a ring.

that the network operation be large enough to amortize the cost of splitting a kernel into pre-network and post-network pieces. The overheads also effectively negate the efforts of network interconnect providers, who have successfully reduced end-to-end wire latencies to 0.6$\mu$s at the time of this writing [66].

Kernel boundary networking is particularly problematic in strong scaling scenarios, where the addition of more nodes decreases the work per node and increases the number of ever smaller messages. Consider the case of the simple Allreduce operation illustrated in Figure 1.3, which is an important primitive in many distributed, GPU-accelerated machine learning applications [2]. In Allreduce, each GPU requires a piecewise combination of the vector present on every other GPU. Data is copied from one node to the next during the communication step, which is followed by a user-specified binary operation combining the data received from the network with a local buffer

in the computation step. At the end, every node has the final result of the reduction. As more GPUs are added to the collective operation with a fixed size input, the amount of work assigned to each GPU decreases and the number of rounds of communication increases. Eventually, the large overheads of entering and exiting a kernel between the computation and communication phases will dominate the runtime, even when there is enough parallelism to otherwise make GPUs attractive for accelerating the application.

## 1.2 Contributions

This dissertation explores new techniques to more tightly integrate GPUs with network adapters to allow efficient communication between GPUs across the network. It evaluates both hardware and software changes to NICs and GPUs to enable end-to-end, user-space communication between networks of GPUs, avoiding critical path CPU interference. The primary contributions can be broken down into the following three topics.

1. **Extended Task Queuing (XTQ)**: XTQ [59] defines an active messaging system enabling direct NIC-to-accelerator kernel dispatch. This scheme enables tightly integrated accelerators to efficiently and directly communicate with each other through a customized NIC. The target-side NIC participates in a tightly coupled queuing model and can directly schedule work to the accelerator, completely eliminating the CPU communication path.

2. **Command Processor Networking (ComP-Net)**: To accelerate networking, ComP-Net [57] uses a little-known feature of modern GPUs: embedded, programmable microprocessors that are typically referred to as Command Processors (CPs). GPU communication latency is decreased by running the network software stack on the CP instead of the host CPU. ComP-Net implements a runtime and programming interface that allows the GPU compute units to take advantage of the unique capabilities of a networking CP. Challenges related to the GPU's relaxed memory model and L2 cache thrashing are addressed to reduce the latency of network communication.

3. **GPU Triggered Networking (GPU-TN)**: GPU-TN [58] defines a high-performance mechanism by which the GPU can directly trigger a network operation on a NIC. In this approach, the host CPU is responsible for creating the network command packet on behalf of the GPU and registering it with the NIC. When the GPU is ready to send a message, it simply "triggers" the NIC using a memory-mapped, posted write operation. GPU-TN is implemented using only a small amount of additional complexity on a high-performance network adapter.

## 1.3 Thesis Statement

GPU networking can be improved by both software and hardware enhancements that enable GPUs to more directly interface with the network control plane.

## 1.4 Organization

The organization of this dissertation is as follows. Chapter 2 describes networking and GPU background information, along with the prior research work in the area of GPU networking. Chapter 3 describes the simulators, methodologies, and workloads used to produce data throughout the dissertation. Chapter 4 describes the Extended Task Queuing mechanism to efficiently launch kernels on remote GPUs. Chapter 5 describes how a GPU's command processor can be leveraged to increase GPU networking performance. Chapter 6 explores a mechanism for the host to pre-register network operations on a NIC that can be triggered by a GPU when the data is ready to be sent. Finally, Chapter 7 concludes the dissertation and suggests future work in the area.

# Chapter 2

# Background and Related Work

This chapter discusses relevant background information on high performance networks and GPU architecture. It also includes a summary and taxonomy of the prior art for GPU networking.

## 2.1 RDMA Networks

Modern high-performance clusters employ Remote Direct Memory Access (RDMA) adapters for lightweight and efficient inter-node data movement. RDMA can mean many different things depending on the context. For the purposes of this dissertation, the term RDMA refers to the direct transfer of data from an initiator node's memory to a target node's memory over a network. Once the communication has been scheduled at the initiator, no more effort from the CPU host is required to move the data until the operation completes. In many ways, RDMA extends the semantics of an intra-node DMA operation across the network.

Data movement from one node to another is accomplished using intelligent hardware built into NICs. While these technologies were previously only available in the highest-performance (and cost) systems, high-speed RDMA

fabrics can now be found on commodity systems. Technologies such as InfiniBand [47], iWARP [48], RDMA over Converged Ethernet (RoCE) [46], OmniPath [16], and Portals 4 [89] all offer high-quality RDMA solutions. Current technologies can reliably provide networking latencies as low as 0.6$\mu$s and throughput as high as 200Gbps [66].

The most important feature of high-performance networking for the purpose of this dissertation is the interface between the network hardware and the lowest-level system software. All high-performance NICs expose at least two types of queues to the system software. The first type of queue is for submitting commands to the NIC (e.g., send a buffer to node $x$ from address $y$). System software produces command packets that fully encapsulate the behavior of the transfer, which is evaluated by a state machine or more traditional computational pipeline on the NIC. The second type of queue is used to deliver notifications from the NIC to the host CPU (e.g., data received from node $x$ at address $y$). This queue is populated by the NIC itself and monitored by a low-level networking runtime or driver.

### 2.1.1 One-Sided Communication

From a programming model perspective, high performance networking is split into two main categories: one-sided and two-sided communication. Two-sided communication is the style typically favored by traditional message passing libraries. In two-sided communication, the sender and receiver must explicitly participate in the network communication by using paired *Send* and

*Receive* calls in the application. While this is easy to conceptualize, there are a number of limitations that can complicate network runtime implementations and degrade performance. Each *Send* and *Receive* typically implies a complicated set of synchronization operations between the sender and receiver before data is ever sent. The sender and the receiver must both coordinate buffer space and perform tag matching so that individual messages are routed to the correct buffer. Furthermore, it is common for paired *Send* and *Receive* calls to be temporally separated in the application, resulting in intermediate buffering at the receiver or a delay in the sender initiating a data transfer.

One-sided communication is an alternative to the paired communication model that seeks to overcome many of the previously described limitations by taking advantage of the capabilities of modern RDMA NICs. In one-sided communication, the initiator performs *Get* and *Put* operations that correspond very closely to the semantics of local *Load* and *Store* operations in a system operating under a relaxed memory consistency model. The target CPU does not need to explicitly participate, as regions of memory on the target are directly exposed to the initiator. Data transfer is allowed to proceed as soon as the network operation is encountered in the application with no intermediate buffering. One-sided communication also separates out synchronization from data movement, allowing a large number of network operations to complete before the application programmer requires any sort of synchronization. This abstraction allows for a much simpler software runtime and can increase performance for irregular workloads.

One-sided communications semantics serve as the cornerstone of the Partitioned Global Address Space (PGAS) style of parallel programming, where the global memory address space is logically partitioned and a partition is assigned to every process. There is a large number of languages and runtimes supporting a PGAS programming model, such as OpenSHMEM [12], GASNet [17], UPC [24], UPC++ [103], Chapel [22], GPI [34], X10 [19], and newer versions of the Message Passing Interface (MPI) [68]. Much of the work in this dissertation is heavily influenced by the semantics of PGAS languages and one-sided communication.

## 2.2 GPU Technology

While GPUs were originally designed to accelerate graphics workloads, researchers have been trying to use the hardware for non-graphics purposes dating back decades [43]. However, it wasn't until the introduction of user programmable vertex engines [62] that GPGPUs (General-purpose computing on graphics processing units) became mainstream and spurred a vast body of research [83]. The GPGPU paradigm uses the computational throughput and memory bandwidth of GPUs to greatly accelerate certain classes of data-parallel workloads. This section will describe the major components of GPU hardware, software, and supporting infrastructure that are relevant to this dissertation.

Unfortunately, there is no standard terminology when describing GPU architecture. The two major GPU vendors (AMD and Nvidia) use slightly

Table 2.1: AMD to Nvidia translator.

| Description | Nvidia | AMD |
|---|---|---|
| Single stream of execution and control flow | Thread | Work-item |
| Concurrently executing threads that share a vector ALU | Warp (32 Threads) | Wavefront (64 Work-items) |
| On-chip scratchpad memory | Shared Memory | Local Data Share (LDS) |
| Logical thread bundle for barrier synchronization and shared scratchpad memory | Thread Block | Work-group |
| Collection of all threads executing in a single kernel | Grid | NDRange |
| Vector ALU for processing wavefronts | CUDA Core | SIMD Unit |
| Collection of SIMD Units sharing a private L1 cache and wavefront scheduler | Streaming Multiprocessor (SM) | Compute Unit (CU) |

different verbiage to describe very similar concepts. This dissertation will use AMD's Graphics Core Next (GCN) architecture [7] and terminology throughout. However, all mentioned technologies have an equivalent component on Nvidia hardware. Table 2.1 provides a brief translation from AMD to Nvidia terminology for the most common concepts.

### 2.2.1 Architecture

Figure 2.1 illustrates the relevant components of a GPU. GPUs are comprised of a number of Compute Units (CUs), each of which are comprised of a collection of Single Instruction, Multiple Data (SIMD) units. Each CU

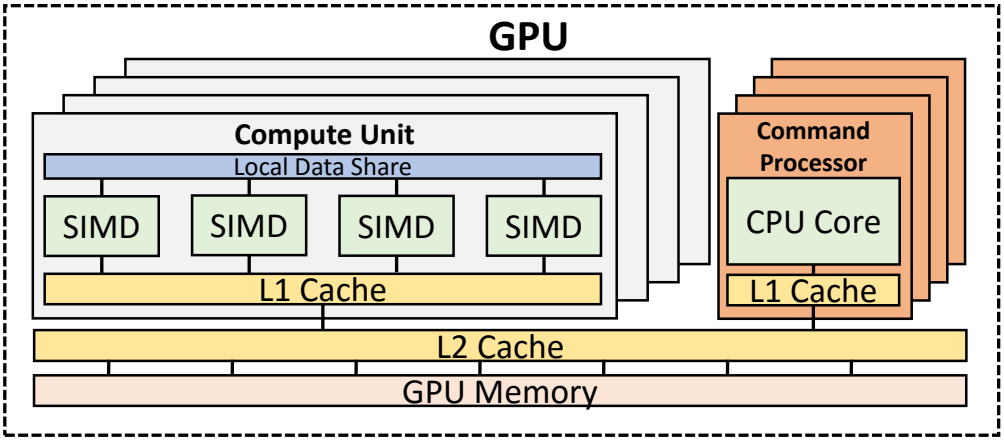



Figure 2.1: GPU architecture described using AMD terminology based on the Graphics Core Next (GCN) architecture.

is connected to a private L1 cache and shared L2 cache, which are not coherent and are maintained by explicit cache management instructions. Groups of work-items are dispatched on the compute units in bundles known as wavefronts. These wavefronts are further bundled into work-groups. Work-groups execute on the same CU and can therefore make use of per-CU scratchpad memory called the Local Data Share (LDS). Good performance on a GPU is achieved by providing a large amount of work and minimizing the amount of control-flow and memory divergence..

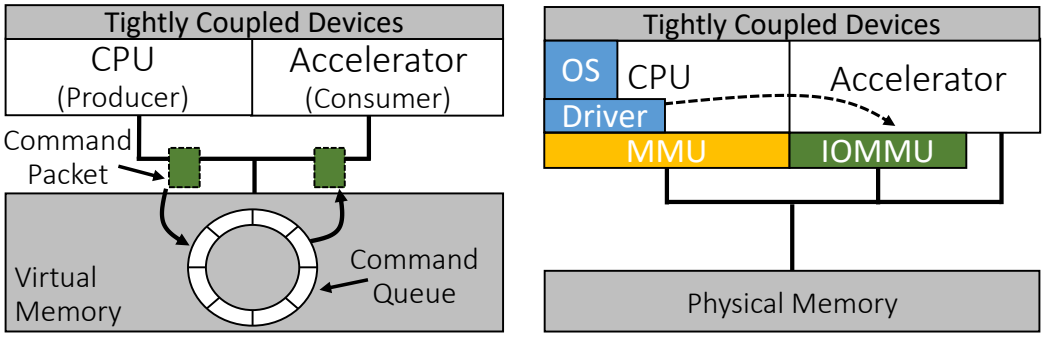
GPUs also contain a components known as the Command Processors (CPs). The primary responsibility of the CPs is to manage the scheduling, launch, and tear-down of GPU kernels by serving as an intermediary between the host CPU and the GPU's work-group scheduler. Each CP contains a private L1 cache that is connected to the same shared L2 cache as the CUs.

### 2.2.2 Programmability

GPUs are programmed by writing Single Instruction, Multiple Thread (SIMT) functions called *kernels*. Each kernel is written from the perspective of a single work-item. The number of work-items comprising a kernel and the number of work-items in a work-group are dispatch parameters controlled by the application programmer, but are subject to hardware limitations. Kernels are dispatched on the GPU using a vendor-provided runtime that may be directly visible to the application or hidden under a more general-purpose runtime. Kernels and launch parameters are communicated with the GPU using in-memory command queues, which are processed by the GPU's CPs.

### 2.2.3 Memory Consistency Model

The GPU operates using a weak, partially software-managed memory consistency model [44]. To share data between different threads on a GPU or other device, the programmer needs to use scoped synchronization operations. A scope defines the level of visibility that a synchronization operation applies to (e.g., local, device, and system). In this dissertation, it is assumed that a synchronization primitive can either be a release operation, which forces all previous memory operations from the current thread to be visible to the requested scope, or an acquire operation, which forces all memory operations below the acquire to see the most recent data from other threads at the requested scope. Essentially, these primitives map to cache maintenance operations and memory fences in the GPU hardware.




(a) HSA tasking model.

(b) Shared virtual memory.

Figure 2.2: Intra-node accelerator integration in HSA.

### 2.2.4 Intra-Node GPU Integration

As this dissertation focuses largely on system-level interactions between various devices on a node, it is important to briefly discuss how GPUs are integrated onto a compute node. Modern system designs are increasingly providing tighter coupling between CPUs, GPUs, and other accelerators. For the remainder of the paper, we will use the Heterogeneous System Architecture (HSA) [45] as the prototypical example of how GPUs are integrated with the rest of the system architecture. HSA is an open industry standard from the HSA Foundation, a consortium formed by AMD, ARM, Imagination Technologies, MediaTek, Texas Instruments, Samsung Electronics, Qualcomm, and others with the objective of helping system designers integrate different kinds of computing elements (e.g., CPUs and GPUs) to enable efficient data sharing and work dispatch. Some important features of the HSA specification are illustrated in Figure 2.2 and are described in the following paragraphs.

#### 2.2.4.1 User-Level Command Queuing

In HSA, applications allocate accelerator task queues in user memory. Devices fetch and execute tasks directly out of these queues, thereby eliminating OS kernel transitions and device-driver overheads on common paths. Figure 2.2a illustrates HSA user-level command queuing. User-mode queues are arranged as circular buffers, with the read and write pointers implemented as monotonically increasing indices. The queue entry format is defined by HSA's Architected Queuing Language (AQL). AQL packets contain all the information needed to launch and synchronize a GPU kernel or CPU function.

#### 2.2.4.2 Shared Virtual Memory

HSA requires that devices access memory using the same virtual addresses seen by the application program. This feature is necessary to allow users to pass pointers directly to devices through the HSA task queues without device driver intervention or validation. Devices must also be capable of initiating page faults to avoid the need to pin device-accessible pages in system memory. Shared translations can be provided to devices by an Input-Output Memory Management Unit (IOMMU) that references the same page tables as the host CPUs [5, 13], as shown in Figure 2.2b.

#### 2.2.4.3 Shared-Memory Synchronization

HSA uses memory-based signal objects for synchronization. Processes indicate to a device that work has been placed in its command queue using

a doorbell signal associated with the queue. The doorbell signal can map to a memory-mapped device range (e.g., for a firmware- or hardware-dispatched device such as a GPU), or to a shared-memory location (on which a software-dispatched device such as a CPU can poll). Devices or threads can also wait on tasks to finish using these in-memory completion signals.

#### 2.2.4.4 Intra-Node GPU Networks

Unlike GPU communication across nodes, communication between multiple GPUs residing on the same node have been highly optimized by GPU vendors. Nvidia's version of this technology is known as NVLink Fabric [77]. NVLink allows for low latency, high bandwidth access from one GPU to another using loads and stores from within a kernel, as well as supporting more traditional host-initiated DMA transfers. Much of the work in this dissertation is towards bringing the performance and ease of use of technologies like NVLink across more traditional multi-node networks.

## 2.3 Related Work

This dissertation builds upon a number of closely related technologies. This section describes the state of the art in industry today and some academic research in the area of GPU networking. A taxonomy of different networking approaches is also described that will be used to refer to groups of related works collectively throughout the dissertation. Figure 2.3 provides a visual representation of each style of GPU networking discussed in this section, and

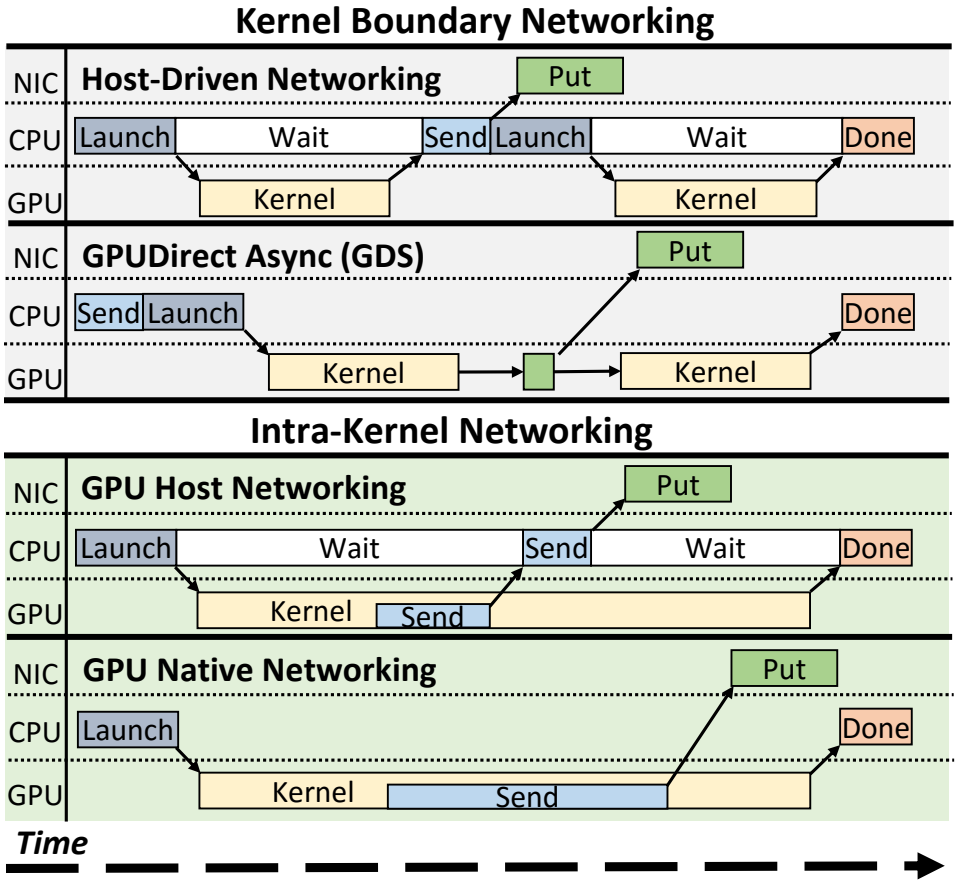



Figure 2.3: Overview of the control flow of different networking strategies on the GPU.

Table 2.2 provides a breakdown of the characteristics and overheads common to the different approaches to GPU networking.

### 2.3.1 Inter-Kernel Networking Optimizations

Most early GPU networking research and all currently available industry work involve optimizing GPU communication at kernel boundaries. This style of kernel-boundary GPU communication will collectively be referred to as

Table 2.2: Taxonomy and comparison of prior art in GPU networking.

| Networking Strategy | Kernel Boundary | GPU Triggered | GPU Overhead | CPU Overhead |
|---|---|---|---|---|
| Host-Driven Networking [64, 29, 93] | Yes | No | - | Network Stack |
| GPU Native Networking [78, 79, 52, 53, 25] | No | Yes | Network Stack | - |
| GPU Host Networking [51, 94, 36, 67, 82] | No | No | CPU/GPU Queue Management | Service Threads, Network Stack |
| GPU Direct Async (GDS) [87] | Yes | Yes | Network Trigger | Partial Network Stack |

***Host-Driven Networking*** (HDN) in this dissertation, since the host directs the networking operations at kernel boundaries.

In academia, Zippy [29] and Compute Unified Device and Systems Architecture (CUDASA) [93] were some of the earliest works in this area. Both expose GPU communication using a PGAS programming style, where communication is performed at kernel boundaries on the CPU itself using custom runtime extensions wrapped around MPI.

Industry has proposed two optimizations to accelerate GPU networking at kernel boundaries. The first technology is a collaboration between Mellanox and AMD/Nvidia, which enables direct peer-to-peer data copy of buffers on a discrete GPU's memory to a NIC without the need for bounce buffers through host memory. On Nvidia hardware, this technology is known as GPUDirect RDMA [64], and on AMD hardware, it is known as ROCn RDMA [9]. Network runtimes such as CUDA-aware OpenSHMEM [37] take advantage of GPUDi-

rect RDMA features to optimize data movement for one-sided communications.

Recently, Nvidia has proposed an update to their baseline GPUDirect technology, known as GPUDirect Async (GDS) [87]. GDS goes so far as to allow the GPU to initiate pre-registered network messages by ringing a doorbell on the NIC. In the GDS model, the CPU posts a network operation and interleaves network initiation between kernel invocations inside of Nvidia CUDA [73] streams. The CP evaluates the stream and rings a doorbell on the NIC when a kernel has completed.

### 2.3.2 Intra-Kernel Networking Optimizations

Recently, a number of academic research efforts have attempted to make networking within a GPU kernel possible. Current intra-kernel networking mechanisms can largely be broken down into two classes depending on their design. A brief overview of each type is presented here.

#### 2.3.2.1 GPU Host Networking

The first intra-kernel networking approach, which will be referred to as ***GPU Host Networking***, defines a lightweight, GPU-optimized interface between the GPU and the CPU. The GPU writes the payload to a bounce buffer and hands it off to the CPU. The CPU performs the heavy lifting of creating a network-compatible command packet pointing to the provided buffer before handing it off to the NIC.

A number of research projects implement intra-kernel networking us-

ing this technique. FLAT [67] allows for the automatic generation of CPU MPI codes from GPU kernels using custom compiler extensions. Distributed Computing for GPU Networks (DCGN) [94] exposes an MPI-like interface for GPU kernels to pass messages to GPUs on remote nodes. CPU helper threads perform communication on behalf of the GPU by tunneling requests through standard MPI. GPUNet [51] provides a socket-based abstraction for the GPU, and also uses CPU helper threads to perform the actual communication. The dCUDA [36] model implements a GPU networking programming model that attempts to hide long latency GPU network events across the cluster. Gravel [82] optimizes irregular GPU messaging applications by employing host-side coalescing of network operations. Gravel is unique among these works in that it focuses solely on APUs (SoCs with both GPUs and CPUs on the same die).

#### 2.3.2.2 GPU Native Networking

The second intra-kernel networking approach, which will be referred to as ***GPU Native Networking***, constructs a networking stack on the GPU itself. GPU scratchpad memory and persistent kernels (i.e., kernels that last for the entire duration of the program) are used to hold network and connection state, allowing the GPU to communicate with the NIC without any intervention from the CPU.

Similar to GPU Host Networking, there has been some exploration of this technique in academia. GPU Global Address Space (GGAS) [78] ex-

plores adding custom hardware in the GPU to support a cluster wide global address space, where GPUs can communicate with each other through simple loads and stores. Oden *et al.* [79] explore implementing InfiniBand entirely on a GPU runtime, with mixed results. However, additional work by the same research group illustrates much more favorable performance [52, 53]. GPUrdma [25] also implements InfiniBand directly on the GPU, although limitations of current GPU hardware can cause correctness problems under high load. NVSHMEM [85] provides an OpenSHMEM-like interface to perform one-sided communication from within a kernel, but is currently limited to a single node. Agostini et al. [3] describes several implementations of GPUDirect Async, one of which of offers similar intra-kernel networking semantics to GPUrdma.

### 2.3.3 Active Messaging and Message Passing Machines

One technique used in this dissertation to improve the performance of GPU networking is based on the concept of active messages. The seminal Active Messages work [27] embeds computation in network messages by directly invoking a user message handler on the target. Several machines proposed in the '90s coupled light-weight tasks with explicit message passing directly in hardware, including the J-Machine [30], M-Machine [71], Star-T Voyager [10]; or some combination of messaging passing and shared memory such as MIT Alewife [1], Typhoon [86], and FLASH [38]. More recently, Besta and Hoefler [14] modify the IOMMU to invoke active-message-like handlers as a side

effect of RDMA *Put* and *Get* operations.

A number of low-level libraries have since implemented active messaging semantics. Willcock et al. [101] developed AM++, which extends low level active messaging primitives with a type-safe, generic programming model. IBM's Deep Computing Messaging Framework (DCMF) [54] and their Low-level Application Programing Interface (LAPI) [92] provide active messaging support for the application programmer on IBM systems. Perhaps the most well-known runtime library to implement active messaging support is GASNet [17]. GASNet is a low-level, portable runtime that is used to implement many PGAS languages. The lowest level of the GASNET core API is designed largely around optimized active message routines.

# Chapter 3

# Methodology

This dissertation relies on cycle-level simulation and power modeling along with some targeted studies and projections from real hardware. This chapter provides an outline of the infrastructure, methodology, and workloads that are used to evaluate the usefulness of the proposed GPU networking optimizations.

## 3.1 Simulation Infrastructure

The open-source gem5 simulator [15] is used for all simulation results reported in this dissertation. The gem5 simulator is a publicly available cycle-level simulation infrastructure commonly used for computer architecture research. It comes with a number of models for commonly used components, such as an out-of-order CPU model and memory model.

In addition to the baseline simulator, the AMD GPU compute model [6] is used as the baseline GPU simulation environment. The AMD baseline GPU model runs an intermediate language called Heterogeneous System Architecture Intermediate Language, or HSAIL. However, recent work has shown that GPU simulators should run their native ISA [35], not an intermediate language,

to replicate the performance of real GPUs. Therefore, the infrastructure used in this work was enhanced to support direct execution of AMD's GCN3 [7] ISA.

The infrastructure simulates multi-node configurations with a simple switch and wire delay model in a star topology. The Portals 4 network programming API is used as the low-level network interface for all simulation results. Portals 4 is a connectionless low-level network API designed to support MPI and various partitioned global address space (PGAS) languages. It is agnostic to the underlying physical layer and exposes both flow-control and RDMA data transfer to higher-level applications and libraries.

One unique capability of Portals 4 is the ability for the NIC to delay the processing of a network command until a certain trigger condition is met. This mechanism, known as triggered operations, was designed primarily as a means of accelerating collective operations [98]. Using triggered operations, it is possible for low-level networking runtime to provide a network command to the NIC which is only executed when a certain message or messages are received at the target. Triggered semantics are used in one part of this dissertation to improve the performance of GPU networking.

Figure 3.1 shows the overall infrastructure, which models an APU-style system. The CPU-side has a traditional 3-level cache hierarchy with a mostly-exclusive policy in the L3. On the GPU-side, each CU shares an instruction cache (I-cache) and a scalar cache (K-cache) with four other CUs and has a private data cache (D-cache) for vector accesses. The GPU's L1 caches are

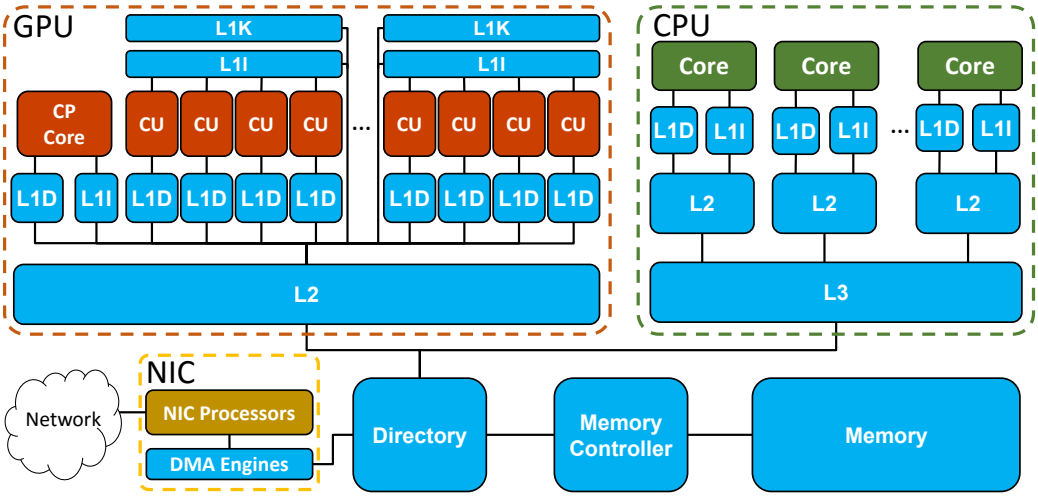



Figure 3.1: Overview of a single node of the simulation infrastructure.

not automatically coherent through hardware; they are managed by flush and invalidate instructions in the GPU ISA. Both the L1 data cache and the SQC are hooked up to a shared L2. In the APU configuration, the GPU's L2 cache uses a write-through policy and receives probes from a system directory, which keeps the L2 cache coherent with other caches in the system. The GPU module also contains a number of CPs with private L1 data and instruction caches connected to the GPU's L2. Both the GPU Last Level Cache (LLC), CPU LLC and NIC are connected to a coherent system directory.

Some experiments in this dissertation describe a discrete GPU (dGPU) baseline. For these experiments, the model is altered to statically separate the memory pool into discrete GPU memory and CPU memory with separate memory controllers. When a device tries to access memory that is not local to it (i.e., CPU accesses dGPU memory or dGPU accesses system memory), the

simulator uses a simple LogP [23] based model to force accesses to experience a latency, bandwidth, and message rate in accordance with PCIe gen4 x16 latencies and bandwidths. None of the lower-level characteristics of the PCIe transaction or physical layer are modeled, however, similar LogP-based analytical models for networks have been shown to be quite accurate [42]. Also, in the dGPU configuration, the GPU's L2 cache no longer writes-through or responds to probes from the system directory. Therefore, the GPU's L2 cache is kept coherent using GPU ISA instructions, similarly to the L1 caches.

While the previous description serves as a baseline for all simulation results, each chapter will employ the simulation environment in a slightly different manner. The precise details and configurations for all the systems under test will be presented before the results are discussed.

### 3.1.1 Power Modeling

CPU power modeling is performed using the McPAT [60] power modeling tool. McPAT is a multi-core power estimation tool that uses models derived from CACTI [102] to estimate the power consumption of various SRAM blocks and accompanying control logic in the core and caches. Results from gem5 simulations are fed into McPAT to obtain energy and power estimates. For this dissertation, McPAT is configured to operate at the 22nm technology node. No GPU power numbers are collected or reported.

## 3.2 Workloads

The techniques presented in this dissertation are evaluated across a number of GPU microbenchmarks and workloads. This section provides a basic overview of these workloads. Later sections will describe modifications to many of these workloads as part of the evaluation and introduce microbenchmarks specifically targeted to the proposed networking optimization.

### 3.2.1 Reduce and Allreduce

*Reduce* and *Allreduce* are collective operation that use a binary operation (e.g., SUM, PROD, or MAX) to combine the corresponding elements in the input buffer of each participating process. For Reduce, the combined results are stored in a result buffer in the root process's address space. For Allreduce, the combined results are stored in the result buffers of all participating processes. Allreduce was described previously in Figure 1.3, where it was used to motivate the need for efficient GPU networking.

### 3.2.2 Accumulate

The *Accumulate* function is a one-sided operation used to combine initiator-resident data with target-resident data through a specified operation. The target data is replaced with the result without any explicit participation of the target process.

Accumulate operations are preceded by a collective window creation operation during which the target process makes a portion of its memory space

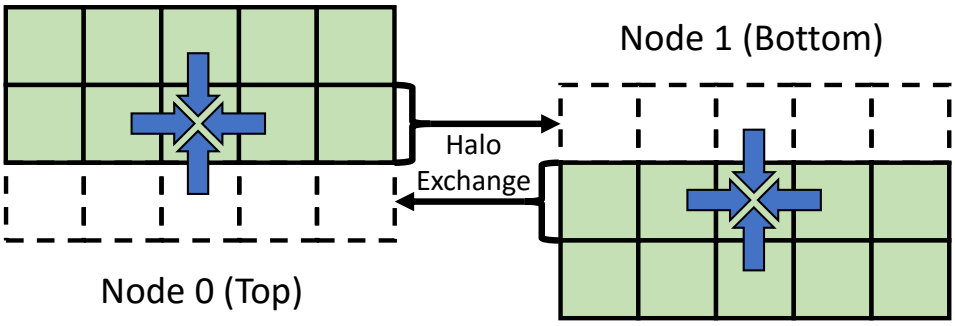



Figure 3.2: Stencil pattern and halo exchange example.

available to other members in the window for direct updates. Subsequent accumulate calls to the target are bracketed by synchronization operations which define access epochs. An initiator process can make multiple accumulate calls to a target within an epoch. However, the accumulate operations are considered complete only after the synchronization operation that closes the epoch.

### 3.2.3 Jacobi Stencil

Multi-dimensional stencils are an extremely common communication pattern for multi-GPU application. This dissertation uses the Jacobi relaxation problem [55] to represent the iterative stencil model of GPU communication. Jacobi is a method used to determine solutions of a diagonally dominant system of linear equations. Figure 3.2 shows the basic pattern of a stencil computation like Jacobi decomposed over two nodes. Each node has a portion of the input matrix, which is divided in multiple dimensions (only 1D in the example) over all participating nodes. On every timestep of the calculation,

Table 3.1: Microsoft Cognitive Toolkit networking behavior.

| Name | Domain | % Blocked | # Reductions |
|---|---|---|---|
| AlexNet | Classification | 14% | 4672 |
| AN4 LSTM | Speech | 50% | 131192 |
| CIFAR | Classification | 4% | 939820 |
| Large Synth | Synthetic | 28% | 52800 |
| MNIST Conv | Text Recognition | 12% | 900000 |
| MNIST Hidden | Text Recognition | 29% | 900000 |

each entry in the matrix is updated based on the value of its neighbors. In the example, every entry is dependent on the values of its four immediate neighbors. At the end of a timestep, the entries on the edge of the matrix are exchanged with each nodes' direct neighbors in a process known as a halo exchange (in a 2D decomposition, the "ghost" entries at the edge form a halo around the matrix). This pattern of computation and communication continues until an iteration bound has been reached, or the residuals from the latest computation falls below a user-defined threshold.

### 3.2.4 Machine Learning

Machine learning is an important class of workload that frequently uses clusters of GPUs to accelerate the training of neural networks. Neural networks are typically trained using some form of iterative stochastic gradient descent (SGD) for a fixed number of training epochs, or until some convergence criterion has been satisfied. In the distributed formulation of SGD, an Allreduce operation is used to transfer and combine the contents of every GPUs' gradient matrix to every other GPU. This gradient Allreduce opera-

tion has been shown to be a significant bottleneck in deep learning workloads, especially those operating in the synchronous training mode. This dissertation uses six machine learning workloads from a variety of application domains from the Microsoft Cognitive Toolkit [2]. A brief description of the networking behavior of each workload is presented in Table 3.1. For the data presented in the table, the workloads are run on a small cluster of 4 nodes each containing one Nvidia GPU. The networks are trained in a synchronous training mode, where the computation blocks when performing a gradient Allreduce. The *%Blocked* heading refers to total time spent blocked on an Allreduce operation, and *Reductions* refers to the total number of reduction calls. This data illustrates that improving GPU networking performance has the potential to significantly speed up the application.

Unfortunately, these machine learning workloads are too big to run entirely on the simulator. Therefore, the collected results are projected from simulator results combined with real hardware runs using the following methodology. First, profiling data is collected from the Stampede [95] supercomputer, where GPUs were used for local Stochastic Gradient Descent, InfiniBand NICs were used for communication, and the CPUs were used for the computation in the Allreduce phase. By combining these real-world runs with detailed simulation of Allreduce operations obtained from gem5, it is possible to project the performance improvement of accelerating the Allreduce function with the various techniques presented in this dissertation. Since the Microsoft Cognitive Toolkit uses blocking Allreduce calls, it is not necessary to worry about

any communication and computation overlap.

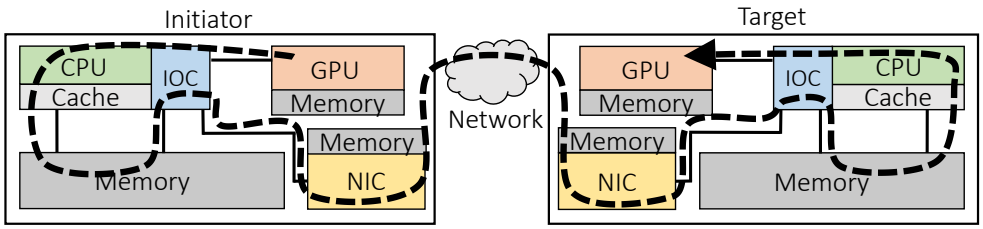
# Chapter 4

# Extended Task Queuing: Active Messages for Heterogeneous Systems

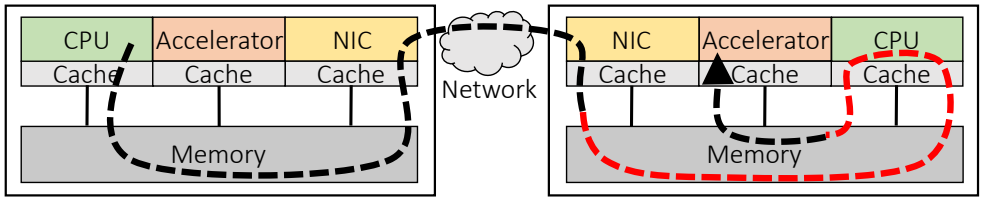
Many of the major heterogeneous platform providers are striving to incorporate accelerators more tightly into a node's compute ecosystem [73, 45, 80]. Most of the major hardware vendors offer a number of standardized features that enable accelerators to participate in computation as peers to the host CPU. These features typically include user-mode task invocation, shared virtual memory with a well-defined memory consistency model, shared-memory-based synchronization, and accelerator context switching. Frameworks that implement the above features at a node level will be referred to as *tightly coupled* compute ecosystems.

While node-level, tightly coupled frameworks remove data copies and heavyweight driver invocations for intra-node accelerators, NICs with Remote Direct Memory Access (RDMA) offer these same benefits for inter-node data transfers. RDMA-capable NICs and fabrics [47, 16, 89, 46, 48] move data from one node to another without involving the target-node CPU by enabling the target-node NIC to perform DMA directly to and from application memory.

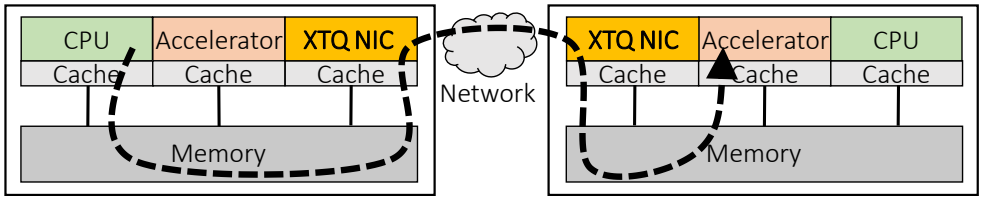
This chapter introduces NIC primitives which combine RDMA with




(a) Conventional heterogeneous systems




(b) Tightly coupled heterogeneous systems




(c) XTQ-enabled, tightly coupled heterogeneous systems

Figure 4.1: Remote task enqueue control path on different heterogeneous, distributed-memory systems.

tightly coupled, user-level task queuing. These primitives enable applications to efficiently enqueue tasks on any compute device in a distributed-memory system, without involving the target-node CPU or the operating system on either the initiator or the target node. This mechanism is called Extended Task Queuing (XTQ) [59], since it extends the lightweight, user-mode task queuing in modern shared-memory platforms across distributed memory systems. XTQ offers a highly efficient active messaging [27] platform for accelerators that improves upon the state of the art.

Figure 4.1 shows the control path of a point-to-point remote task invocation implemented on three types of systems. In these examples, a CPU on the initiator node schedules work on a remote accelerator. In a conventional heterogeneous node, the communication flow is similar to Figure 4.1a. The initiator CPU uses a high-performance NIC sitting on an I/O bus to transfer the task and associated data to the target via RDMA. While emerging technologies such as Nvidia's GPUDirect RDMA [64] allow for NICs to transfer data directly to a GPU's on-board memory, launching a kernel still requires the intervention of the target CPU's runtime and kernel driver.

Figure 4.1b shows the same operation implemented on a contemporary, tightly coupled SoC. The CPU and the accelerator share the same memory, obviating the need to transfer data from the target's main memory to the accelerator's local device memory. However, the target-side CPU must still service the request from the NIC and explicitly schedule work on its local accelerator.

XTQ provides a mechanism enabling the direct CPU-to-accelerator communication presented in Figure 4.1c. This scheme enables tightly integrated accelerators to efficiently and directly communicate with each other through a customized hardware NIC. The target-side NIC participates in a tightly coupled queuing model and can directly schedule work to the accelerator, completely eliminating the CPU communication path in Figure 4.1b.

Directly interfacing an intra-node tasking framework with inter-node RDMA through XTQ offers a number of benefits, including:

- **A unified active messaging framework for all compute devices in the system.** By leveraging the user-mode task invocation of tightly coupled systems, it is possible to design an active messaging framework that uses the same interface to spawn remote tasks on any device (CPU, GPU, FPGA, Processor-in-Memory (PIM), etc.) in the system. Unified active messaging offers exciting new acceleration possibilities for future applications and runtime libraries.

- **A reduction in remote accelerator task launch latency.** RDMA provides the means for low-latency, CPU-less data transfer without redundant data copies. Tightly coupled architectures provide the means for lightweight, direct accelerator-to-accelerator communication within a shared-memory node. By marrying the two, a target-side NIC can schedule work directly on a local accelerator without critical-path software on the CPU. Bypassing the CPU decreases accelerator launch latency and opens up the possibility of fine-grained remote tasking models for accelerators.

- **Removal of message processing and task launch overheads on the target CPU.** Message progress threads can impose a significant overhead in distributed systems [40]. This problem is exacerbated when the progress thread not only has to handle messages, but also construct command packets and schedule work on accelerators. Direct NIC-to-accelerator task invocation frees the CPU to either perform more useful computation or to enter a low power state.

This chapter provides an overview of an XTQ-enabled, tightly coupled system architecture. The exploration of XTQ is organized into the following three topics:

- **The NIC hardware design.** Cross-node heterogeneous integration can be achieved with the addition of a small amount of hardware to an RDMA-capable NIC.
- **Programming via lightweight extensions to an RDMA-capable host API.** The XTQ remote tasking primitive is implemented as a simple extension to a generic RDMA network programming interface. This API extension leverages one-sided communication semantics to allow programmers to schedule active messages on any computing device in the cluster. It is easy to express XTQ task invocations using only a few API calls.
- **Performance on a number of important primitive operations and machine learning applications.** XTQ can improve GPU-bound active message performance by 10-15% over non-XTQ enhanced messages, while eliminating message and task enqueue overheads on the target-side CPU. Latency decompositions for important steps in the XTQ task flow and performance improvements for microbenchmarks are shown. XTQ can also enhance important MPI primitives such as Accumulate, Reduce, and Allreduce operations. XTQ benefits scale as the

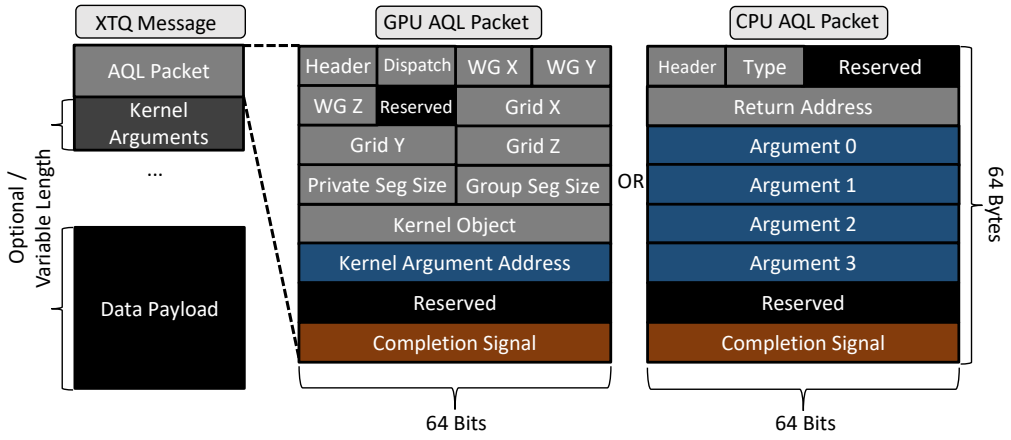



Figure 4.2: XTQ message format.

number of nodes increases for a fixed problem size. Finally, XTQ can improve performance on distributed deep learning workloads implemented using the Microsoft Cognitive Toolkit [2].

## 4.1 Architecture

This section illustrates the main components of the XTQ hardware architecture. XTQ introduces one basic primitive to an RDMA-capable NIC: direct, user-mode task invocation on a remote accelerator. The lightweight hardware that implements this operation is described in the following paragraphs. For the purposes of the exploration of XTQ, the descriptions will refer specifically to a system with tightly coupled CPU and GPU. However, the same scheme is generalizable to other accelerators in a tightly coupled system architecture.

### 4.1.1 Message Format

Figure 4.2 illustrates the main components of a typical XTQ message along with the AQL-like [45] command packet formats for CPU and GPU tasks. For GPU tasks, the command packet contains all the information needed to launch a kernel, such as a pointer to the kernel code object, a pointer to the kernel arguments, and work-item/work-group sizes. Variable-sized kernel arguments are appended to the message at the end of the command packet. For CPU tasks, the command packet contains fields such as a function pointer and embedded scalar arguments.

After the command packet and kernel arguments is a variable-size payload. This payload is generally a task input buffer provided by the initiator, but there are no specific requirements for its usage. The API and NIC consume two separate pointers for the command packet/kernel argument combination and the payload. The NIC performs a gather operation on the two buffers before transmitting to the target. Gathering the payload and command packet separately avoids an unnecessary memory copy in the application code.

### 4.1.2 Remote Task Dispatch

This section illustrates the steps involved for a CPU to schedule a unit of work on a remote GPU. The first step in a remote task invocation is for the initiator's CPU to enqueue one or more remote *XtqPut* operations on the NIC's command queue. The NIC's software interface is provided pointers to two distinct memory buffers: one for the command packet and kernel/function

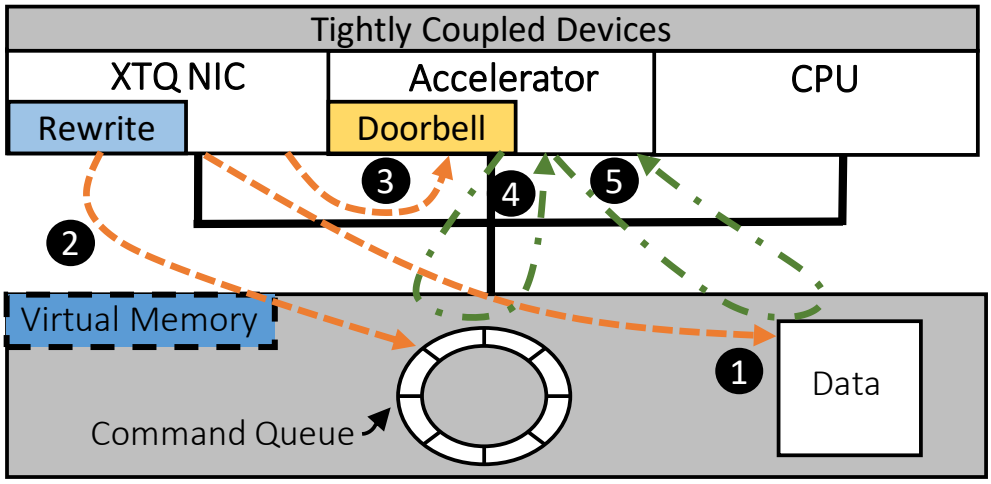



Figure 4.3: Target side steps in *XtqPut* operation.

arguments and one for the optional data payload. The host library notifies the NIC of the local memory buffers through a shared command queue and doorbell mechanism. The NIC then performs a local gather operation and transfers data over the network.

Figure 4.3 illustrates the steps involved in queuing a task on the target GPU from the NIC. First, the target NIC receives the XTQ message from the network. The payload portion of the message is streamed directly into the receive buffer in main memory ❶. The NIC also extracts the command packet from the message and performs the rewriting services described in Section 4.1.3. Before enqueuing the packet, the NIC first accesses the command queue descriptor. If the queue is full, then the NIC triggers the flow control mechanism discussed later. Otherwise, the rewritten command packet is enqueued to the target GPU's command queue ❷. After the payload write and

command enqueue is completed, the NIC writes the command queue index to the GPU's memory-mapped doorbell register ❸. In the presented configuration, the GPU contains a Command Processor (CP), which is responsible for reading packets from the command queue when the doorbell is updated with the newest write index. The CP proceeds to dequeue the packet from its command queue ❹, decodes the launch parameters, and schedules the work on the GPU's compute threads. The GPU threads then perform global load and store operations to access the kernel arguments and payload data ❺. Optionally, the GPU can notify the local CPU of kernel completion using a shared-memory signal (not shown).

It is important to note that all of the node-local operations take place in a unified virtual memory environment. The pointer addresses used for the command queue and data buffer are virtual addresses. In this scheme it is assumed that both the NIC and GPU have access to an IOMMU as described in Section 2.2.4.

Security and process isolation are critical concerns for distributed, multi-process workloads. For the XTQ extensions, the security concerns are handled by the underlying transport layer. As an example, the Portals 4 network programming API provides clear semantics for isolation of its own per-process data structures. To the first order, an XTQ extension to the Portals 4 framework would inherit these same security mechanisms to protect its task queues and other auxiliary data structures. A more thorough treatment of the impact of a system like XTQ on the security model of the cluster is left as future work.

A similar argument exists for flow control. XTQ utilizes shared-memory queues as the interface between the NIC and the GPU, which can become full. XTQ can utilize any existing hardware transport-level flow-control mechanism. In the continuing example with Portals 4, the philosophy is to provide building blocks for a higher software layer to implement arbitrarily sophisticated policies. Portals 4 can identify when target-side resources are full and generate events to notify the initiator or target that a message was not successfully delivered. The same monitoring and messaging facilities are used for the XTQ extension to Portals 4.

### 4.1.3 Rewrite Semantics

One issue with remote task invocation is that the initiator is unaware of the addresses of important resources which are dynamically allocated by the target. Even if all machines have an identical operating system and execute the exact same code, security techniques such as Address Space Layout Randomization (ASLR) [84] can provide different virtual addresses for both static and dynamically allocated variables on different machines. Some examples are target-resident kernel input/output buffers, completion signals, and the GPU's command queue. One simple way to solve this issue is to broadcast the virtual addresses of all data needed for task execution. However, it is costly for initiators to keep track of resource descriptors for thousands of possible target nodes; this scenario is particularly germane in scenarios when resources are frequently allocated and deallocated during program execution.

To solve this problem, XTQ draws inspiration from CPU-side active messaging frameworks such as GASNet [17]. In GASNet, the address of request handlers may be at different address on each node, however, all nodes have the same number of handlers registered in the exact same order. Therefore, active messages can refer to a particular handler using an index, and the implementation at the target simply needs to use this index to offset into a table of registered handlers. XTQ takes this a step further and leverages coordinated indices to refer to all target-resident data. The initiator populates the command packet with these indices, and the target performs a translation from an index to the correct target-local virtual address. Therefore, the initiator does not need to store any target address information, and the target only needs to keep index translations for its own resident data. Since one of the design goals is to avoid invoking the CPU on tasks that are not specifically destined for it, the XTQ framework incorporates the logic to substitute virtual addresses into the target-side NIC. The semantics of this "XTQ rewrite" operation will be described in detail for both CPU and GPU bound tasks. The term "rewrite" is used as opposed to the arguably more accurate term "translation" to avoid confusion with virtual to physical address translations.

#### 4.1.3.1 Lookup Tables

The NIC manages a number of per-process lookup tables to hold the index-to-virtual address rewrites needed for the NIC to enqueue tasks on a compute device. There are three different types of lookup tables: the Kernel

Lookup Table, Function Lookup Table, and Queue Lookup Table. Every XTQ packet will perform one lookup in either the Kernel or Function Lookup Table depending on the type of the packet. Additionally, all messages will trigger a lookup in the Queue Lookup Table to extract the base pointer of the target command queue. The entries in the lookup tables are populated by the host CPU using the XTQ API described in Section 4.2.

One lookup table entry is needed for each function, kernel, and queue that wishes to participate in XTQ's direct NIC-to-compute device tasking. For the applications and microbenchmarks that were studied, 64 kernel and function registrations were sufficient, producing Kernel and Function Lookup Tables that are around 4KB per process. For a small number of nodes, these data structures can be resident on dedicated tables on the NIC. This is the approach taken by the prototype implementation of XTQ evaluated in Section 4.3. For larger numbers of nodes, these tables would need to reside in system memory. In this case, the NIC can implement an on-chip cache to provide low-latency access to frequently used table entries. The design of such a cache structure is strongly dependent on the characteristics of the application and is left as future work.

#### 4.1.3.2 Rewrite Procedure

Figure 4.4 shows how the NIC selectively replaces certain fields in a GPU AQL packet. The initiator places a lookup table index in the field reserved for the kernel object pointer. The target NIC uses this index to offset

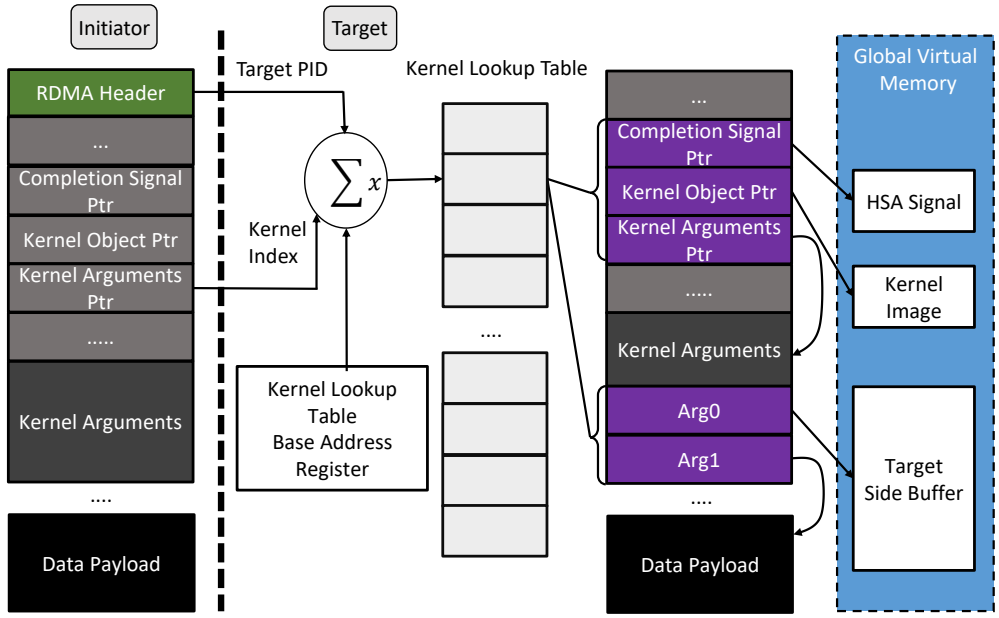



Figure 4.4: Target-side XTQ rewrite semantics.

into the Kernel Lookup Table to replace the kernel pointer with the correct value in the target's virtual address space. The actual kernel arguments are aligned directly after the command packet in the receive buffer. XTQ replaces the kernel argument pointer in the command packet with the address at which the kernel arguments will be written to memory. Finally, the first two kernel arguments are replaced with a pointer to a pre-registered target side buffer and the initiator-provided data payload, respectively. A pointer table can be registered instead of a target-side buffer if more registrations are needed. Kernel Lookup Table entries also contain room for the registration of a target-resident, shared-memory completion signal. When the GPU finishes execution of a task, this completion signal is decremented by the CP to let other agents

efficiently wait for the task to complete. XTQ replaces the command packet completion signal entry with the completion signal registered in the Kernel Lookup Table if it is valid. The rest of the fields are passed through as they are received and are assumed to be properly populated by the initiator.

A similar rewrite procedure is performed for CPU tasks. For CPU tasks, the NIC references the Function Lookup Table instead of the Kernel Lookup Table. The primary difference is that the first four function arguments are embedded directly in the AQL packet.

Finally, the NIC must identify in which of many possible user-mode queues to place the AQL packet. The queue address is extracted with one final table lookup, using an index embedded in the AQL reserved bits to access the Queue Lookup Table.

## 4.2 Programming Model

The XTQ tasking framework defines an API that the host CPU can use to schedule remote tasks on a target compute device. This API is implemented as an extension to a generic RDMA network programming interface that supports a remote *Put* operation. The XTQ-specific extensions can be broken down into the remote task launch function (*XtqPut*) and a number of registration functions used to populate the XTQ lookup tables.

### 4.2.1 *XtqPut* Function

XTQ contains one remote-tasking operation: *XtqPut*. The *XtqPut* operation performs the same one-sided, RDMA data movement operation as a basic *Put* operation, with additional semantics for launching tasks on the target. The exact mechanism for launching tasks is described in detail in Section 4.1.

### 4.2.2 Lookup Table Registration

The XTQ lookup tables are populated by the host CPU using a registration API. The API provides three lookup table registration functions:

- *XtqRegisterFunction:* Associates a lookup table index with a function pointer and an optional target resident buffer.
- *XtqRegisterKernel:* Associates a lookup table index with a kernel pointer, an optional target resident buffer, and an optional completion signal.
- *XtqRegisterQueue:* Associates a lookup table index with a command queue descriptor.

These functions are meant to be invoked using globally known, coordinated indices, as discussed in Section 4.1.3. Such indices are common in SPMD programming techniques and are already available in distributed, multiprocess programming frameworks such as MPI.

```
int main(int argc, char *argv[]) {
    // ❶ Initialize RDMA comm layer
    int rank = RdmaInit();
    int index = 42;
    if (rank == INITIATOR) {
        // ❷ Construct XTQ payload and command
        void *payload = malloc(BUFFER_SIZE);
        void *cmd = ConstructCmd(CMD_SIZE, 42);
        // Post initialization sync with target
        ExecutionBarrier();
        // ❸ Launch on remote GPU using XTQ
        XtqPut(TARGET, cmd, CMD_SIZE,
                  payload, BUFFER_SIZE);
    } else {
        // ❹ Post receive buffer
        void *recv_buf = malloc(BUFFER_SIZE);
        RdmaPostBuffer(recv_buf);
        // ❺ Initialize HSA CPU Runtime
        signal_t signal;
        kernel_t kernel;
        queue_t queue;
        TaskingInit(&signal, &kernel, &queue);
        // ❻ Register Kernel/Queues
        XtqRegisterKernel(signal, kernel, 42);
        XtqRegisterQueue(queue, 42);
        // Post initialization sync with initiator
        ExecutionBarrier();
        // ❼ Wait for GPU to complete task
        SignalWait(signal);
    }
}
```

Figure 4.5: Pseudocode for *XtqPut* operation.

### 4.2.3 Example Program

To ground the discussion of the API, Figure 4.5 illustrates a small program written in the SPMD style utilizing the primary components of XTQ. In this example, the initiator CPU enqueues a task on the target node's GPU. The target-side CPU simply waits on a shared-memory signal until the target-side GPU has completed the task.

Both CPUs begin by initializing the RDMA communication layer and NIC ❶. On the initiator side, the CPU allocates a payload and creates a command packet encapsulating the task to execute on the target ❷. In this example, the coordinated index 42 is used to associate this command with queue, kernel, and signal registrations at the target. This task is then supplied to the NIC using the *XtqPut* operation ❸. An *XtqPut* triggers the NIC to send the input data and command packet to the target.

Meanwhile, the target CPU posts the receive buffer using the RDMA communication layer ❹. Next, it initializes the local accelerator runtime and creates a kernel, completion signal, and user-mode command queue ❺. The target then registers the kernel, signal, and queue with the XTQ NIC using the *XtqRegisterKernel* and *XtqRegisterQueue* functions at index 42 ❻. These functions populate the lookup tables used by the target-side NIC. When the initiator's RDMA operation arrives at the target, the target NIC recognizes it as an XTQ-enabled RDMA and uses the Kernel Lookup Table and Queue Lookup Table to replace components of the packet and select a command queue. After the RDMA operation is complete, the NIC enqueues the packet

to the GPU, which begins execution of the kernel. Meanwhile, the target CPU waits for the operation to complete using a shared-memory signal ❼.

After initialization, an execution barrier is entered between steps ❷ and ❸ on the initiator and steps ❻ and ❼ on the target. This barrier blocks the initiator from sending a command to the target before the target's lookup tables have been populated.

In addition to illustrating the XTQ API, this example program drives home two related contributions of the XTQ framework. First, the NIC delivers the task descriptor directly to the GPU without host CPU involvement, minimizing the task launch latency for the GPU. Second, the host CPU does not have to use any cycles servicing the network request and scheduling the task on the GPU. In this simple example, the target CPU waits for the GPU to complete its operation, but in more complex workloads the CPU could be free to perform meaningful computations.

While this example may seem primitive, it is actually possible to support many features with a small amount of software support from higher-level runtime libraries. For example, the target can send the data to another node for further computation by issuing another *XtqPut* operation after the shared-memory signal resolves. Complex, cross-node dependencies and task graphs can be implemented using shared-memory signals and XTQ.

Table 4.1: XTQ simulation configuration.

| CPU and Memory Configuration | |
|---|---|
| **Type** | 4-Wide OOO, x86, 8 cores @ 3GHz |
| **I, D-Cache** | 64KB, 2-way, 2 cycles |
| **L2-Cache** | 2MB, 8-way, 8 cycles |
| **L3-Cache** | 16MB, 16-way, 20 cycles |
| **DRAM** | DDR3, 8 Channels, 800MHz |
| GPU Configuration | |
| **Type** | AMD GCN3 @1GHz |
| **CU Config** | 24 CUs with 4 SIMD-16 engines |
| **Wavefronts** | 40 Waves per SIMD (64 lanes) |
| **V-Cache** | 32kB, 16-way, 12 cycles, per CU |
| **K-Cache** | 32kB, 8-way, 12 cycles, per 4 CUs |
| **I-Cache** | 64kB, 8-way, 12 cycles, per 4 CUs |
| **L2-Cache** | 1MB, 16-way, 8 banks, 100 cycles |
| Network Configuration | |
| **Switch Latency** | 100ns |
| **Link Bandwidth** | 100Gbps |
| **Topology** | Star (single switch) |

## 4.3 Evaluation

XTQ provides new opportunities to re-evaluate design decisions in existing applications and to redesign communication in emerging applications. This section evaluates the XTQ tasking model on microbenchmarks designed to expose latencies, primitives that are intrinsic to many MPI programs, and the Microsoft Cognitive Toolkit machine learning framework.

### 4.3.1 Experimental Setup

The baseline simulation infrastructure was previously described in Section 3.1. Table 4.1 shows the specific configuration for the major components

of the infrastructure.

The simulation environment models forward-looking GPU launch latencies. Once the CP has been notified of an available command packet using its doorbell, it performs a read of the command packet and then immediately schedules the kernel when a compute unit becomes available. This limit represents the conceptual overhead of launching a kernel assuming that real-world launch overheads will trend towards this limit as tightly coupled frameworks continue to integrate GPUs more closely with CPUs. Slower launch latencies will negatively impact end-to-end speedup from XTQ since the launch latency itself will dominate total runtime. XTQ's reduction in latency would remain constant, but would contribute less to total performance.

These experiments compare three different remote tasking interfaces that will be referred to as CPU, HSA, and XTQ. These configurations are defined as follows:

- **CPU:** Remote tasking is accomplished by using two-sided send/receive pairs through user-space RDMA. These results use the application thread for message progress and active message execution unless otherwise indicated. The CPU baseline is representative of modern CPU-only active messaging schemes such as those found in the GASNet [17] runtime. The CPU configuration is included to separate the baseline benefits of GPU acceleration from those of XTQ, and also to identify problem sizes that are too small for GPU acceleration.

- **HSA:** HSA also uses two-sided send/receive pairs over RDMA, but launches the active message on the GPU after the CPU thread has received the data. The CPU communicates to the GPU through user-mode command queues. The HSA configuration represents user-space tasking on a tightly-coupled architecture without XTQ.

- **XTQ:** XTQ uses one-sided *XtqPut*s through user-space RDMA to remotely enqueue active messages on the target's GPU. The NIC places AQL packets directly into the GPU's command queue for execution.

### 4.3.2 Latency Analysis

To precisely quantify the benefits of XTQ in the model, a microbenchmark very similar to the API code example presented in Figure 4.5 is analyzed. In this benchmark, an initiator node sends a variable sized payload to the target, which simply performs a memcpy operation as the active message. This example seeks to illustrate the overheads involved and is not so much interested in the computational aspect of the active message at the target.

Figure 4.6 shows a time breakdown of CPU, XTQ, and HSA remote task spawn for a small 64B payload and a 4KB payload. As is expected, a CPU active message is still significantly faster than either GPU implementation, since it incurs no launch overheads. The small 64B payload over XTQ takes approximately 1.35$\mu$s to complete a remote task, while the same payload over HSA takes approximately 1.7$\mu$s. In XTQ, the data transfer phases from initiator to target take slightly longer than the HSA transfers. XTQ's usage

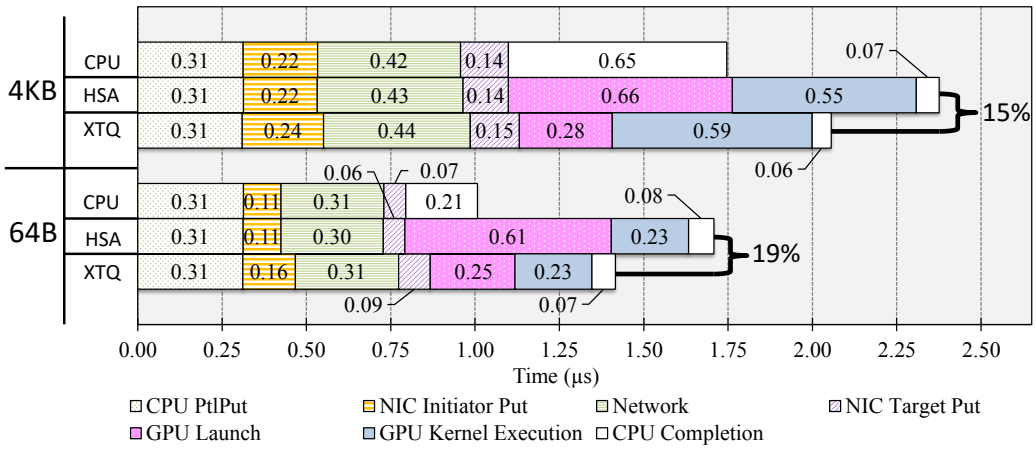



Figure 4.6: Time breakdown of remote GPU kernel launch.

of 64B HSA-like packet format slightly increases the payload size over an optimized Portals 4 implementation. However, the performance penalty incurred from transferring the command structure is dwarfed by the benefits in task launch latency. XTQ saves approximately 350ns over HSA during the task launch phase, since the NIC can directly schedule work on the GPU's command queues, while HSA requires the CPU to process an RDMA event, create the GPU task descriptor, and place it in the GPU's command queue. 4kB payloads exhibit similar savings during task enqueue, although the speedup is proportionally less due to the increase in data transfer and kernel execution time.

### 4.3.3 MPI Integration

The Message Passing Interface (MPI) [68] is the *de facto* communication library for distributed-memory HPC applications. Several MPI functions,

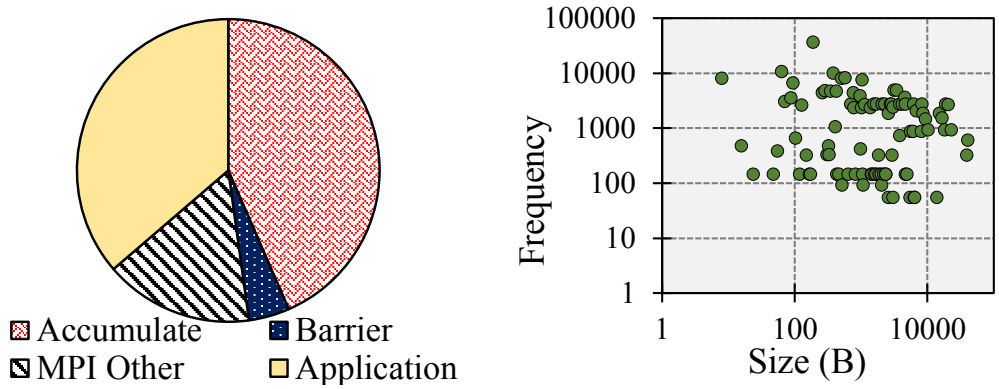
such as one-sided Accumulate operations and Reduce/Allreduce collectives, have a strong computational component that can be parallelized. Using XTQ to offload this computation to the GPU can lead to improved performance and reduced CPU overheads. By freeing the CPU to do independent work, XTQ enables non-blocking variants to achieve substantial computation overlap between the primary application thread and the accelerator.

One-sided and collective frameworks of the Open MPI [31] implementation of the MPI-3.0 specification are extended to incorporate XTQ-based Accumulates and Reductions. This implementation enables library users to reap the benefits of XTQ acceleration without altering any code.

#### 4.3.3.1 One-Sided Accumulates

The baseline CPU Accumulate implementation for Portals 4 in Open-MPI is not a true one-sided communication model: it is layered over two-sided, send-receive calls. The HSA-based implementation is similar, except for the fact that the MPI library at the target enqueues the operation on a local GPU instead of performing it directly.

XTQ-based Accumulates, however, are completely one-sided, relieving the target-side MPI process from receiving data and executing the operation. On receipt of data from the initiator, the target-side NIC enqueues the appropriate command directly on the GPU's command queue. When execution completes, the GPU updates the target-side Accumulate buffer with the result and an internal progress thread sends an acknowledgement to the initiator.




(a) Runtime distribution.

(b) Accumulate data size distribution.

Figure 4.7: NWChem *tce_ozone* Accumulate statistics.

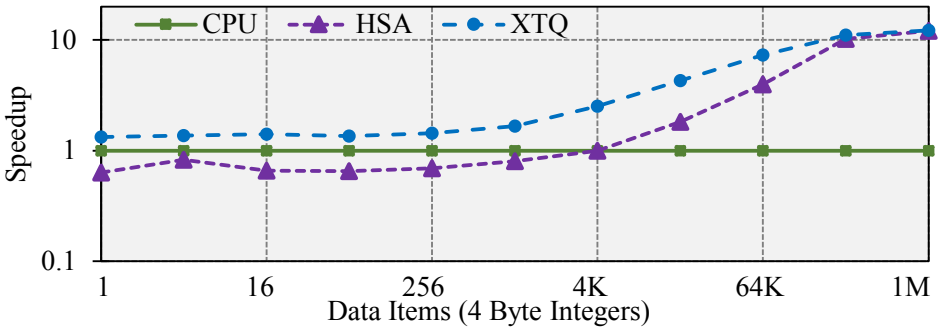
Accumulate operations are used extensively in the NWChem [99] computational chemistry package through the ARMCI/MPI3 library [70]. As an example, the *tce_ozone* workload from the NWChem regression tests spends over 58% of its time in the MPI library on a 4 node cluster, measured on real server-class hardware. Figure 4.7a shows the percentage of time spent performing various MPI functions on a single rank. The chart shows that the 43% of its total execution time performing Accumulate-related operations. Figure 4.7b shows a histogram of the payload size and number of Accumulates that occur during *tce_ozone*. The reader will observe a large concentration of small-to-medium sized Accumulates, which are ideal for XTQ lightweight messaging acceleration. While an actual simulation of NWChem on the model described in Section 3.1 was not possible due to the size of the workload, the profiling data points towards the fact that NWChem will realize measurable benefits from XTQ-based GPU acceleration.

#### 4.3.3.2 Reduce and Allreduce

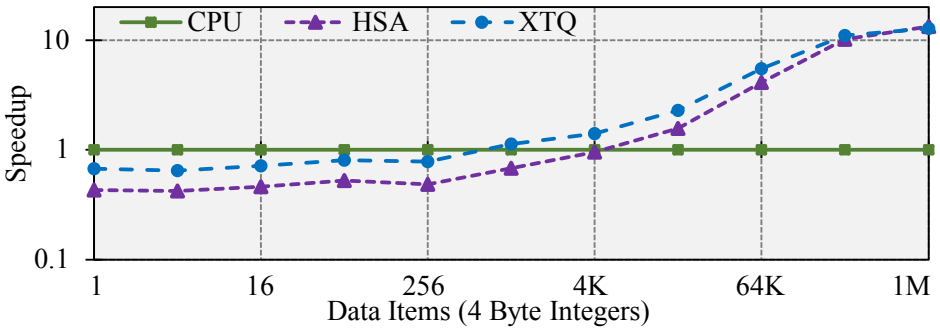
The implementation of both reductions using XTQ is built on the LibNBC library [41]. LibNBC was designed to support non-blocking collectives on generic architectures. In doing so, it creates a schedule: a directive to execute a set of operations. The schedule is modified to layer reductions on $XtqPut$s instead of two-sided, send-receive operations. An inherent problem with implementing reductions with active messages is that the target-side resources must be allocated before an active message can be received at the target. To address this problem, XTQ encodes an explicit synchronization step into the schedule by issuing zero-length send/receive pairs between initiators and targets. This step makes sure that target resources are available before any $XtqPut$s are issued.

#### 4.3.3.3 MPI Benchmarks

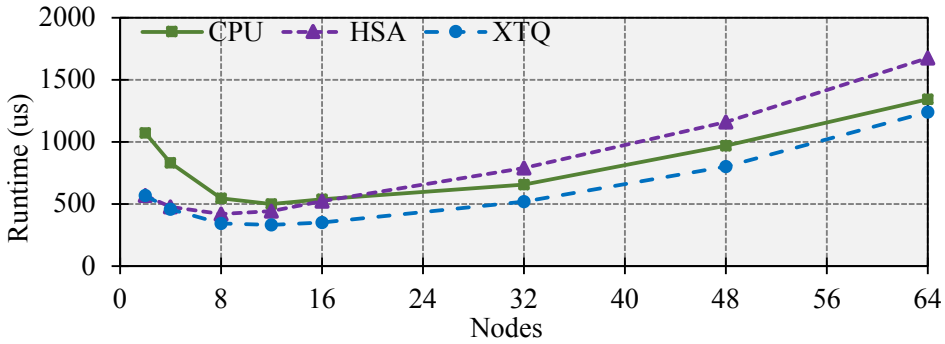
Figure 4.8 shows performance results for Accumulate, Reduce, and Allreduce implemented on CPU, HSA, and XTQ as defined in Section 4.3.1. For sufficiently large benchmarks, GPUs perform much better than CPUs for these data-parallel operations. For a two-node Accumulate operation (Figure 4.8a), XTQ performs significantly better than HSA. Interestingly, XTQ also performs better than CPU even for very small Accumulate sizes. This result occurs because CPU implements one-sided Accumulates using two-sided send/recvs, while XTQ leverages one-sided puts directly through Portals 4. XTQ offers a reasonable performance improvement of around 10-15% for a




(a) Accumulate acceleration on 2 nodes.




(b) Reduce acceleration on 2 nodes.




(c) 4MB Allreduce acceleration.

Figure 4.8: Acceleration of MPI Accumulate, Reduce, and Allreduce operations over XTQ.

two-node reduction (Figure 4.8b) over a standard HSA-enabled GPU. For both operations, the benefits of XTQ over HSA decrease as the payload increases over approximately 64KB. All of these algorithms, however, are amenable to software pipelining, which will push the payload size back into a range where XTQ shows significant benefits, even for very large transfers.

Figure 4.8c illustrates how XTQ performs on Allreduce when strong scaling up to 64 nodes for a fixed-size global data set of 4MB. Unlike Accumulate and Reduce, Allreduce requires the result to be transmitted to all nodes participating in the collective operation. Allreduce is implemented as a Reduce-Scatter followed by an Allgather, which is an efficient implementation for vector Allreduce operations [96]. With a fixed-size data set, increasing the number of nodes decreases the computation per node while increasing the total number of messages required to complete the reduction. The inflection point at approximately 12 nodes indicates the point where the overhead of sending more messages outweighs the benefits of less computation.

The figure illustrates that for small node counts, the size of each round's messages are large enough to benefit from GPU acceleration with or without XTQ. However, for larger node counts, non-XTQ-enabled GPUs are unable to amortize the high launch latency over the execution of smaller messages. Only XTQ-enabled GPUs are able to maintain performance improvements over a CPU reduction up to 64 nodes.

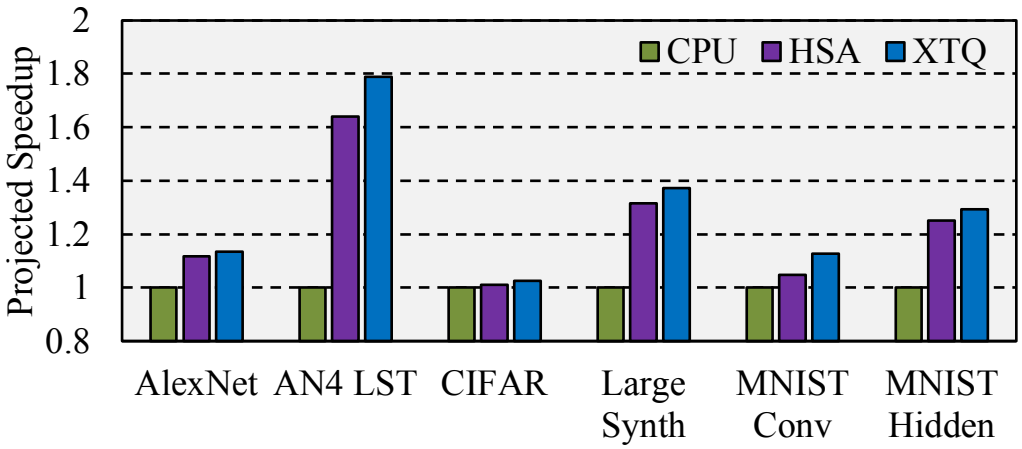



Figure 4.9: XTQ performance on Microsoft Cognitive Toolkit workloads across 8 GPU-enabled nodes.

### 4.3.4 Machine Learning

This section evaluates XTQ on a distributed deep-learning framework. The vehicle of exploration for this case study is Microsoft's Cognitive Toolkit [2]. Figure 4.9 shows the results for machine learning workloads across 8 high performance compute nodes. Utilizing GPUs for Allreduce compute provides a 23% average improvement in runtime over a baseline CPU version. XTQ-enabled acceleration gives on average an additional 8% improvement over the HSA baseline, with AN4 LST improving by 15%. CIFAR does not significantly benefit from either form of GPU Allreduce acceleration, since it is more bound by the local SGD compute phase than the gradient reduction.

## 4.4 Conclusion

Emerging node-level architectures tightly couple accelerators into host platforms. These frameworks significantly reduce task launch latency via user-level task queues and eliminate data copy overhead via shared virtual memory, paving the path for fine-grained, heterogeneous tasking models in shared-memory environments. Concurrently, RDMA enables highly efficient user-level network data transfers. This chapter proposes Extended Task Queuing (XTQ), a mechanism that combines tightly coupled, user-level task queuing with RDMA to provide heterogeneous, lightweight tasking across distributed-memory systems. XTQ is implemented as an extension to a tightly integrated, RDMA-capable NIC and enables applications to schedule tasks on accelerators and CPUs across nodes. Using XTQ, applications can send messages to remote accelerators, bypassing the operating system on both nodes and not involving the target CPU. Bypassing the target-side CPU reduces task launch latency by 10-15% for small-to-medium sized messages and frees a CPU thread from message processing to perform more useful computation.

Because XTQ lies at the intersection of several emerging paradigms, such as accelerator-based programming and lightweight distributed tasking, few existing applications directly leverage its benefits. XTQ opens up new possibilities for applications to leverage accelerators for tightly coupled communication and computation in distributed systems. Towards this end, XTQ enables the use of GPUs to accelerate Reduce, Allreduce, and Accumulate operations across a variety of payload sizes and on clusters up to 64 nodes.

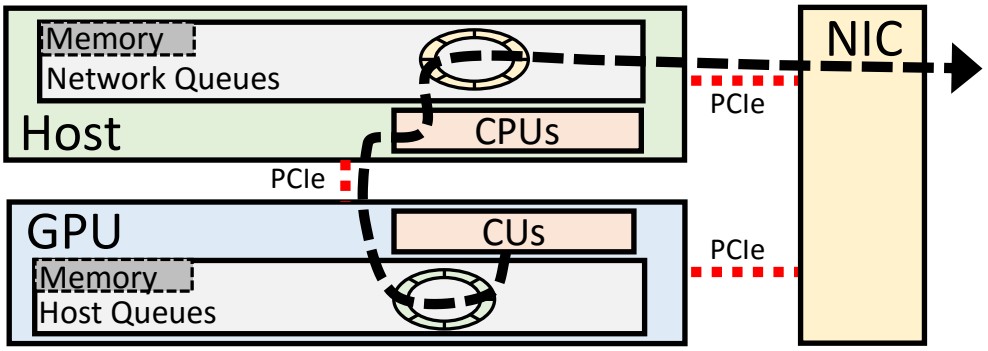
Finally, XTQ can provide up to 15% performance improvement for emerging deep learning workloads in the Microsoft's Cognitive Toolkit.

# Chapter 5

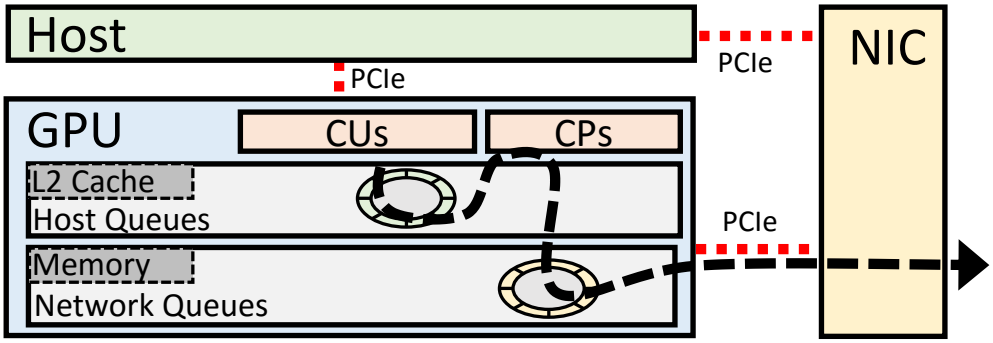
# ComP-Net: Command Processor Networking for Efficient Intra-kernel Communications on GPUs

XTQ offers a highly optimized remote kernel invocation scheme that does not involve CPU intervention to launch remote kernels at the target. While this is a useful optimization for traditional kernel-based GPU applications, it is often more natural to embed network operations within a kernel using a GPU-side runtime library, similar to how CPU threads can send remote messages by calling into MPI. Indeed, there already exists some research in this area, as was discussed in Section 2.3.2. The next two chapters in this dissertation will focus on optimizing intra-kernel networking.

This chapter improves the performance and energy consumption of intra-kernel networking using a little-known feature of modern GPUs: embedded, programmable microprocessors that are typically referred to as *Command Processors (CPs).* These processors exist on the GPU device itself and are utilized to perform the serial tasks involved with launching and tearing down a GPU kernel [81, 7]. However, in the presence of intra-kernel networking, programmers are encouraged to use larger (fewer) kernels, as they no longer need to break down kernels across network communication points.




(a) Inter-kernel networking through host threads.




(b) Intra-kernel networking through ComP-Net.

Figure 5.1: Comparison of ComP-Net to traditional intra-kernel networking schemes.

This leaves the Command Processors otherwise idle and available to assist with GPU networking.

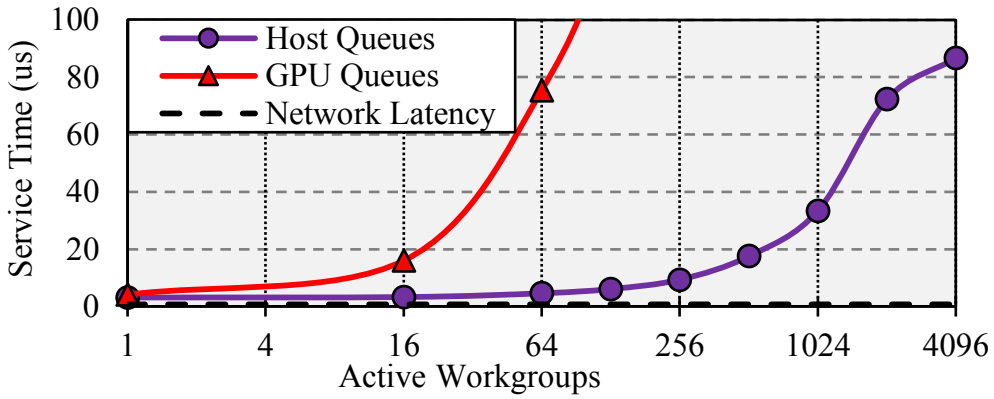
The proposed solution, which is called **Com**mand **P**rocessor **Net**working (ComP-Net) [57], moves the network service thread from the host CPU over to the GPU-resident CP. Figure 5.1 compares ComP-Net to traditional intra-kernel networking schemes where the network service threads reside on the host. In ComP-Net, GPU work-groups submit networking operations to the

CP through the GPU's shared cache hierarchy on per-work-group command queues. By hosting the networking runtime on the CP versus the host CPU, ComP-Net achieve a large reduction in latency for network operations, an increase in scalability in multi-GPU systems, and a significant decrease in energy consumption associated with the network service thread.

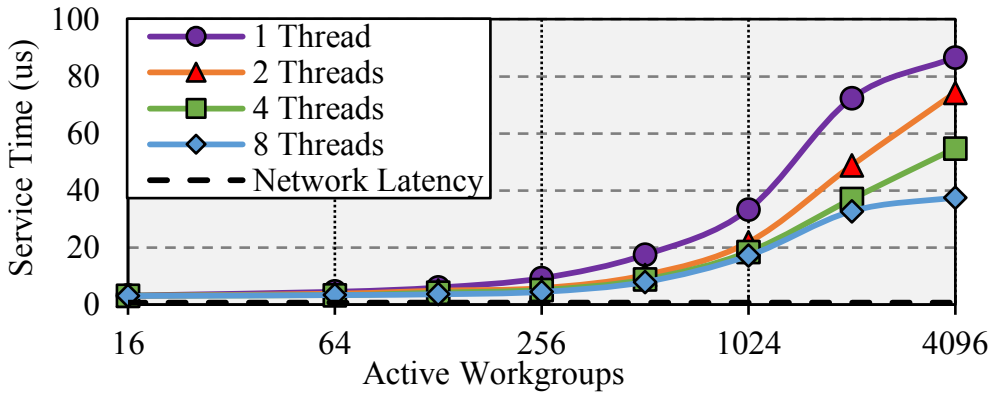
This chapter describes the ComP-Net runtime and programming interface, which was designed to take advantage of the unique capabilities of a networking CP. It discusses practical challenges related to the relaxed memory consistency model on the CP and the GPU. It also discusses ways to mitigate performance problems when sharing data between these two devices by applying some simple architectural enhancements to the GPU's L2 cache. Finally, it performs a detailed evaluation of ComP-Net versus traditional kernel boundary communication and with other intra-kernel networking designs. It is shown that ComP-Net can improve application performance by 20% and reduce energy consumption of the network service thread by up to 50% versus other intra-kernel networking designs on a 2D Jacobi stencil, Allreduce collectives, and machine learning workloads.

## 5.1 Motivating ComP-Net

This section dives into the limitations of existing intra-kernel networking schemes that use threads on the host CPU and describe how ComP-Net responds to these limitations.




(a) Average service time of CPU/GPU queues with the queue placed on the GPU and in host memory.




(b) Average service time of CPU/GPU queues with varying number of CPU helper threads.

Figure 5.2: Latency and scalability issues with intra-kernel networking via host forwarding

### 5.1.1 High Latencies

As previously mentioned, most currently existing intra-kernel networking schemes require communication between CPU network service threads and a GPU's work-groups. Unfortunately, in discrete GPU form factors, these two devices are separated by a high latency I/O interconnect.

Figure 5.2a shows the best-case latencies of intra-kernel communication between a host CPU and GPU, as observed by work-groups on the GPU. This experiment uses a simple producer/consumer queue for communication between the CPU and the GPU. There are two locations in which the queue can be placed. In the first design, the command queue is placed in the GPU device memory. The queue is either mapped to the CPU's address space and accessed with loads and stores, or it is accessed by a runtime call. Either way, the performance is quite poor, especially when the CPU is required to monitor multiple queues at once, since the long latency reads block the CPU from making forward progress.

In the second design, the command queue is placed in host memory. The GPU maps the host memory to its address space and does PCIe stores and atomics to synchronize. While this approach does perform better than the previous design, the access latency is still incredibly high, on the order of 5-80$\mu$s, which is 1 or 2 orders of magnitude higher than network latencies of 0.6$\mu$s [66]. No matter where the queue is placed, latencies are large.

This dissertation is not the first to note this restriction. Previous

works have illustrated considerable latencies that far surpasses the latency of a network interface. For example, DCGN [94] quotes latencies of 330$\mu$s and Gravel [82] uses a 125$\mu$s timeout to flush pending messages. Even recent works on powerful modern hardware, such as dCUDA [36], only achieve latencies of approximately 20$\mu$s in the best case.

While high latencies may not matter much when performing bulk synchronous transfers of large data, many network applications, even on a GPU, require more than just support for streaming transfers. Even applications that are mostly parallel still have frequent periods of serial behavior that cannot be easily overlapped. Consider the popular stencil pattern of computation, which is frequently accelerated on GPUs. In these applications, a reduction is typically performed after each relaxation phase to determine whether a convergence criterion has been met. After a local reduction across the GPU, each device contributes a small amount of data, such as the calculation of residuals or synchronization between time-steps of an iterative calculation. This step of the algorithm resides directly on the critical path where there is not enough parallelism for latency hiding to apply. It is precisely these use cases that are targeted with ComP-Net.

### 5.1.2 Poor Scalability

On GPUs, it is very likely that there will be many work-groups which need to access the network simultaneously. Figure 5.2b shows an implementation of intra-kernel networking on the host that sweeps both the number

of work-groups participating in a network operation and the number of host threads allocated to service these requests. The graph shows that a large number of host threads are required to maintain reasonable quality of service for network operations on a *single* 64 CU GPU. Requiring a large number of threads to service the GPU limits the scalability of the design. The number of required threads will only become more of an issue as the number of threads on a GPU, and the number of GPUs attached to each CPU socket, continues to increase. Trends in hot application domains such as machine learning indicate that many GPUs per node seems to be the prevailing trend in the highest performing systems [75].

There are also second-order effects associated with consuming many cores on the host. For workloads that could benefit from simultaneous CPU compute, these helper threads draw from resources that are available to the application. For workloads that only use the GPU, host threads burn unnecessary power and prevent the host CPU from entering a deeper sleep state.

### 5.1.3 The Case for ComP-Net

ComP-Net provides all the benefits of intra-kernel networking while simultaneously addressing all the above concerns. While using a networking CP, latency is drastically improved. The CP/GPU command queue is placed directly in GPU memory, which both the CP and GPU can access without traversing an I/O bus. Additionally, the CP is located behind the GPU's L2 cache. This means that the CP and the GPU can communicate with a latency

of approximately one hundred GPU cycles if the data is resident in the L2 cache. While this is still quite a bit higher than cache-to-cache communication between threads on a CPU, it is far less expensive than synchronizing over PCIe.

Scalability is also elegantly addressed by ComP-Net. As opposed to a regular CPU, CPs scale naturally with additional GPUs in a system. Since CPs are *a part of* the GPU, adding additional GPUs in a system allows you to gain more CPs for network processing. Additionally, a CP is much smaller than a core on the host, which results in significant savings in power and energy. All these effects are shown across several workloads in Section 5.4.

## 5.2 Programming Model

ComP-Net implements an OpenSHMEM-based API [12] that is exposed to the GPU programmer through a device side library. The semantics of OpenSHMEM are very close to the features offered natively by many NIC hardware vendors. This reduces the required software complexity, which makes it a natural choice for running on an embedded CP. Each ComP-Net operation is implemented as a work-group collective; the runtime executes a work-group barrier after each API call.

Work-groups are a natural granularity to perform networking on a GPU. Any larger, and ComP-Net would need to synchronize across work-groups, which is expensive and limits the ability for work-groups to overlap. Any smaller, and the message size would be too small to saturate the network

link [63]. Modern NICs require each message to be ¿= 2KB in order to properly saturate the link. The common case in GPU programming is that each work-item in a work-group is responsible for a small 4 or 8 byte element of a larger array. Assuming a work-group is on average 1K work-items in size, this means that messages would be on average 4KB-8KB per work-group.

Each ComP-Net API call (put/get/collective/etc.) takes the same arguments as a standard OpenSHMEM implementation (source/destination/length-/etc.) with the addition of a GPU-only context that provides the information needed to communicate with the CP. These arguments are placed in a producer/consumer queue and forwarded to the CP, the details of which are described in Section 5.3.

The practical details of ComP-Net communication are best described through a small example. Figure 5.3 illustrates a simple ping-pong benchmark between two GPUs. The example is written using AMD's Heterogeneous-compute Interface for Portability (HIP) [8], which has a very similar syntax to Nvidia's CUDA [73]. The pong step is omitted since it is similar to ping and offers no additional information regarding ComP-Net's API. Figure 5.3a shows the host-facing API for ComP-Net. First, the host initializes the ComP-Net runtime and creates a handle for the GPU ❶. This initialization step allocates a number of service threads on the GPU's CPs to handle messages and brings up a standard OpenSHMEM runtime under the hood. In the proposed design, Sandia OpenSHMEM (SOS) [90] is used due to its support for contexts and direct implementation on top of Portals 4, which is the API for the simulation

```
__host__ void
hostInit()
{
    // ❶ Initialize ComP-Net
    cpnet_handle_t* cpnet_handle;
    cpnet_init(&cpnet_handle, GRID_SZ / WG_SZ);
    // ❷ Allocate symmetric heap memory
    char* buf = cpnet_shmalloc(sizeof(char) *
                               GRID_SZ / WG_SZ);
    // ❸ Initiator/target launches kernel
    if (cpnet_handle->pe == INITIATOR) {
        hipLaunchKernel(Ping, GRID_SZ,
                        GRID_SZ / WG_SZ, 0, 0,
                        cpnet_handle, buf);
    } else { /* Launch target kernel. */ }
}
```

(a) Initialization and host code.

```
__device__ void
Ping(cpnet_handle_t *cpnet_handle
     char* wg_buffer)
{
    // ❹ Extract context from global handle
    __shared__ cpnet_ctx_t cpnet_ctx;
    cpnet_ctx_create(cpnet_handle, cpnet_ctx);
    // ❺ Each WG pings target
    cpnet_shmem_char_p(cpnet_ctx,
        wg_buffer[hipBlockIdx_x],
        1, TARGET);
    // ❻ Each WG waits for pong target
    cpnet_shmem_char_wait_until(
        wg_buffer[hipBlockIdx_x, 1);
    cpnet_ctx_destroy(cpnet_ctx);
}
```

(b) Device ping to remote GPU using ComP-Net.

Figure 5.3: ComP-Net ping/pong example on host and device.

environment.

Next, the host allocates a network accessible buffer on a symmetric heap allocated from GPU memory ❷. A symmetric heap is a buffer that resides on each participating process. Data allocations are a collective operation where a variable is allocated locally on each symmetric heap at the same location. Processes can reference memory on another process by simply using the pointer to the local value along with the PE of the target. In Comp-Net, the symmetric heap code in SOS is modified to allow allocation of memory on dGPU devices. The details of this are beyond the scope of this work, but GPU-side symmetric heap allocators have been explored in the prior art [37]. Finally, a GPU kernel is launched with the ComP-Net handle and the allocated buffer ❸.

Figure 5.3b illustrates the device side API from ComP-Net. The GPU first calls an initialization function with the host-provided ComP-Net handle ❹. This API creates a private communication context for each work-group. This context is allocated in scratch-pad memory and initializes its data from the global ComP-Net handle. The next two steps perform standard one-sided network calls to perform a remote put on the target ❺ and wait for the corresponding ping ❻. Each work-group performs a separate ping operation on an independent buffer entry. The details of what happens internally in ComP-Net are described in Section 5.3.

One important detail of ComP-Net is the use of OpenSHMEM contexts. Contexts were recently added to the specification as a way to wait on (i.e., quiet) a subset of the outstanding network operations [12]. While useful for

CPUs, this becomes a critical requirement on GPUs. Work-groups should not be stalled waiting for unrelated messages, as this significantly reduces the amount of available communication and computation overlap.

The prototype ComP-Net implementation does contain two programming model limitations. First, the initial implementation only allows for a single symmetric heap to be bound to a single PE. Therefore, all allocated ComP-Net symmetric heap memory is always placed on GPU memory. Finally, ComP-Net does not check for and short-circuit intra-process communication or inter-process communication where both processes reside on the same node.

## 5.3 Architecture

This section discusses the runtime architecture for GPU/CP communication and synchronization, as well as GPU L2 cache enhancements to reduce cache thrashing for ComP-Net.

### 5.3.1 GPU/CP Communication

The CP and the GPU communicate through per-work-group producer/consumer queues. However, building producer/consumer queues between the GPU and the CP on top of a weakly coherent cache hierarchy is different than on a fully coherent CPU cache hierarchy. The memory consistency model of the GPU is described in more detail in Section 2.2.3.

Figure 5.4 describes the details of both the producer and consumer side

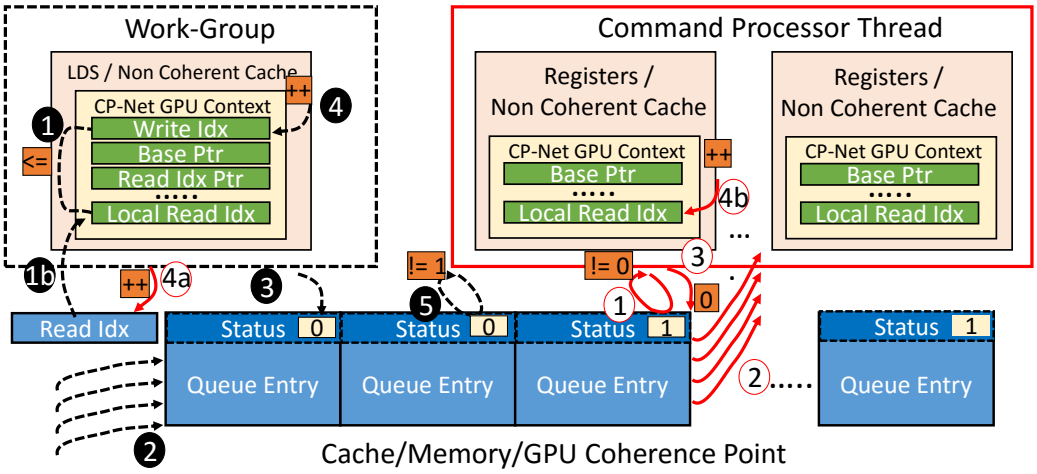



Figure 5.4: Illustration of work-groups and CP network service threads communicating using ComP-Net.

of a ComP-Net operation. The producer (a GPU work-group) illustrates steps with black circles and dotted lines, while the consumer (a CP network service thread) illustrates steps with white circles and solid lines. In ComP-Net, each work-group maintains a context that contains the write index, base pointer of the queue, a pointer to the read index that is shared between the CP and work-group, and a local copy of the read index. When a GPU work-group wishes to communicate a network packet with the CP, it first needs to reserve space in the producer-consumer queue. A single GPU work-item in a work-group is selected to perform all the serial operations. This does cause some amount of control flow divergence on the GPU, but it is still significantly faster than ending the kernel. First, the work-group checks if the queue is full by comparing the local read index to its private write index 1a. If the work-group thinks that the queue is full using its local read index, it goes ahead

and refreshes its local copy with the version in shared memory and repeats step ❶b. This reduces accesses to global GPU memory in the common case. Once there is space in the queue, the work-group then fills the slot with all the information necessary to perform a network operation (operation type, destination, length, etc.) and enqueues a release marker so that the data is visible to the CP ❷. If the data to be sent is less than 8 bytes, it is copied directly into the queue entry to enable the CP to inline the data. Otherwise, a pointer to the data buffer in GPU memory is copied. Next, the work-group sets a status bit in the queue entry to inform the CP that the data is ready for consumption with another device scope write with release marker ❸, and increments its local write index to complete the operation ❹. On a blocking call, or a quiet operation for in-flight non-blocking calls, the work-group needs to check on completion for any outstanding requests. This is done by polling on the status bits on all requests between the read and the write index ❺. An acquire marker needs to be inserted after every iteration of the loop to invalidate the non-coherent L1 cache for the work-group.

On the consumer side, the command processor also keeps a context for each work-group it is responsible for. The command processor adds its local read index to the base pointer of the queue and poll on the status bit of the next queue entry ①. Similarly to the work-group side, the CP needs to enqueue an acquire marker to invalidate its L1 cache. After the CP detects that a network packet is available, it reads out the appropriate data and calls into a standard OpenSHMEM implementation to perform the operation ②. If

the operation is non-blocking, the CP immediately marks it as complete by setting the status bit followed by a release operation ③, and increments the read pointer in both its local memory 4b and on the host 4a. If the operation is blocking, the CP translates the operation into a nonblocking OpenSHMEM call, but does not mark the queue entry as complete. This translation is to prevent the CP network service thread from blocking, which would leave it unable to service other requests from other work-groups. After performing a predefined number of polling rounds through all queues assigned to it, the CP will quiet the network and mark all blocking queue entries as complete.

### 5.3.2 CP Atomic Operations

Once the CP networking thread(s) has received a command from the host, it forwards it to an OpenSHMEM library. Largely, the OpenSHMEM implementation is unmodified except for the addition of acquire/release markers to communicate between the CP and the NIC. The procedure is similar to GPU/CP synchronization, except the operations are performed at system scope instead of device scope.

However, while operating in a multi-threaded environment with multiple CPs, the OpenSHMEM library makes heavy use of mutexes to protect shared network data structures across threads. Typically, GPUs resolve device scope atomics at the point of device-level coherence, which, in the case for AMD GPUs, is the L2 cache. CPUs work in an entirely different manner. For the prototype implementation, it is assumed that the CP is running an x86 in-

struction set, which uses Read-Modify-Write (RMW) prefixes on instructions to take ownership of critical sections. On a standard CPU, this is implemented by two completely different requests. A CPU maintains atomicity by locking either the cache line or the entire response path of the coherent L1 cache. On a non-coherent GPU, this implementation will not provide atomicity. Therefore, a locking cache state is introduced to the GPU's L2 cache to support RMW instructions. All RMW instructions issued from the CP automatically bypass the L1 cache. The read cycle in the RMW locks the cache line, and the write instruction unlocks it. This also has the side effect that RMWs are not only atomic with respect to other CP threads, but also to the GPU itself.

### 5.3.3 Controlling Cache Thrashing

To reduce latency between the CP and the GPU, ComP-Net should leverage the shared Last-Level Cache (LLC) between the two components. Unfortunately, the preliminary design exploration of ComP-Net revealed a major limitation when hooking a networking CP up to the GPU's LLC. In most applications that fully utilize the GPU, the time data is resident in the LLC is rather low. This is mainly due to the fact that the GPU performs a significant amount of streaming accesses coupled with the relatively small ratio of cache space to the number of GPU compute threads.

This fact has major performance implications on ComP-Net. If enough wavefronts were engaging in streaming operations, the data shared between the CP and the GPU through the LLC would be evicted. This forces the

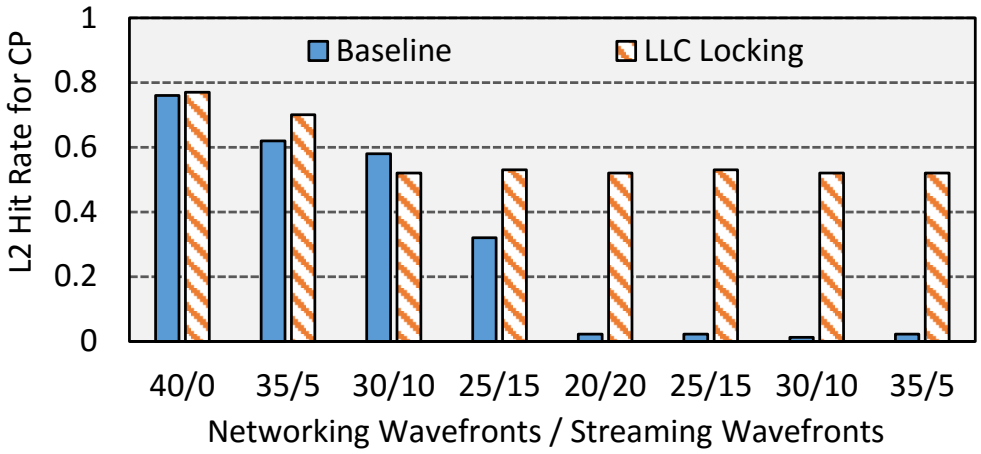



Figure 5.5: L2 hit rate for CP-generated accesses under different GPU load conditions.

CP and the GPU to communicate through relatively slower GPU memory. To illustrate this point, Figure 5.5 shows an experiment where the ratio of networking work-groups to streaming work-groups are sweeped. For the purposes of this experiment, we only use a single CU on the GPU and reduce the size of the L2 cache accordingly. L2 hit rates are reported only from accesses that are generated by the CP. The experiment shows that, in the absence of streaming wavefronts, the L2 hit rate for the CP is relatively high, indicating that the CP and the GPU are sharing data through the L2 cache successfully. However, as successively more streaming wavefronts are added, the CP L2 hit rate plummets to almost zero.

Fortunately, the same technique previously used for implementing CPU mutexes can be extended to prevent eviction of control plane data from the LLC. Towards this end, the GPU ISA is modified to allow a locked store

operation that puts a cache line in the same lock state as a CPU-side RMW operation. This data is only unlocked when it has been accessed by the CP. Since the control plane data is small, and the CP will most likely access the data quickly, the total number of cache lines and the amount of time that a cache line is locked is very small (on the order of 800ns - 1$\mu$s).

Using this modification, an additional experiment is performed and labeled 'LLC Locking' in the graph. LLC locking significantly improves the hit rate for networking work-groups that share an L2 cache with streaming work-groups. However, while the CP hit rate no longer plummets in the presence of streaming work-groups, it is still reduced by 20% in the worst case from the baseline with no streaming interference. This reduction is due to the fact that only data shared by the GPU and the CPU is locked. Data that is used solely by the CP that does not fit in the CPs relatively small L1 cache is spilled out to the L2 and affected by thrashing. This indicates that there are still performance optimizations to be gained by drawing on more sophisticated cache partitioning or locking schemes from the literature. A deeper exploration of this interaction is left as future work.

## 5.4 Evaluation

This section evaluates ComP-Net performance and energy consumption on a number of different workloads.

Table 5.1: ComP-Net simulation configuration.

| CPU and Memory Configuration | |
|---|---|
| **Type** | 8-Wide OOO, x86, 16 cores @ 4GHz |
| **I, D-Cache** | 64K, 2-way, 2 cycles |
| **L2-Cache** | 2MB, 8-way, 8 cycles |
| **L3-Cache** | 16MB, 16-way, 20 cycles |
| **DRAM** | DDR4, 8 Channels, 2400MHz |
| GPU Configuration | |
| **Type** | AMD GCN3 @1.5GHz |
| **CU Config** | 12 CUs with 4 SIMD-16 engines |
| **Wavefronts** | 40 Waves per SIMD (64 lanes) |
| **V-Cache** | 32kB, 16-way, 12 cycles, per CU |
| **K-Cache** | 32kB, 8-way, 12 cycles, per 4 CUs |
| **I-Cache** | 64kB, 8-way, 12 cycles, per 4 CUs |
| **L2-Cache** | 1MB, 8 banks, 16-way, 100 cycles |
| CP Configuration | |
| **Type** | 2-Wide OOO, x86, 2 cores @ 2GHz |
| **D-Cache** | 32kB, 8-way, 4 cycles |
| **I-Cache** | 16kB, 8-way, 4 cycles |
| **L2-Cache** | Shared with GPU |
| Network Configuration | |
| **Switch Latency** | 100ns |
| **Link Bandwidth** | 100Gbps |
| **Topology** | Star (single switch) |

### 5.4.1 Experimental Setup

ComP-Net is evaluated using the simulation infrastructure that was previously described in Section 3.1. Table 5.1 shows the specific configuration for the major components of the infrastructure. The CP itself is configured according to the specifications listed in Orr et al. [81].

The experiments compare five different implementations of GPU net-

working:

- **CPU:** Standard node with just a CPU and a NIC. OpenMP is used for thread-level parallelism, and MPI is used for multi-node communication.

- **HDN:** Host-Driven Networking represents a traditional GPU networking node that can be bought off the shelf today. Kernels are launched by the host to perform computation and all networking is routed through MPI at kernel boundaries. This is representative of industry technologies such as GPUDirect RDMA [64].

- **APU:** Intra-kernel networking by placing the network thread on the CPU on an APU. The GPU can communicate through host memory and is coherent through a directory-based protocol. This is representative of the Gravel intra-kernel networking implementation for APUs [82]. Gravel targeted highly irregular applications where each work-item potentially needs to communicate with a different node than its neighbors. Coalescing was employed across the entire GPU to generate larger messages to maximize network efficiency. The applications used in this dissertation are more structured, so coalescing is not incorporated into the APU baseline.

- **dGPU:** Intra-kernel networking by placing the network thread on the CPU of a host machine in a standard off the shelf dGPU-enabled system. In this baseline, the GPU producer/consumer queue is placed in GPU

memory. The CPU reads and writes to and from the producer/consumer queue through a PCIe bus model. This is representative of most previous works that have attempted intra-kernel networking using helper threads on the host [51, 94, 36]. The dGPU simulation also serves as the baseline for all results that report normalized energy consumption or speedups.

- **ComP-Net:** Intra-kernel networking using ComP-Net. The network thread is placed on the CP. The CP and GPU communicate through a shared L2 cache on the GPU.

The APU versus ComP-Net results are surprising enough to address up front. Although ComP-Net is more energy efficient, for most of the results, APU and ComP-Net have very similar performance. Since neither communicate over PCIe, this result implies that the gains in synchronizing through the GPU's L2 cache (in ComP-Net) are balanced out by the relative decrease in performance of a CP versus CPU for running the network stack itself.

These observations do not diminish the usefulness of ComP-Net. Although APU is included for completeness, virtually all GPU compute deployments employ discrete GPUs since APU-based designs do not offer enough compute units or memory bandwidth for real applications. With this in mind, the correct comparison point for ComP-Net is the other discrete GPU baselines (either HDN or dGPU).

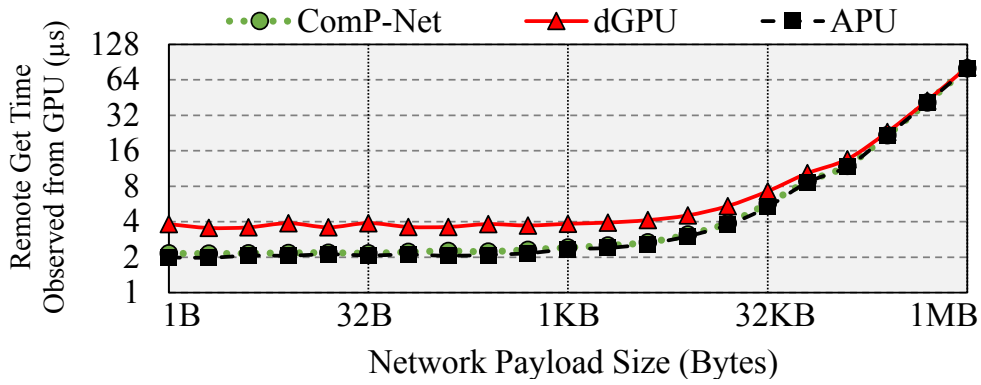
### 5.4.2 Microbenchmarks

This section describes the performance of ComP-Net and competing designs on a number of controlled microbenchmarks. This section only compares the three intra-kernel networking designs (i.e., dGPU, ComP-Net, and APU).

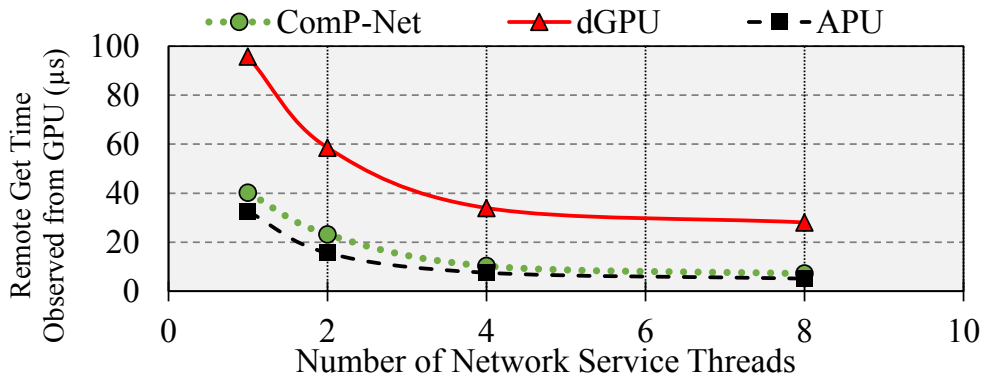
Figure 5.6a shows the latency of a single work-group performing remote Get network operations of varying payload sizes across different intra-kernel networking designs. dGPU based designs incur over 2x the latency of both ComP-Net and APU designs. As the payload size increases, network bandwidth becomes the ultimate determining factor, and the all three intra-kernel networking designs start to perform similarly.

Figure 5.6b shows the performance of the three intra-kernel networking schemes when fully loading the GPU with network requests. In this example, 480 simultaneous work-groups of 64 threads each are scheduled, which will fully saturate the 12 CU system. The experiment then sweeps the number of network service threads, equally distributing the load across all the network service threads. Both ComP-Net and APU perform better than dGPU. The dGPU design performs particularly poorly when there are multiple work-groups since the CPU service threads have to poll several queue descriptors over PCIe.

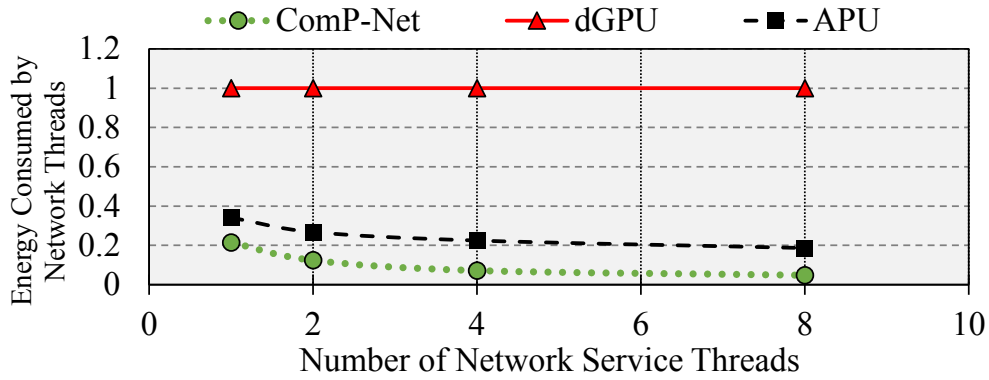
Figure 5.6c shows the energy consumption of the previous multi-threaded experiment. This study and all subsequent energy studies will focus just on the




(a) Sweep of payload size for a single work-group and network service thread.




(b) Sweep of threads for 1 Byte transfers with 480 work-groups.




(c) Sweep of CPU energy consumption for 1 Byte transfers with 480 work-groups.

Figure 5.6: Microbenchmarks of ComP-Net vs other intra-kernel networking baselines.

energy consumed by the network service thread(s). While the GPU baseline will consume slightly less energy on APU and ComP-Net due to less time spent polling on the completion of requests, the major energy reduction is assumed to come from the vastly different power profiles of a large host CPU versus a much smaller embedded CP. It is observed that ComP-Net offers significant energy savings over both APU and dGPU. ComP-Net consumes a third of the energy of dGPU, and half the energy of APU.

### 5.4.3 Jacobi 2D Stencil

This section evaluates the performance of ComP-Net over a Jacobi relaxation problem. In Jacobi, a series of operations are performed on a local data set, followed by a halo exchange of neighboring data. In the example, a two-dimensional stencil is split in one dimension over all participating nodes. For the CPU and HDN version, the algorithm follows three main phases. First, the next value of the local stencil is calculated (either on the GPU or the host). Next, the halo region is exchanged with a node's adjacent peers. Finally, a residual is reduced over the stencil to determine whether to continue the relaxation.

The intra-kernel version is implemented similarly, but with an important distinction; the host is no longer needed beyond data preparation. Since network transfers can now be performed from within a kernel, the main relaxation loop can be moved onto the GPU. Additionally, work-groups that are performing a halo exchange on the edge of the stencil can automatically

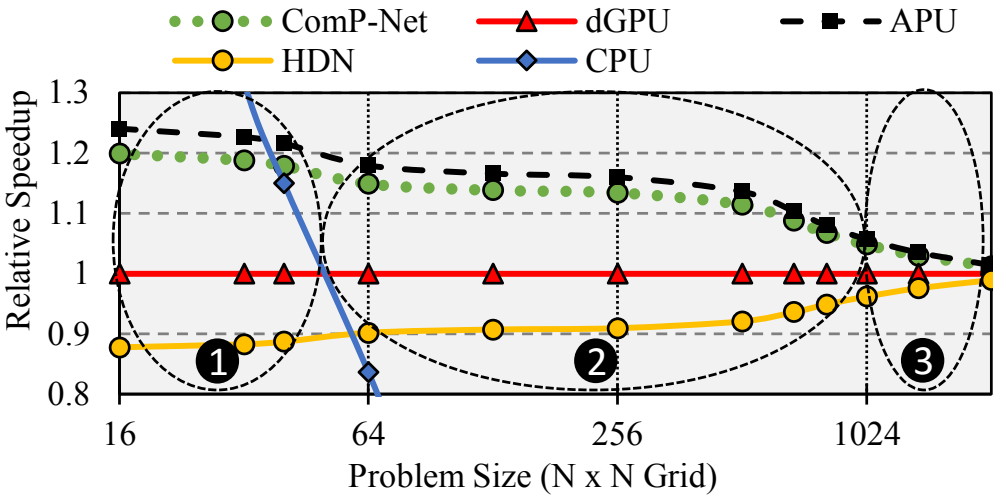



Figure 5.7: Performance of different networking techniques on various stencil sizes.

overlap with work-groups on the interior. Without intra-kernel networking, this overlap would need to performed using an exterior and interior kernel.

Figure 5.7 illustrates the results of the Jacobi relaxation on the sample systems. The results are presented as speedup to the dGPU baseline and represent a single iteration of Jacobi with varying local problem sizes. The figure shows three regions of interest. In Region ❶, the CPU performs best. This is because the problem size is much smaller than can be accelerated on the GPU. In Region ❷ GPUs start to become advantageous. In this design, ComP-Net and APUs perform better than dGPU and HDN by 10-20%. In Region ❸, all the GPU versions start performing similarly, since the problem size is large enough where intra-kernel networking latencies are no longer the bottleneck.

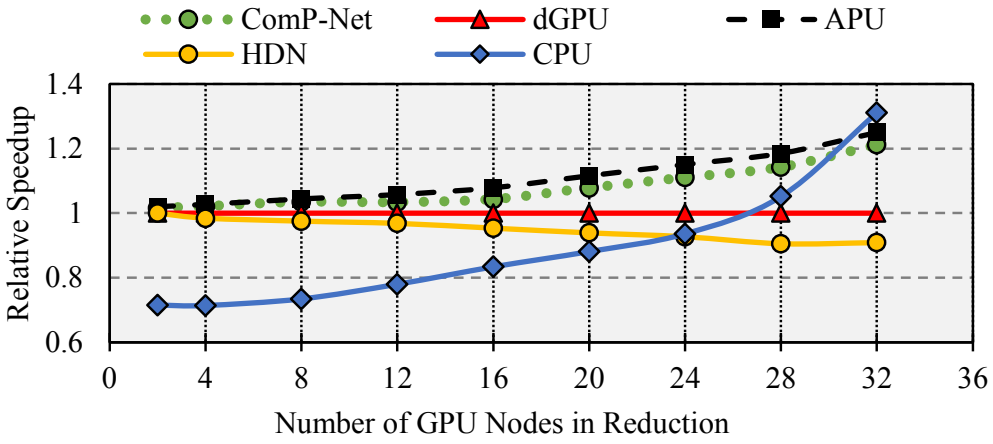
### 5.4.4 Allreduce

Collective operations are critical for a large number of distributed GPU applications [21, 72]. This section explores the performance of collective operations by measuring the performance of the Allreduce algorithm on ComP-Net.

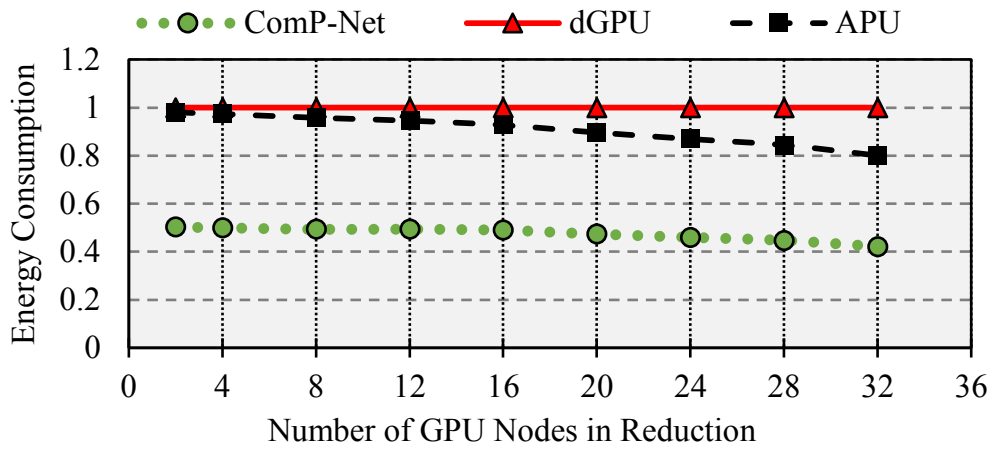
The CPU and HDN baselines use the Allreduce implementation provided by Baidu [11]. This design implements an Allreduce using nonblocking send and receive operations, with computation performed on the CPU or the GPU.

The proposed intra-kernel Allreduce design uses a multi-ring algorithm that maximizes GPU and NIC utilization with fine-grained overlap of communication and computation. The number of rings is defined by the number of work-groups participating in the Allreduce on each GPU. In $Ring_i$, $WG_i$ on Kernel/GPU of process $P$ receives data from $WG_i$ of process $P-1$ and sends data to $WG_i$ of process $P+1$. The ideal number of work-groups (rings) per process is a function of the message-size, chunk size, and bandwidth of the network.

Figure 5.8 shows a strong-scaling study of a 64MB Allreduce operation on all the evaluated configurations. Figure 5.8a, illustrates that, for a small number of nodes, the GPU results all look similar, since the average problem size per GPU is large and network latencies do not significantly impact performance. As the number of nodes increase and the amount of work per GPU decreases, the performance benefits of ComP-Net over competing approaches




(a) Performance w.r.t dGPU.




(b) Energy consumption w.r.t dGPU.

Figure 5.8: Performance and energy of different networking techniques on Allreduce of different input sizes.

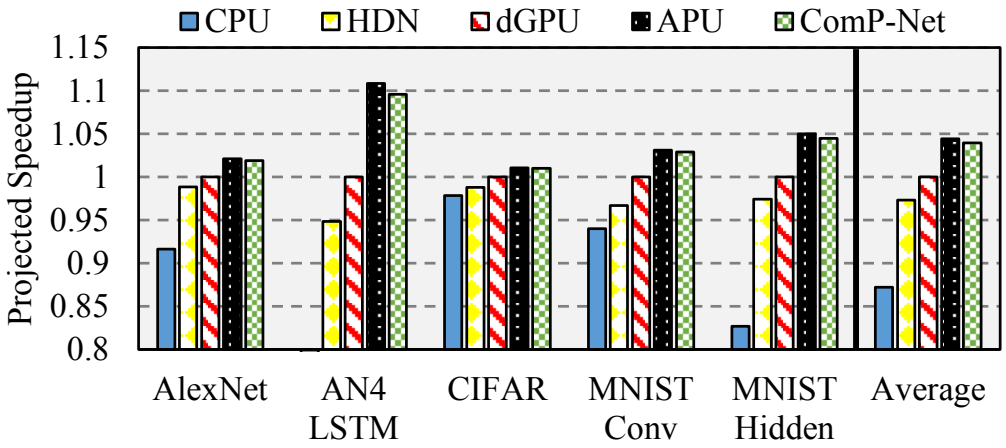



Figure 5.9: Projected speedups on Microsoft Cognitive Toolkit workloads with intra-kernel Allreduce on ComP-Net.

(except APU), becomes more pronounced. Eventually, the problem size per node becomes small enough where the reduction is optimally calculated on the CPU. Figure 5.8b shows the energy consumption of ComP-Net compared to other approaches. It is observed that ComP-Net is 50% more energy efficient than both dGPU and APU baselines.

### 5.4.5 Machine Learning

Figure 5.9 shows how ComP-Net improves performance of GPU training of neural networks on a cluster of 8 nodes with a single GPU each. The difference between the experiments is the device and networking policy (intra- versus inter-kernel) used for the reduction.

The biggest jump in performance occurs when switching from a CPU-based reduction to a GPU-based one. Further optimizing the Allreduce portion

of the training phase through ComP-Net improves total workload performance over a dGPU baseline by 5% on average. However, depending on how much the application is bound by networking, that number can vary from 11% in AN4 LSTM to 2% in CIFAR.

## 5.5 Conclusion

This work improves the performance and energy efficiency of intra-kernel communication using a lesser known feature of modern GPUs: embedded microprocessors that are typically referred to as Command Processors (CPs). The proposed design, which is called **Com**mand **P**rocessor **Net**working (ComP-Net), moves the network service thread from the host CPU over to the GPU-resident CP. This chapter described the ComP-Net programming model, discussed a detailed mechanism for GPU/CP synchronization, and implemented architectural modifications to reduce cache thrashing between the GPU and CP. Overall, ComP-Net can improve application performance up to 20% and provide up to 50% energy reduction of networking threads versus other GPU networking solutions on a Jacobi stencil, Allreduce collective, and machine learning workloads.

# Chapter 6

# GPU Triggered Networking for Intra-Kernel Communications

ComP-Net optimizes the traditional GPU Host Networking approach for intra-kernel networking by relocating the networking runtime from the host CPU to the embedded GPU CP. However, the GPU still needs to talk to the CP thread in the critical path of sending a message. This chapter introduces a new flavor of intra-kernel GPU communication called ***GPU Triggered Networking*** (GPU-TN). GPU-TN implements a NIC hardware mechanism by which the GPU can directly trigger the NIC *from within a kernel* as in GPU Native/Host Networking, while still providing high levels of performance without a critical path CP/CPU helper thread. In this approach, the host CPU is responsible for creating the network command packet on behalf of the GPU and registering it with the NIC. When the GPU is ready to send a message, it simply "triggers" the NIC using a memory-mapped store operation. A small amount of additional hardware in the NIC collects these writes from the GPU and initiates the pending network operation when a threshold condition has been met. GPU-TN provides the following advantages over the previously discussed GPU networking paradigms:

- **GPU Triggered**: Like GDS and GPU Native Networking approaches, GPU-TN utilizes the GPU to ring the doorbell of the NIC. Critical path control flow switches between the CPU and GPU are avoided by allowing the GPU to initiate network transfers by communicating directly with the network adapter.

- **Intra-Kernel Initiation**: GPU-TN allows for GPU kernel code to specify network initiation points. This programming model enables more fine-grained and frequent messaging capabilities than kernel-boundary communication. Additionally, it is much easier to overlap network operations with local computation since individual work-groups and threads can send messages independently.

- **Reduced GPU Overhead**: Since the CPU constructs the network packet and registers it with the NIC, GPU-TN avoids performance problems that have impaired some previous GPU Native Networking intra-kernel solutions. Additionally, GPU-TN eliminates the heavyweight kernel startup/teardown costs implicit to kernel boundary networking strategies. This is particularly important with strong scaling, as high kernel launch overheads will eventually dominate total execution time.

- **Reduced CPU Overhead**: GPU-TN does not require helper threads on the CPU to poll for and service GPU message requests. Removing helper threads on the CPU saves power and frees up the CPU to perform more useful work. These helper threads are common to all GPU Host

Networking programming models.

- **Relaxed Synchronization**: The GPU can initiate messages that have not yet been posted by the CPU, providing hardware-level synchronization to associate the two operations on the NIC. This allows for overlapping the network post operation from the host with the kernel launch on the GPU.

GPU-TN is inspired by triggered operations [89], which are used to optimize sequences of related networking activities in high-performance NICs and switches. It is also inspired by a CPU-side, multi-threaded message passing technique called *partitioned send* [33], which optimizes communication in systems where each thread contributes a small portion of data to the total message.

This chapter explores the design and evaluation of GPU-TN. It describes the division of responsibilities between the host CPU and the GPU, and the small amount of hardware changes that are needed to implement triggered operation semantics on the NIC. It also describes how GPU-TN's intra-kernel API offers a high degree of flexibility for the kernel programmer. Finally, it evaluates GPU-TN in the context of a simple microbenchmark, a Jacobi decomposition representative of many iterative stencils, an important MPI collective operation, and emerging machine learning applications. GPU-TN can provide up to 25% performance improvement over GDS-like approaches, and up to 35% over an optimized HDN solution across varying size

clusters up to 32 nodes.

## 6.1 Architecture

This section describes the GPU-TN model. It explores how the CPU and GPU interact during normal operation and how race conditions between the two are resolved. It also illustrates that GPU-TN can be incorporated into a high-performance NIC with little hardware complexity.

### 6.1.1 Overview

GPU-TN uses a hybrid CPU/GPU primitive to enable network communication initiated by a GPU from *within* a kernel. The main idea is to support efficient networking from the GPU by offloading the serial communications runtime and network packet creation to the CPU, while still allowing the GPU to initiate the network operation directly by performing a simple memory-mapped write operation of a tag to a particular address. Each trigger operation is completely independent and can be activated separately, which enables efficient networking from within a kernel. This avoids the high hardware scheduler cost present in kernel-boundary networking solutions and enables more fine-grained messaging capabilities.

Figure 6.1 shows the steps involved in performing a GPU-TN enhanced networking operation on the initiator. The CPU first creates the network operation, allocates memory for the message buffer, and sends the command to the NIC ❶. The CPU is responsible for creating the network operation

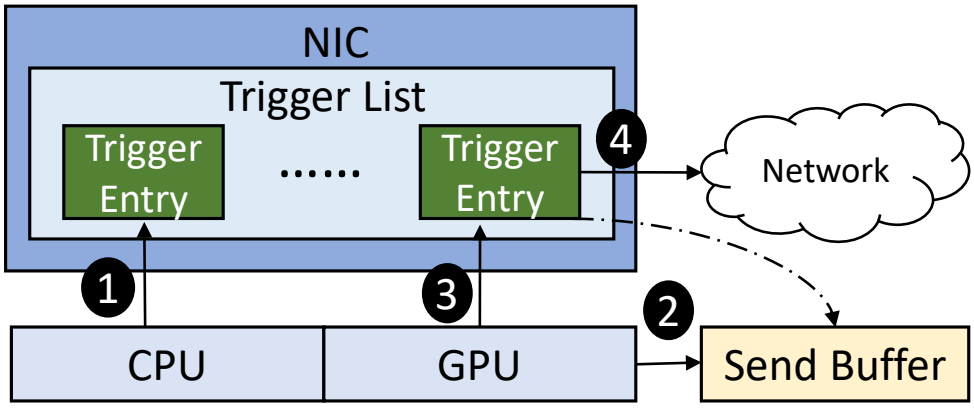



Figure 6.1: Overview of a GPU triggered operation in GPU-TN.

using the triggered operations API (see Section 6.2) and registering it with the NIC. The network runtime library allocates a *trigger entry* to represent the state of a triggered operation on the NIC and appends this entry to a list of all registered entries called the *trigger list*. A trigger entry is composed of the following fields:

- **Network Operation**: Description of the network operation and all the metadata required to execute that operation, such as a pointer to the memory resident send buffer, length, target ID, etc.
- **Tag**: Unique identifier for this trigger entry.
- **Counter**: A counter collecting the number of writes to the trigger address matching this Tag.
- **Threshold**: Constant value representing the number of writes to collect before initiating the network operation.

Once a trigger entry has been allocated and is visible to the NIC, the GPU kernel is launched and is provided one or more tags, along with a memory-mapped address with which to activate trigger operations. This address will be referred to as the *trigger address*. During kernel execution, the GPU will populate the send buffer with data to send to another node ❷. After the send buffer is populated, the GPU selects the tag corresponding to the pre-registered message that it currently wishes to send. The GPU then notifies the NIC that the triggered put operation is ready by performing a posted write of the tag to the memory-mapped trigger address ❸. This write is routed to the NIC and placed in a FIFO associated with the trigger address. The NIC pops entries from the FIFO and searches the trigger list for a tag match on a trigger entry. When a match is found, the NIC increments the counter value associated with the matching trigger entry. When the counter value becomes greater than or equal to the CPU-provided threshold, the NIC performs the associated network operation ❹. The counter is used to delay initiation of a message until multiple triggers occur for a given tag and is used to implement many different granularities of messaging which are discussed in detail in Section 6.2.

### 6.1.2 Relaxed Synchronization Model

As presented, the current design requires the CPU to first post the network operation before the GPU activates it. This dependency implies explicit software-based synchronization between the CPU and GPU, which once

again places the CPU in the critical path of GPU operation. However, a small modification of the base GPU-TN design can resolve these races by allowing the CPU and GPU to naturally synchronize with hardware support on the NIC. The GPU can safely trigger operations that have not yet been registered with the NIC. This is a useful performance optimization, as the posting of the network operation can be overlapped with the kernel execution with no synchronization between the CPU and GPU.

If the NIC receives a write to the trigger address that does not match any tags, then the NIC allocates a trigger entry for this tag without a corresponding network operation or threshold. Subsequent writes to the trigger address that match this tag will increment the counter as normal. However, the NIC will not initiate the network operations, as the CPU has not yet provided the operation or threshold.

When the host CPU registers the triggered network operation, the NIC checks to see if the tag matches any trigger entries that are already allocated in the trigger list. If so, the new triggered operation is associated with the existing counter. If the counter value is already greater than or equal to the threshold, the network operation is executed immediately. Otherwise, the threshold and network operation fields of the matching trigger entry are populated and the system works as previously described in Section 6.1.1.

Note that even though the CPU and GPU can register/trigger network commands to the NIC in any order, they must still agree on the value of the tags. Otherwise, it is possible for the GPU to trigger the wrong network

operation. This fact can have ramifications when attempting to reuse tag values. Therefore, explicit synchronization between the CPU and the GPU should always be used when reusing tags.

### 6.1.3 NIC Hardware Extensions

GPU-TN requires minimal modifications to a standard RDMA network interface to support these new semantics. The most straightforward implementation would implement the trigger entries on the NIC itself using custom hardware. This is the method evaluated in the prototype implementation of GPU-TN discussed in Section 6.3. If the trigger list is particularly large, a design that stores the trigger list in main memory could be preferred. A cache on the NIC could then be used to save frequently accessed structures. Each trigger entry would be relatively small (on the order of one or two 64-byte cache lines) depending on the size of the associated network operation. The details of such a cache design are largely dependent on the requirements of the application and are left to future work.

If the NIC is implemented using a programmable microprocessor, the logic-level changes required for GPU-TN would be simple to add in software. If the NIC is implemented using custom logic, Figure 6.2 illustrates the primary modifications. Two comparators and an incrementer can be added specifically for the purpose of performing triggered operations, or the arithmetic could utilize shared resources in a more traditional computational pipeline.

As described, trigger entries are logically organized as a linked list.

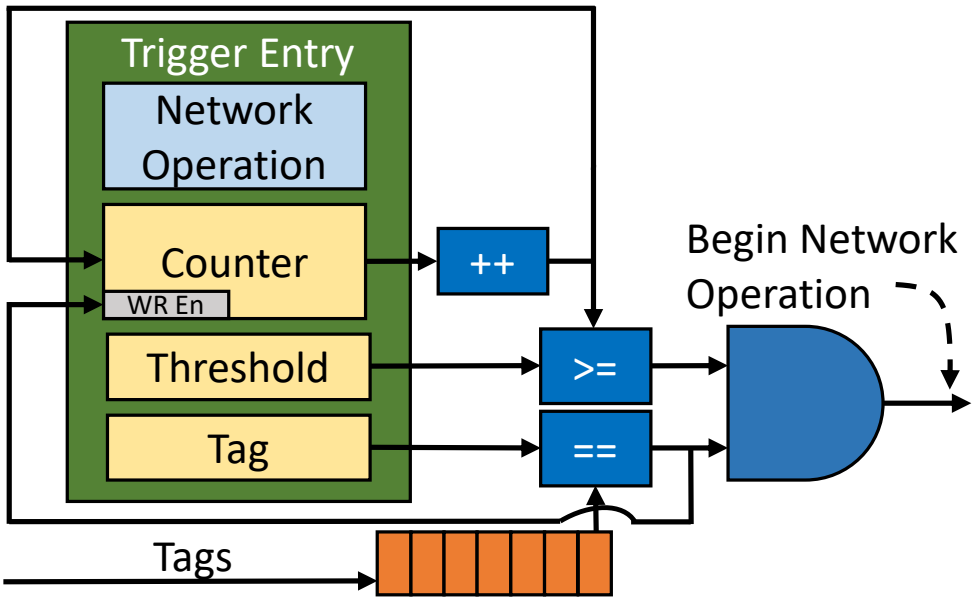



Figure 6.2: Tag matching behavior of trigger entries.

When a GPU writes to the trigger address, the NIC must be able to efficiently search the trigger list to see if there is a match. While at least one commercial product has successfully implemented hardware linked lists to satisfy the Portals 4 specification [18], simpler design alternatives can be considered to reduce hardware complexity. Additionally, the NIC needs to be able to support absorbing triggers from potentially thousands of GPU threads in quick succession, which further motivates the adoption of a lightweight trigger entry lookup.

Limiting the number of active trigger entries would make it possible to perform a simple associative lookup in hardware to perform tag matching. Alternatively, a simple hash table structure can be used to avoid extensive list

traversals. The prototype implementation evaluated in Section 6.3 requires no more than 16 simultaneous active trigger operations, which allows us to adopt the associative lookup optimization with small hardware overhead.

### 6.1.4 Dynamic Communication

GPU-TN's fundamental approach of dictating the communications pattern on the CPU imposes a static networking scheme. Buffer locations, message sizes, target nodes, and other important networking metadata are predetermined on the CPU and are not dynamically computed on the GPU. While this scheme is useful for a variety of important networking primitives and applications (see Section 6.3), it could prove limiting for some more dynamic applications.

The applications explored in this work are able to adhere to the static GPU-TN scheduling scheme, which offers the best performance at the cost of some flexibility. However, the base GPU-TN design can be extended to support more dynamic communication capabilities at the cost of some additional GPU-side control flow divergence. Instead of merely writing a tag to the NIC's trigger address, the GPU could contribute more fields dynamically, such as the input buffer pointer or target node identifier. In some sense, GPU-TN currently exists as one extreme point on a continuum of GPU networking styles that trade off performance and flexibility. A detailed treatment and analysis of a more dynamic implementation of GPU-TN is left to future work.

```
...
// ❶ Initialize RDMA comm layer
int rank = RdmaInit();
void * buf = malloc(BUFFER_SIZE);
// ❷ Register operations with the NIC
for (int i = 0; i < N_MSGS; i++)
    TrigPut(TAG + i, buf, target, thresh, ...);
// ❸ Request trigger address from NIC
char *trigAddr = GetTriggerAddr();
// ❹ Launch GPU Kernel
LaunchKern(trigAddr, TAG, N_MSGS, buf, ...);
// ❺ Cleanup, do more compute, etc.
...
```

Figure 6.3: Pseudocode illustrating the responsibilities of the host CPU in GPU-TN.

## 6.2 Programming Model

GPU-TN provides a low-level programming interface suitable for runtime library developers to implement highly optimized networking code. This section describes the host-facing API for registering triggered operations with the NIC. It also describes a number of sample GPU kernels illustrating the flexibility of the GPU-TN programming model.

### 6.2.1 Host API

The CPU-side interface of GPU-TN is responsible for performing the serial tasks of packet construction and network runtime management. Figure 6.3 shows the essential host-side steps in GPU-TN. First, the network communications runtime performs general network initialization and allocates the send buffer ❶. Then, the host code registers a number of operations

with the NIC, providing a threshold and unique tag-based identifier for every operation ❷. The NIC runtime library allocates a trigger entry for each operation. In this example, $N\_MSGS$ represents the number of messages that the GPU will trigger during its execution. Next, the memory-mapped triggered address is extracted from the networking runtime so that it can be provided to the GPU ❸. This trigger address is then passed as a kernel argument when the kernel is launched, along with one or more tags ❹. The GPU can then write one or more tags to the trigger address to increment the counter on the NIC, which will perform the network operation when this counter reaches the threshold. Finally, the CPU continues performing other useful computations and network management tasks ❺.

One important feature of GPU-TN is that steps ❹ and ❷ do not need to occur in the order presented in the example. An optimized implementation can launch the kernel at the beginning of the program and post the triggered operations to the network at a later time. This allows overlap of the network operation post and the execution of the kernel. The architecture needed to support this feature is described in Section 6.1.2.

### 6.2.2 Kernel API

Intra-kernel networking offers numerous benefits over traditional kernel-boundary communication [78, 79, 25, 51, 94, 52, 53]. In GPU-TN, network operations can be initiated as a store instruction from the perspective of the GPU; this offers a simple, yet powerful, networking interface for kernel pro-

```
__kernel void
kern1(__global char *trigAddr,
      const int tagBase,
      __global void *buffer)
{
  // do work
  ...
  buffer = ...;
  atomic_work_item_fence(...);
  int id = get_global_id(...);
  atomic_store_explicit(
    trigAddr, tagBase + id,
    ...);
  // do additional work
  ...
  ...
  ...
}
```

(a) Work-item-level networking.

```
__kernel void
kern2(__global char *trigAddr,
      const int tagBase,
      __global void *buffer)
{
  // do work
  ...
  buffer = ...;
  work_group_barrier(...);
  if (!get_local_id(...)) {
    int id = get_group_id(...);
    atomic_store_explicit(
      trigAddr, tagBase + id,
      ...);
  }
  // do additional work
  ...
}
```

(b) Work-group-level networking.

```
__kernel void
kern3(__global char *trigAddr,
      const int tag,
      __global void *buffer)
{
  // do work
  ...
  buffer = ...;
  work_group_barrier(...);
  if (!get_local_id(...)) {
    atomic_store_explicit(
      trigAddr, tag, ...);
  }
  // do additional work
  ...
}
```

(c) Kernel-level networking.

Figure 6.4: GPU kernel pseudocode illustrating how to trigger network transfers through GPU-TN for different granularities.

grammers and runtime developers. This section illustrates how GPU-TN can be supported at multiple granularities. Figure 6.4 provides example kernels for each granularity using an OpenCL-like pseudocode syntax.

#### 6.2.2.1 Work-Item/Work-Group-Level

In Figures 6.4a and 6.4b, network operations are triggered at the work-item/work-group level. Every work-item/work-group is associated with a tag, and a range of tags corresponding to the total number of work-items/work-groups are allocated to this kernel by the host and passed in as a kernel argument. The CPU-provided threshold value for triggering the operation would, in this case, be 1. The only difference between the work-item and work-group interface is the presence of a work-group barrier in the latter.

#### 6.2.2.2 Kernel-Level

Figure 6.4c shows an example where network operations are triggered at the kernel-level. Since there are currently no efficient kernel-level synchronization primitives available in OpenCL, this approach uses the counter in the trigger entry on the NIC to synchronize. Like the work-group-level example before it, each work-group writes to the trigger address using a leader work-item after a work-group-level barrier. However, only one tag is provided for the entire kernel, and the CPU provided threshold is set to the number of work-groups that need to be executed in this kernel. The NIC-resident counter is decremented and sends the message when it receives a number of writes from

the GPU equal to the number of work-groups in the kernel.

It is important to note that the above work-group and kernel messaging approaches could also be accomplished without control flow divergence by having every work-item in the work-group/kernel write the same tag and setting the NIC counter value to the number of work-items in a work-group/kernel. However, since efficient work-group barriers are available in all GPUs, memory accesses can be avoided by using the leader work-item approach.

#### 6.2.2.3 Mixed-Granularity

Additional granularities that are combinations of the above can be expressed by taking advantage of the trigger entry threshold and counter. For example, it would be simple to send a message for every pair of work-items by setting the threshold for the operation to 2 instead of 1, and using half as many tags as the single work-item approach. This offers the programmer a significant amount of freedom to experiment with different message sizes and quantities to take advantage of the natural tendencies of the underlying algorithm, and to experiment with optimal patterns for the hardware.

#### 6.2.2.4 Local Completion

Finally, while the host CPU manages the complicated NIC data structures, it is important to expose an additional hook to the GPU so that it can check completion of the network operation. For puts, this defines when it is safe for the GPU to reuse the send buffer. For gets, completion defines

when the data has been received from the target. GPU-TN simply exposes an additional global variable for each trigger operation that is set by the NIC on message completion. While this is not shown in the simple examples in Figure 6.4, the GPU threads can query this location to determine completion status of individual network operations without the complexity of monitoring a network completion queue.

#### 6.2.2.5 Target-Side Completion

GPU-TN implements a one-sided communication style described in Section 2.1.1, which fits very naturally with the hardware capabilities of the GPU [37]. Complex tag matching and deep runtime stacks present in two-sided communication paradigms like MPI introduce software complexity that is difficult to implement efficiently on GPUs. As with many one-sided communication styles, GPU-TN does not define the target-side semantics for a remote GPU to receive messages.

If the target needs to know that it has received data in the case of a put, either the host CPU or the GPU itself can monitor a network completion queue. Alternatively, many PGAS languages that leverage one-sided communication use polling on variables at the target to build notification mechanisms. More complex semantics such as execution barriers can be built out of these primitives.

#### 6.2.2.6 Scoped Memory Model Interactions

GPU-TN is a GPU-centric networking paradigm that must comply with the GPU's relaxed memory model as discussed in Section 2.2.3. By default, languages like OpenCL only provide visibility within a work-group. System scope accesses from within a GPU kernel are particularly difficult and require the use of OpenCL 2.0 atomics with the appropriate global memory scope specifier (in this case, memory_scope_all_svm_devices), which may not be supported on all current GPU devices. This limitation is only a temporary constraint, as future GPUs are likely to implement more system scope operations.

The examples in Figure 6.4 contain two interesting interactions with the GPU memory model. The first is the write to the trigAddr variable, which must be accessed using an explicit atomic store to system scope so that the GPU caches are bypassed.

The second, and more interesting, interaction concerns the network buffer itself. This buffer must be globally visible to the NIC before the write to trigAddr occurs. This is accomplished by setting the scope of the synchronization after the buffer write to the system level with release memory ordering semantics. Similarly, a system scope acquire operation must be used so that the GPU sees updates from the NIC itself.

Table 6.1: GPU-TN simulation configuration.

| CPU and Memory Configuration | |
|---|---|
| **Type** | 8-Wide OOO, x86, 8 cores @ 4Ghz |
| **I,D-Cache** | 64K, 2-way, 2 cycles |
| **L2-Cache** | 2MB, 8-way, 8 cycles |
| **L3-Cache** | 16MB, 16-way, 20 cycles |
| **System Memory** | DDR4, 8 Channels, 2133MHz |
| GPU Configuration | |
| **Type** | AMD GCN3 @1.5GHz |
| **CU Config** | 24 CUs with 4 SIMD-16 engines |
| **V-Cache** | 32kB, 16-way, 12 cycles, per CU |
| **K-Cache** | 32kB, 8-way, 12 cycles, per 4 CUs |
| **I-Cache** | 64kB, 8-way, 12 cycles, per 4 CUs |
| **L2-Cache** | 1MB, 8 banks, 16-way, 100 cycles |
| **Kernel Latencies** | 1.5$\mu$s launch / 1.5$\mu$s teardown |
| Network Configuration | |
| **Latency** | 100ns |
| **Bandwidth** | 100Gbps |
| **Topology** | Star (single switch) |

## 6.3 Evaluation

GPU-TN can offer significant performance improvements in systems utilizing networks of GPUs. This section evaluates GPU-TN over a latency microbenchmark, a 2D Jacobi relaxation stencil, an important MPI collective operation, and deep learning workloads.

### 6.3.1 Experimental Setup

The baseline simulation infrastructure was previously described in Section 3.1. Table 6.1 shows the specific configuration for the major components of the infrastructure. GPU-TN functions are implemented using an API sim-

ilar to existing Portals 4 triggered operations.

The removal of GPU kernel boundary latencies to send network messages is a major motivating factor behind GPU-TN. The simulation infrastructure is calibrated to model some of the more optimistic numbers derived from the experiments in Figure 1.2. The performance results presented for GPU-TN are based on 3$\mu$s of kernel overhead evenly divided between the launch and teardown phases. For situations where the number of available kernels exposed to the hardware scheduler at once are small, Figure 1.2 indicates that the performance improvement of GPU-TN could be even higher than the results reported in this section.

The experiments compare five different networking strategies that will be referred to as CPU, HDN, GHN, GDS, and GPU-TN. These configurations are defined as follows:

- **CPU:** All computation and communication are performed by a CPU. The CPU configuration represents a non-GPU-accelerated system and is included to separate the baseline benefits of GPU acceleration from those of GPU-TN as well as provide a sanity check for problem sizes where GPU acceleration no longer makes sense.

- **HDN:** Host-Driven Networking uses the CPU for all communication and the GPU for acceleration of workload-specific portions of the computation. Network messages are performed on GPU kernel boundaries using

two sided send/recv semantics. This represents the classic coprocessor approach to GPU networking found in most clusters.

- **GHN:** GPU Host networking places the network thread on the CPU of a host machine. This is representative of most previous works that have attempted intra-kernel networking using helper threads on the host [51, 94, 36, 82].

- **GDS:** The GDS baseline approximates the behavior of GPUDirect Async [87] kernel boundary communication in the simulation environment. GDS uses the CPU to post a sequence of network operations to the NIC. After the messages are posted, network initiation points are integrated into CUDA streams at kernel boundaries. The GPU front-end scheduling unit initiates the network operation by ringing a doorbell on the NIC after dependent kernels have completed.

- **GPU-TN:** GPU Triggered Networking uses triggered operations to efficiently communicate across nodes. Using this scheme, CPUs register network messages with the NIC. These messages are initiated from within a GPU kernel using system scope synchronization and memory-mapped writes when the network data is ready to send.

The results do not explicitly compare against GPU Native Networking approaches as defined in Section 2.3. However, it is expected that GPU-TN will offer improved latency and decreased control flow divergence, due to the

fact that the serial task of creating a network compatible command packet is offloaded to the CPU.

The results for GPU Host Networking are based on data from the APU baseline discussed in Section 5.4. Since both GPU-TN and GHN employ intra-kernel networking techniques, the performance benefit of GPU-TN is modest. The main advantage of GPU-TN over GHN approaches is that it can provide the same or better performance without requiring dedicated polling threads on the CPU. Polling threads on the CPU have a number of disadvantages, such as a lack of scalability, higher messaging latency, and the wasteful consumption of host threads that could be used for additional computation. However, this benefit is difficult to quantify in the absence of workloads that could leverage those extra threads.

### 6.3.2 Latency Analysis

This section analyzes a small microbenchmark to explore where important latencies reside in the GPU-TN networking flow. In this example, a kernel executing on an initiator node sends a message to a target node. The kernel executed by the GPU in this case is a simple vector copy operation of a single cache line and is not of particular importance. Most of the time in the GPU kernel itself is spent during kernel initialization and teardown.

Figure 6.5 illustrates a latency decomposition of the microbenchmark implemented using HDN, GDS, and GPU-TN. Both the initiator and target are separated and displayed on the same absolute time scale. In HDN, the

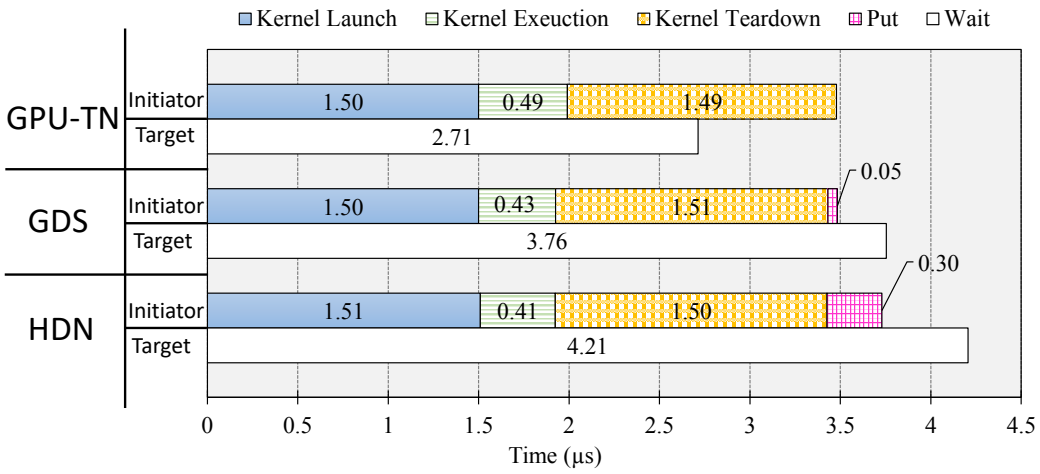



Figure 6.5: GPU-TN vs HDN vs GDS latency decomposition from a small microbenchmark.

transitions between GPU and CPU control flow are obvious; after a kernel completes, the CPU initiates a network operation to send data to the target. The target polls on a memory location to determine when the data has been sent.

For the GDS baseline, the GPU itself initiates the communication after the kernel has finished execution. The control flow switch from the GPU back to the CPU is avoided as well as the critical path construction of the network packet (in GDS, network operations are posted before-hand by the CPU). GDS results in around a 10% reduction in latency over the HDN baseline. However, it should be noted that a system architecture employing a more traditional discrete GPU setup could see much larger performance improvement from GDS, since it would avoid a costly critical path control flow switch over the I/O bus.

There are two distinct differences when comparing the GDS implementation against GPU-TN. The first is that in GPU-TN, network operations are initiated in the kernel itself, causing the execution of a GPU-TN kernel to take slightly longer than the corresponding GDS kernel. The second observation is that the target node receives the network data before the kernel on the initiator completes. This phenomenon is a direct result of GPU-TN's intra-kernel networking. The network message does not need to wait for kernel termination before sending the message; a kernel can initiate a network operation whenever the data is ready. Overall, the GPU-TN approach achieves approximately 25% performance improvement over GDS, and approximately 35% improvement over HDN.

### 6.3.3 Jacobi 2D Stencil

This section evaluates the performance of GPU-TN over a 2D Jacobi relaxation problem [55] with various input sizes. The Jacobi relaxation is implemented over the 4 example systems by splitting the input in 2D. CPU is a standard implementation of Jacobi using OpenMP for thread-level parallelism, while HDN is implemented by exiting the kernel and returning to the host for MPI send/receives after every round. GHN provides intra-kernel networking and forwards messages from the GPU to the host using helper threads on the CPU. GDS and GPU-TN both pre-register the communication, which is known beforehand since the communication is highly structured. The difference between GDS and GPU-TN is that GDS stops and starts a kernel every

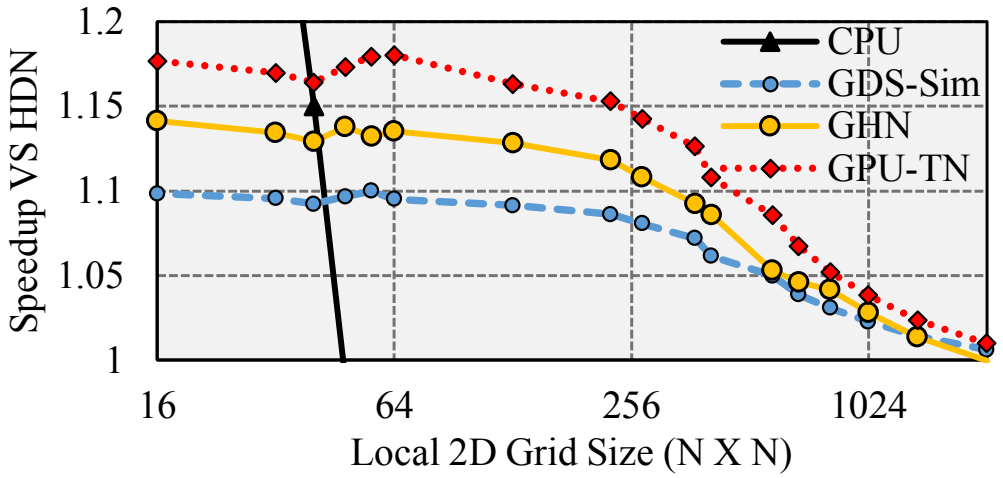



Figure 6.6: Performance on a single iteration of a 2D Jacobi Relaxation computation over different NxN grid sizes.

time, and GPU-TN uses a single kernel for the entire duration of the program.

Figure 6.6 illustrates the results of the Jacobi relaxation on the sample systems. The results are presented as speedup to the HDN baseline and represent a single iteration of Jacobi with varying local problem sizes. When strong scaling Jacobi, one would move "left" on the graph, while weak scaling would stay at the same point, since the communication patterns do not significantly change with the introduction of more nodes. Overall, GPU-TN achieves approximately 6% improvement over GDS, approximately 18% improvement over HDN, and 3% speedup over GHN on medium problem sizes. CPU results are included in the figure to make sure that the range of problem sizes GPU-TN offers benefits on do not fall outside what is useful to offload onto a GPU.

### 6.3.4 Allreduce

Collective operations on clusters of GPUs are a critical primitive operation for a large number of applications, including deep learning, parallel FFT, molecular dynamics, and graph analytics [72, 21]. This section uses GPU-TN to implement the Allreduce collective operation in MPI.

GPU-TN is used to accelerate allreduce through the libNBC [41] non-blocking collectives library. When a collective application is called from the application, libNBC creates a schedule of subtasks that completely define all operations and dependencies. In this manner, the collective operation is performed asynchronously by stepping through the schedule of tasks in the MPI runtime itself. Schedule creation in libNBC maps perfectly to the triggered operation semantics in GPU-TN. Indeed, collective operations were one of the original motivations for the introduction of triggered network semantics [98].

The Allreduce algorithm is implemented on CPU, GDS, GPU-TN, GHN, and HDN systems. The implementations of CPU, GDS, and HDN are similar to what was described in the Jacobi benchmark evaluation. The implementation of GHN for Allreduce was described in the evaluation section for ComP-Net. In GPU-TN, the entire collective operation is performed from within a single GPU kernel. The GPU kernel polls on a memory location to know when an adjacent node has contributed data for the reduction. The GPU work-items then perform the arithmetic operation and triggers the GPU to send data for the next phase. The implementation triggers the network operation at the granularity of a work-group; this allows for easy software

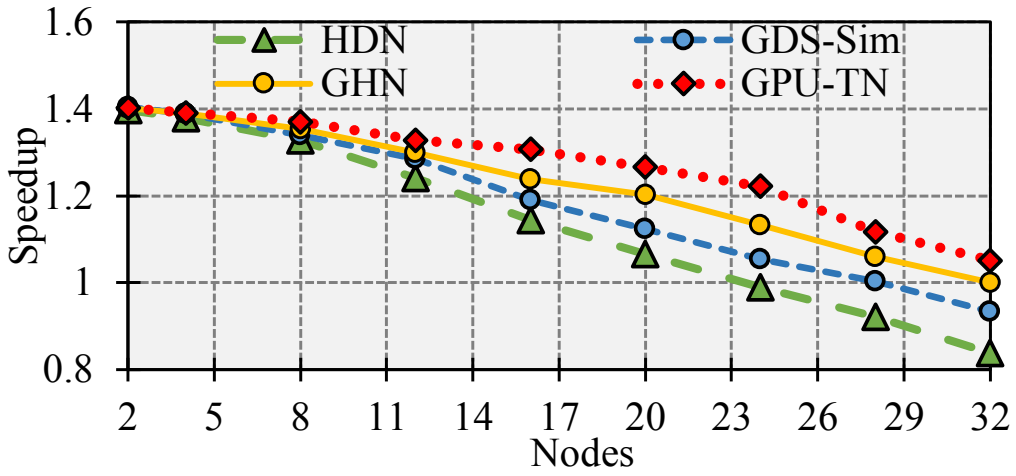



Figure 6.7: GPU-TN strong scaling performance evaluation on an 8MB MPI Allreduce collective operation.

pipelining of the computation and network transfer.

Figure 6.7 shows a strong-scaling study of an 8MB collective operation on all the evaluated configurations. In this example, the data is single-precision floating point and the operation is a simple binary addition. Results are reported as speedup relative to the same operation occurring entirely on the CPU. For large payload sizes (i.e., small node counts), HDN, GPU-TN, and GDS provide roughly 1.4x speedup over an optimized CPU Allreduce operation. In this case, the savings gained from quick network initiation in GPU-TN are dwarfed by the transfer and computation time. However, as the payload size of each reduction message decreases (i.e., as node count increases) GPU-TN provides significantly more speed-up over HDN and GDS. At approximately 24 nodes, HDN Allreduce operations actually become slower than the equivalent operation performed on a CPU, while GPU-TN continues

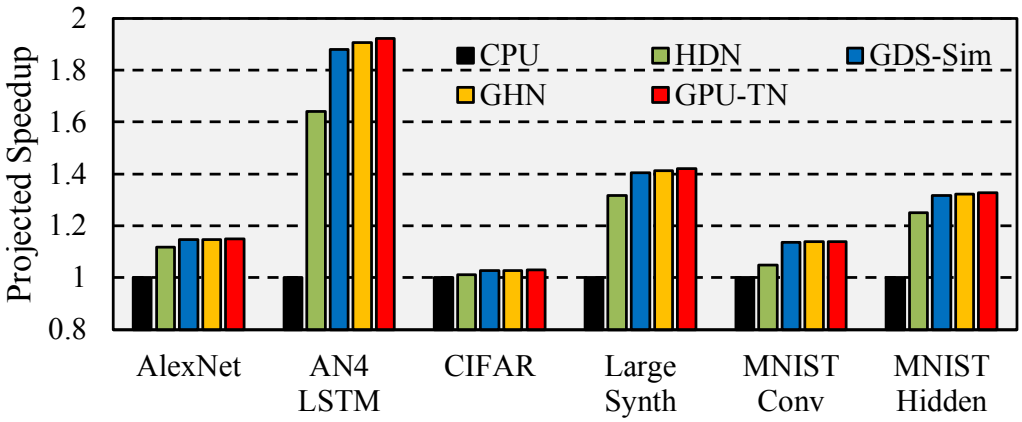



Figure 6.8: GPU-TN performance across six deep learning workloads on a cluster of 8 nodes.

to provide speedup into 32 nodes and beyond.

### 6.3.5 Machine Learning

This section demonstrates GPU-TN's ability to accelerate the performance of workloads written in a popular machine learning framework, Microsoft's Cognitive Toolkit [2]. Figure 6.8 shows how GPU-TN can be used to accelerate training of neural networks on the Microsoft Cognitive Toolkit [2] deep learning platform on a cluster of 8 nodes. Results vary from little improvement as in the CIFAR workload up to approximately 20% improvement over HDN and 5% improvement over GDS in AN4 LSTM. This variability has to do with the different characteristics of Allreduce operations found in these workloads. The frequency and size of Allreduce operations are dependent on the type of neural network being trained, the number of participating nodes, and the characteristics of the input data set described in Table 3.1. GPU-

TN provides the most benefit in scenarios where there are a large number of small-to-medium-sized collective operations. In all workloads, the performance difference between GPU-TN and GHN is negligible.

## 6.4 Conclusion

This chapter introduced GPU Triggered Networking (GPU-TN), a new networking scheme that can combine the best of traditional host networking approaches without critical path CPU interactions when initiating network messages from within a GPU kernel. In GPU-TN, triggered operations are used to pre-register network operations on the NIC that can be later triggered by the GPU using a simple memory-mapped write operation. GPU-TN networking decouples the CPU and GPU, while still allowing the CPU to perform serial networking tasks that are not easily implemented on a GPU. Additionally, GPU-TN can provide variable granularities of messaging to support a variety of programming paradigms, while relaxing the kernel-boundary networking restriction that can impair performance on competing approaches.

GPU-TN was evaluated across a latency microbenchmark, a 2D Jacobi relaxation stencil, the important Allreduce collective operation, and emerging machine learning workloads. GPU-TN is capable of achieving up to 25% performance improvement against a simulated GDS solution, and up to 35% performance improvement against a traditional Host Driven Networking approach at scales of up to 32 nodes.

# Chapter 7

# Conclusions

GPUs are frequently deployed to efficiently accelerate a number of important data-parallel applications from a variety of fields. To solve the largest problems quickly, researchers and data scientists employ clusters of GPU-enabled nodes connected over a high-performance computer network. Unfortunately, modern GPUs are forced to channel communication through a driver stack on the host CPU, despite the availability of host-bypass RDMA networks. Often, the intra-node overheads associated with setting up a network transfer dwarf the overheads associated with the transfer itself. This work explores mechanisms to improve the performance of GPU networking using software and hardware techniques innovations to enable GPUs as first-class networking clients.

## 7.1 Summary

This dissertation provides three contributions to enhance GPU networking. First, Extended Task Queuing (XTQ) [59] provides the ability to launch remote kernels without intervention of a host CPU at the target. Inspired by classic work on active messaging [27], XTQ uses NIC architectural

modifications to support remote kernel launch without the participation of the remote CPU. Bypassing the remote CPU significantly reduces remote kernel launch latencies and allows a more decentralized, cluster-wide work dispatch system that supports task-based runtime systems.

Next, intra-kernel, GPU-initiated communication is optimized through the ComP-Net framework. ComP-Net uses a little-known feature of modern GPUs: embedded, programmable microprocessors that are typically referred to as Command Processors (CPs). By running the network software stack on the CP instead of the host CPU, GPU communication latency is decreased. ComP-Net implements a runtime and programming interface that allows the GPU compute units to take advantage of the unique capabilities of a networking CP. Challenges related to the GPU's relaxed memory model and L2 cache thrashing are addressed to reduce the latency of network communication.

Finally, GPU Triggered Networking (GPU-TN) [58] provides an alternative intra-kernel networking scheme for a GPU to directly trigger network operations from within a GPU kernel without the involvement of any CPU on the critical path. Inspired by Portals 4 triggered operations [89], GPU Triggered Networking implements a NIC hardware mechanism by which the GPU can directly trigger the network adapter to send messages. In this approach, the host CPU is responsible for creating the network command packet on behalf of the GPU and registering it with the NIC. When the GPU is ready to send a message, it simply "triggers" the NIC using a memory-mapped store operation. A small amount of additional hardware in the NIC collects these

Table 7.1: Comparison of prior art and proposed GPU networking techniques.

| Networking Strategy | Kernel Boundary | GPU Triggered | GPU Overhead | CPU Overhead |
|---|---|---|---|---|
| Host-Driven Networking [64, 29, 93] | Yes | No | - | Network Stack |
| GPU Native Networking [78, 79, 52, 53, 25] | No | Yes | Network Stack | - |
| GPU Host Networking [51, 94, 36, 67, 82] | No | No | CPU/GPU Queue Management | Service Threads, Network Stack |
| GPU Direct Async (GDS) [87] | Yes | Yes | Network Trigger | Partial Network Stack |
| Extended Task Queuing [59] | Yes (Optimized) | No | - | Network Stack (Initiator Only) |
| GPU Triggered Networking [58] | No | Yes | Network Trigger | Partial Network Stack |
| Command Processor Networking [57] | No | No | CPU/GPU Queue Management | Service Threads, Network Stack (On CP) |

writes from the GPU and initiates the pending network operation when a threshold condition has been met. These optimizations allow for extremely fine-grained communications remote communication without ending a kernel.

## 7.2 Qualitative Comparison of Proposed Techniques

The three techniques described in this work all improve GPU networking performance over the prior art. However, each technique approaches the problem from a different perspective. There are certain applications or paradigms that may prefer one optimization or programming model over another. Table 7.1 provides a qualitative overview of the proposed techniques and

their relationship to each other and the prior art. This rest of this section will discuss each technique and its relationship to the other proposed techniques and applications.

Porting applications to the interfaces presented in this dissertation can vary in difficulty based on the technique chosen and the design of the original program. XTQ can be more easily applied to existing distributed GPU applications than ComP-Net or GPU-TN. This observation is due to the fact that most existing distributed GPU applications assume that network operations are only available at kernel boundaries. XTQ is a kernel boundary networking API while ComP-Net and GPU-TN are both implementations of the intra-kernel networking programming paradigm. However, applications that are currently designed to run on a single GPU might prefer the API presented by GPU-TN or ComP-Net. Intra-kernel networking enables existing single node GPU applications to access the network without breaking the application into additional kernels.

While the ComP-Net and GPU-TN both offer intra-kernel interfaces, there are differences in performance and flexibility. The key difference between ComP-Net and GPU-TN is the flexibility of the programming model. GPU-TN advocates pre-registration of messages when the communication pattern is static, while ComP-Net uses a CP thread to dynamically create network requests as needed. Although this dissertation does not directly compare ComP-Net vs. GPU-TN, it is expected that GPU-TN will perform better than ComP-Net at the cost of requiring the communications pattern to be known

beforehand. This is because the cost of triggering a network communication through GPU-TN is a single write operation from the GPU, while ComP-Net must forward the networking results to the Command Processor threads.

Finally, while each optimization is presented independently, it is important to note that the designs are complimentary to each other. There is no reason that each design could not exist in the same system as the others, or that the techniques could be used concurrently, depending on the needs of the application. For example, XTQ could be used to launch a remote kernel which takes advantage of the intra-kernel networking techniques enabled by ComP-Net or GPU-TN. GPU-TN itself could be slightly modified so that the the thread that is responsible for pre-registering the network command operation resides on the CP instead of the host CPU, resulting in a design that is a hybrid of ComP-Net and GPU-TN. A complete evaluation of the utility of hybrid approaches on real application is left to future work.

## 7.3 Future Work

There are a number of future research directions that can extend and enhance the GPU communication techniques described in this dissertation. This section explores a number of promising next-steps to optimize GPU networking and the use of GPU networking primitives in applications.

### 7.3.1 Application Studies

Most GPU applications that currently exist are designed around the limitations of existing GPU networking solutions. Practically, this means that multi-node GPU applications must overlap communication and computation at a very coarse granularity in order to hide the overheads associated with networking. While the high performance GPU networking techniques presented in this dissertation are shown to accelerate a number of MPI primitives, stencils, and machine learning applications, there remains a significant amount of work in this area.

This is especially true for the two presented intra-kernel networking techniques, ComP-Net and GPU-TN. Initiating networking from within a kernel is especially useful for applications where the communication patters are irregular. One particular class of applications that might benefit from intra-kernel networking are those that form a graph with dependencies across nodes.

One example of an application exhibiting this characteristic is graph processing algorithms. Most real world graphs are by their very nature highly irregular. However, they still contain a massive amount of parallelism that makes some algorithms amenable to GPU acceleration. Many graph processing frameworks that are designed to span more than one node use very lightweight partitioning algorithms, which generates a great deal of communication [56]. Some work towards supporting graph analytics through GPU intra-kernel networking has been explored by Orr et al. [82], but there are still many opportunities to explore further.

Another interesting area for optimized GPU networking is sparse triangular solvers that serve as an important component in a large number of numerical linear algebra routines, such as least squares problems, preconditioned iterative methods, and direct methods. While sparse triangular solvers still have a large amount of parallelism, they have been traditionally difficult to accelerate on GPUs due to the presence of many data dependencies. Most modern implementations on single GPUs involve breaking the problem down into a large DAG which is then processed by the individual work-groups. These work-groups activate other work-groups with dependent inputs when computation is complete [69]. While the DAG is essentially static throughout execution, spreading the nodes across a number of machines will cause a great deal of irregular communication that requires an efficient GPU networking implementation.

Finally, there have been a great deal of advances in training neural networks at scale since the research conducted in this dissertation. The models and training methodology employed in this work assumes a data-parallel distribution and synchronous training using an Allreduce computation. However, recent algorithmic advances have tried to mitigate the amount of network communication performed by machine learning at scale [91, 61, 100]. Additionally, researchers have proposed asynchronous training methods that more aggressively overlaps communication and computation, potentially reducing the amount of time that a node is blocked with gradient updates [26, 20]. In these algorithms, dedicated parameter servers hold the gradients and asynchronously

receive updates from all training nodes using point-to-point communication. These techniques could potentially reduce the performance improvement offered by the techniques presented in this dissertation, since the algorithms themselves make networking less of a bottleneck to overall application performance. A complete study of the impact of these modern algorithms on the GPU networking techniques discussed in this dissertation is left as future work.

### 7.3.2 Leveraging Emerging NIC Technologies for GPUs

Recent advances in networking seek to add programmable elements to high-performance RDMA NICs. These so called "smart NICs" provide additional intelligence to allow users to perform advanced packet routing [50] [49], implement programmable message handlers [39], or accelerate collective operations in the network itself [88] [4] [32]. Even more aggressive designs, such as Mellanox's BlueField [65], propose adding a number of full-on processor cores to the NIC itself.

The work in this dissertation largely assumed traditional high performance networks that contain a minimal amount of offload capability. Smarter networks offer an alternative strategy to perform efficient networking on a GPU apart from new hardware. As an example, both XTQ and GPU-TN could both implement their NIC-side logic on a programmable smart NIC with a proper interface. The CP-side service thread on ComP-Net could also be located on the NIC itself. A complete treatment of how emerging smart

NICs and offloads can be leveraged to accelerate GPU networking is a rich area for future work.

# Bibliography

[1] A. Agarwal, R. Bianchini, D. Chaiken, K. Johnson, D. Kranz, J. Kubiatowicz, B.-H. Lim, K. Mackenzie, and D. Yeung, "The MIT Alewife Machine: Architecture and Performance," in *Intl. Symp. on Computer Architecture (ISCA)*, 1995, pp. 2–13.

[2] A. Agarwal, E. Akchurin, C. Basoglu, G. Chen, S. Cyphers, J. Droppo, A. Eversole, B. Guenter, M. Hillebrand, T. R. Hoens, X. Huang, Z. Huang, V. Ivanov, A. Kamenev, P. Kranen, O. Kuchaiev, W. Manousek, A. May, B. Mitra, O. Nano, G. Navarro, A. Orlov, H. Parthasarathi, B. Peng, M. Radmilac, A. Reznichenko, F. Seide, M. L. Seltzer, M. Slaney, A. Stolcke, H. Wang, Y. Wang, K. Yao, D. Yu, Y. Zhang, and G. Zweig, "An Introduction to Computational Networks and the Computational Network Toolkit," Microsoft, Technical Report, 2014. [Online]. Available: https://www.microsoft.com/en-us/research/wp-content/uploads/2014/08/CNTKBook-20160217.pdf

[3] E. Agostini, D. Rossetti, and S. Potluri, "Offloading Communication Control Logic in GPU Accelerated Applications," in *Intl. Symp. on Cluster, Cloud and Grid Computing (CCGrid)*, 2017.

[4] B. Alverson, E. Froese, L. Kaplan, and D. Roweth. (2015) Cray

XC Series Network. [Online]. Available: http://www.cray.com/sites/default/files/resources/CrayXC30Networking.pdf

[5] AMD. (2015) AMD I/O Virtualization Technology (IOMMU) Specification. [Online]. Available: http://support.amd.com/TechDocs/48882_IOMMU.pdf

[6] AMD. (2015) The AMD gem5 APU Simulator: Modeling Heterogeneous Systems in gem5. [Online]. Available: http://gem5.org/GPU_Models

[7] AMD. (2017) Graphics Core Next Architecture, Generation 3 ISA. [Online]. Available: http://gpuopen.com/compute-product/amd-gcn3-isa-architecture-manual/

[8] AMD. (2018) HIP: Heterogeneous-Computing Interface for Portability. [Online]. Available: http://rocm-developer-tools.github.io/HIP/

[9] AMD. (2018) ROCn RDMA. [Online]. Available: https://github.com/RadeonOpenCompute/ROCnRDMA

[10] B. S. Ang, D. Chiou, D. Rosenband, M. Ehrlich, L. Rudolph, and Arvind, "StarT-Voyager: A Flexible Platform for Exploring Scalable SMP Issues," in *Intl. Conf. for High Performance Computing, Networking, Storage and Analysis (SC)*, 1998, pp. 1–13.

[11] Baidu. (2018) Baidu-Allreduce. [Online]. Available: https://github.com/baidu-research/baidu-allreduce

[12] M. Baker, S. Boehm, A. Bouteiller, B. Chapman, R. Cernohous, J. Culhane, T. Curtis, J. Dinan, M. Dubman, K. Feind, M. G. Venkata, M. Grossman, K. Hamidouche, J. Hammond, Y. Itigin, B. Lam, D. Knaak, J. Kuehn, J. Manser, T. M. Mintz, D. Ozog, N. Park, S. Poole, W. Poole, S. Pophale, S. Potluri, H. Pritchard, N. Ravichandrasekaran, M. Raymond, J. Ross, P. Shamis, S. Shende, and L. Smith. (2018) OpenSHMEM Specification. [Online]. Available: http://openshmem.org/site/sites/default/site_files/OpenSHMEM-1.4.pdf

[13] M. Ben-Yehuda, J. Mason, L. Van Doorn, and E. Wahlig, "Utilizing IOMMUs for Virtualization in Linux and Xen," in *In proc. of the Linux Symp.*, 2006.

[14] M. Besta and T. Hoefler, "Active Access: A Mechanism for High-Performance Distributed Data-Centric Computations," in *Intl. Conf. on Supercomputing (ICS)*, 2015, pp. 155–164.

[15] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The Gem5 Simulator," *SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, Aug. 2011.

[16] M. S. Birrittella, M. Debbage, R. Huggahalli, J. Kunz, T. Lovett, T. Rimmer, K. D. Underwood, and R. C. Zak, "Intel Omni-Path

Architecture Technology Overview," 2015. [Online]. Available: http://www.hoti.org/hoti23/slides/rimmer.pdf

[17] D. Bonachea. (2017) GASNet Specification, v1.8.1. [Online]. Available: https://gasnet.lbl.gov/dist/docs/gasnet.pdf

[18] Bull. (2017) BXI: Bull eXascale Interconnect. [Online]. Available: https://atos.net/en/products/high-performance-computing-hpc/bxi-bull-exascale-interconnect

[19] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, "X10: An Object-oriented Approach to Non-uniform Cluster Computing," in *Conf. on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2005, pp. 519–538.

[20] T. Chilimbi, Y. Suzue, J. Apacible, and K. Kalyanaraman, "Project Adam: Building an Efficient and Scalable Deep Learning Training System," in *Proc. of the 11th USENIX Conf. on Operating Systems Design and Implementation (OSDI)*, 2014, pp. 571–582.

[21] C.-H. Chu, K. Hamidouche, A. Venkatesh, A. A. Awan, and D. K. Panda, "CUDA Kernel Based Collective Reduction Operations on Large-scale GPU Clusters," in *Intl. Symp. on Cluster, Cloud and Grid Computing (CCGrid)*, 2016, pp. 726–735.

[22] Cray. (2015) The Chapel Parallel Programming Language. [Online]. Available: http://chapel.cray.com/

[23] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken, "LogP: Towards a Realistic Model of Parallel Computation," in *Symp. on Principles and Practice of Parallel Programming (PPOPP)*, 1993, pp. 1–12.

[24] G. F. Dan Bonachea, "UPC Language Specifications, Version 1.3," Lawrence Berkeley National Lab (LBNL), Tech. Rep., 2013. [Online]. Available: http://upc.lbl.gov/publications/upc-spec-1.3.pdf

[25] F. Daoud, A. Watad, and M. Silberstein, "GPUrdma: GPU-side Library for High Performance Networking from GPU Kernels," in *Intl. Workshop on Runtime and Operating Systems for Supercomputers (ROSS)*, 2016, pp. 6:1–6:8.

[26] J. Dean, G. S. Corrado, R. Monga, K. Chen, M. Devin, Q. V. Le, M. Z. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang, and A. Y. Ng, "Large Scale Distributed Deep Networks," in *Proc. of the 25th Intl. Conf. on Neural Information Processing Systems - Volume 1 (NIPS)*, 2012, pp. 1223–1231.

[27] T. Eicken, D. Culler, S. Goldstein, and K. Schauser, "Active Messages: A Mechanism for Integrated Communication and Computation," in *Intl. Symp. on Computer Architecture (ISCA)*, 1992, pp. 256–266.

[28] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger, "Dark Silicon and the End of Multicore Scaling," in *Intl. Symp. on Computer Architecture (ISCA)*, 2011, pp. 365–376.

[29] Z. Fan, F. Qiu, and A. E. Kaufman, "Zippy: A Framework for Computation and Visualization on a GPU Cluster," *Computer Graphics Forum*, vol. 27, no. 2, pp. 341–350, 2008.

[30] M. Fillo, S. W. Keckler, W. J. Dally, N. P. Carter, A. Chang, Y. Gurevich, and W. S. Lee, "The M-Machine Multicomputer," in *Intl. Symp. on Microarchitecture (MICRO)*, 1995, pp. 146–156.

[31] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall, "Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation," in *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*, 2004, pp. 97–104.

[32] R. L. Graham, S. Poole, P. Shamis, G. Bloch, N. Bloch, H. Chapman, M. Kagan, A. Shahar, I. Rabinovitz, and G. Shainer, "ConnectX-2 InfiniBand Management Queues: First Investigation of the New Support for Network Offloaded Collective Operations," in *Intl. Conf. on Cluster, Cloud and Grid Computing (CCGrid)*, 2010, pp. 53–62.

[33] R. E. Grant, A. Skjellum, and V. Purushotham, "Lightweight Threading

with MPI using Persistent Communications Semantics," in *Workshop on Exascale MPI (ExaMPI)*, 2015.

[34] D. Grünewald and C. Simmendinger, "The GASPI API Specification and its Implementation GPI 2.0," in *Intl. Conf. on Partitioned Global Address Space Programming Models (PGAS)*, 2013.

[35] A. Gutierrez, B. M. Beckmann, A. Dutu, J. Gross, M. LeBeane, J. Kalamatianos, O. Kayiran, M. Poremba, B. Potter, S. Puthoor, M. D. Sinclair, M. Wyse, J. Yin, X. Zhang, A. Jain, and T. Rogers, "Lost in Abstraction: Pitfalls of Analyzing GPUs at the Intermediate Language Level," in *Intl. Symp. on High Performance Computer Architecture (HPCA)*, 2018, pp. 608–619.

[36] T. Gysi, J. Bär, and T. Hoefler, "dCUDA: Hardware Supported Overlap of Computation and Communication," in *Intl. Conf. for High Performance Computing, Networking, Storage and Analysis (SC)*, 2016, pp. 52:1–52:12.

[37] K. Hamidouche, A. Venkatesh, A. A. Awan, H. Subramoni, C.-H. Chu, and D. K. Panda, "CUDA-Aware OpenSHMEM: Extensions and Designs for High Performance OpenSHMEM on GPU Clusters," *Parallel Computing*, vol. 58, pp. 27–36, 2016.

[38] J. Heinlein, K. Gharachorloo, S. Dresser, and A. Gupta, "Integration of Message Passing and Shared Memory in the Stanford FLASH Mul-

tiprocessor," in *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1994, pp. 38–50.

[39] T. Hoefler, S. Di Girolamo, K. Taranov, R. E. Grant, and R. Brightwell, "sPIN: High-performance streaming Processing In the Network," in *Intl. Conf. for High Performance Computing, Networking, Storage and Analysis (SC)*, 2017, pp. 59:1–59:16.

[40] T. Hoefler and A. Lumsdaine, "Message Progression in Parallel Computing - To Thread or not to Thread?" in *Intl. Conf. on Cluster Computing (Cluster)*, 2008, pp. 213–222.

[41] T. Hoefler, A. Lumsdaine, and W. Rehm, "Implementation and Performance Analysis of Non-Blocking Collective Operations for MPI," in *Proc. of the Intl. Conf. for High Performance Computing, Networking, Storage and Analysis (SC)*, 2007, pp. 52:1–52:10.

[42] T. Hoefler, T. Schneider, and A. Lumsdaine, "LogGOPSim: Simulating Large-Scale Applications in the LogGOPS Model," in *Proc. of the Intl. Symp. on High Performance Distributed Computing (HPDC)*, 2010, pp. 597–604.

[43] K. E. Hoff III, J. Keyser, M. Lin, D. Manocha, and T. Culver, "Fast Computation of Generalized Voronoi Diagrams using Graphics Hardware," in *Intl. Conf. on Computer Graphics and Interactive Techniques (SIGGRAPH)*, 1999, pp. 277–286.

[44] 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 *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.

[45] HSA Foundation. (2018) HSA Platform System Architecture Specification 1.2. [Online]. Available: http://www.hsafoundation.com/standards/

[46] InfiniBand Trade Association. (2014) RDMA over Converged Ethernet v2. [Online]. Available: https://cw.infinibandta.org/document/dl/7781

[47] InfiniBand Trade Association. (2016) InfiniBand Architecture Specification: Release 1.3.1. [Online]. Available: http://www.infinibandta.org/content/pages.php?pg=technology_download

[48] Intel. (2010) Internet Wide Area RDMA Protocol (iWARP). [Online]. Available: http://www.intel.com/content/dam/doc/technology-brief/iwarp-brief.pdf

[49] Intel. (2018) Intel IXP45X and Intel IXP46X Product Line of Network Processor. [Online]. Available: https://www.intel.com/content/dam/www/public/us/en/documents/datasheets/ixp45x-ixp46x-product-line-network-processors-datasheet.pdf

[50] A. Kaufmann, S. Peter, N. K. Sharma, T. Anderson, and A. Krishnamurthy, "High Performance Packet Processing with FlexNIC," in *Intl.*

*Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2016, pp. 67–81.

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

[52] B. Klenk, L. Oden, and H. Fröning, "Analyzing Put/Get APIs for Thread-Collaborative Processors," in *Intl. Conf. on Parallel Processing (ICPP) Workshops*, 2014, pp. 411–418.

[53] B. Klenk, L. Oden, and H. Fröning, "Analyzing Communication Models for Distributed Thread-Collaborative Processors in Terms of Energy and Time," in *Intl. Symp. on Performance Analysis of Systems and Software (ISPASS)*, 2015.

[54] S. Kumar, G. Dozsa, G. Almasi, P. Heidelberger, D. Chen, M. E. Giampapa, M. Blocksome, A. Faraj, J. Parker, J. Ratterman, B. Smith, and C. J. Archer, "The Deep Computing Messaging Framework: Generalized Scalable Message Passing on the Blue Gene/P Supercomputer," in *Intl. Conf. on Supercomputing (ICS)*, 2008, pp. 94–103.

[55] J. Lambers. (2010) Jacobi Methods. [Online]. Available: http://web.stanford.edu/class/cme335/lecture7.pdf

[56] M. LeBeane, S. Song, R. Panda, J. H. Ryoo, and L. K. John, "Data Partitioning Strategies for Graph Workloads on Heterogeneous Clusters," in *Intl. Conf. for High Performance Computing, Networking, Storage and Analysis (SC)*, 2015, pp. 56:1–56:12.

[57] M. LeBeane, K. Hamidouche, B. Benton, M. Breternitz, and S. K. Reinhardt, "ComP-Net: Command Processor Networking for Efficient Intra-kernel Communications on GPUs," in *Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT)*, 2018.

[58] M. LeBeane, K. Hamidouche, B. Benton, M. Breternitz, S. K. Reinhardt, and L. K. John, "GPU Triggered Networking for Intra-Kernel Communications," in *Proc. of the Intl. Conf. for High Performance Computing, Networking, Storage and Analysis (SC)*, 2017, pp. 22:1–22:12.

[59] M. LeBeane, B. Potter, A. Pan, A. Dutu, V. Agarwala, W. Lee, D. Majeti, B. Ghimire, E. Van Tassell, S. Wasmundt, B. Benton, M. Breternitz, M. L. Chu, M. Thottethodi, L. K. John, and S. K. Reinhardt, "Extended Task Queuing: Active Messages for Heterogeneous Systems," in *Proc. of the Intl. Conf. for High Performance Computing, Networking, Storage and Analysis (SC)*, 2016, pp. 80:1–80:12.

[60] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures," in *Intl. Symp. on Microarchitecture (MICRO)*, 2009, pp. 469–480.

[61] Y. Lin, S. Han, H. Mao, Y. Wang, and W. J. Dally, "Deep Gradient Compression: Reducing the Communication Bandwidth for Distributed Training," in *Intl. Conf. on Learning Representations (ICLR)*, 2018.

[62] E. Lindholm, M. J. Kilgard, and H. Moreton, "A User-Programmable Vertex Engine," in *Intl. Conf. on Computer Graphics and Interactive Techniques (SIGGRAPH)*. ACM, 2001, pp. 149–158.

[63] Mellanox. (2015) EDR InfiniBand. [Online]. Available: https://www.openfabrics.org/images/eventpresos/workshops2015/UGWorkshop/Friday/friday_01.pdf

[64] Mellanox. (2017) Mellanox OFED GPUDirect RDMA. [Online]. Available: http://www.mellanox.com/page/products_dyn?product_family=116

[65] Mellanox. (2018) BlueField Multicore System on Chip. [Online]. Available: http://www.mellanox.com/related-docs/npu-multicore-processors/PB_Bluefield_SoC.pdf

[66] Mellanox. (2018) Introducing 200G HDR InfiniBand Solutions. [Online]. Available: http://www.mellanox.com/related-docs/whitepapers/WP_Introducing_200G_HDR_InfiniBand_Solutions.pdf

[67] T. Miyoshi, H. Irie, K. Shima, H. Honda, M. Kondo, and T. Yoshinaga, "FLAT: A GPU Programming Framework to Provide Embedded MPI,"

in *Workshop on General Purpose Processing with Graphics Processing Units (GPGPU)*, 2012, pp. 20–29.

[68] MPI Forum. (2015) MPI: A Message-Passing Interface Standard. Ver. 3.1. [Online]. Available: https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf

[69] M. Naumov, "Parallel Solution of Sparse Triangular Linear Systems in the Preconditioned Iterative Methods on the GPU," Nvidia, Tech. Rep. [Online]. Available: http://research.nvidia.com/sites/default/files/publications/nvr-2011-001.pdf

[70] J. Nieplocha and B. Carpenter, "ARMCI: A Portable Remote Memory Copy Library for Distributed Array Libraries and Compiler Run-time Systems," in *Intl. Parallel Processing Symp. (IPPS)*, 1999, pp. 533–546.

[71] M. D. Noakes, D. A. Wallach, and W. J. Dally, "The J-Machine Multicomputer," in *Intl. Symp. on Computer Architecture (ISCA)*, 1993, pp. 224–235.

[72] Nvidia. (2016) Fast Multi-GPU collectives with NCCL. [Online]. Available: https://devblogs.nvidia.com/parallelforall/fast-multi-gpu-collectives-nccl/

[73] Nvidia. (2018) CUDA Toolkit 9.2. [Online]. Available: https://developer.nvidia.com/cuda-toolkit

[74] Nvidia. (2018) GPU-Accelerated Applications. [Online]. Available: https://www.nvidia.com/content/gpu-applications/PDF/gpu-applications-catalog.pdf

[75] Nvidia. (2018) Nvidia DGX-2. [Online]. Available: https://www.nvidia.com/en-us/data-center/dgx-2/

[76] Nvidia. (2018) Nvidia Tesla v100. [Online]. Available: https://www.nvidia.com/en-us/data-center/tesla-v100/

[77] Nvidia. (2018) NVlink Fabric. [Online]. Available: https://www.nvidia.com/en-us/data-center/nvlink/

[78] L. Oden and H. Fröning, "GGAS: Global GPU Address Spaces for Efficient Communication in Heterogeneous Clusters," in *Intl. Conf. on Cluster Computing (CLUSTER)*, 2013, pp. 1–8.

[79] L. Oden, H. Fröning, and F.-J. Pfreundt, "InfiniBand-Verbs on GPU: A Case Study of Controlling an InfiniBand Network Device from the GPU," in *Intl. Conf. on Parallel Distributed Processing Symp. Workshops (IPDPSW)*, 2014, pp. 976–983.

[80] OpenCAPI Consortium. (2018) OpenCAPI Specification. [Online]. Available: https://opencapi.org/technical/specifications/

[81] M. S. Orr, B. M. Beckmann, S. K. Reinhardt, and D. A. Wood, "Fine-grain Task Aggregation and Coordination on GPUs," in *Intl. Symp. on Computer Architecture (ISCA)*, 2014, pp. 181–192.

[82] M. S. Orr, S. Che, B. M. Beckmann, M. Oskin, S. K. Reinhardt, and D. A. Wood, "Gravel: Fine-Grain GPU-Initiated Network Messages," in *Proc. of the Intl. Conf. for High Performance Computing, Networking, Storage and Analysis (SC)*, 2017, pp. 23:1–23:12.

[83] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell, "A Survey of General-Purpose Computation on Graphics Hardware," in *Computer Graphics Forum*, vol. 26, no. 1, 2007, pp. 80–113.

[84] PaX Team. (2001) Design of ASLR in PaX. [Online]. Available: https://pax.grsecurity.net/docs/aslr.txt

[85] S. Potluri, N. Luehr, and N. Sakharnykh. (2016) Simplifying Multi-GPU Communication with NVSHMEM. [Online]. Available: http://on-demand-gtc.gputechconf.com/gtc-quicklink/7D7mU

[86] S. Reinhardt, J. Larus, and D. Wood, "Tempest and Typhoon: User-Level Shared Memory," in *Intl. Symp. on Computer Architecture (ISCA)*, 1994, pp. 325–336.

[87] D. Rossetti. (2015) GPUDirect Async. [Online]. Available: http://on-demand.gputechconf.com/gtc/2015/presentation/S5412-Davide-Rossetti.pdf

[88] D. Roweth and A. Pittman, "Optimized Global Reduction on QsNetII," in *Symp. on High Performance Interconnects (Hot Interconnects)*, 2005,

pp. 23–28.

[89] Sandia National Laboratories. (2017) The Portals 4.1 Network Programming Interface. [Online]. Available: http://www.cs.sandia.gov/Portals/portals41.pdf

[90] Sandia National Laboratories. (2018) Sandia OpenSHMEM. [Online]. Available: https://github.com/Sandia-OpenSHMEM/SOS

[91] F. Seide, H. Fu, J. Droppo, G. Li, and D. Yu, "1-Bit Stochastic Gradient Descent and Application to Data-Parallel Distributed Training of Speech DNNs," in *Annual Conf. of the Intl. Speech Communication Association (Interspeech)*, 2014, pp. 1058–1062.

[92] G. Shah and C. Bender, "Performance and Experience with LAPI – A New High-Performance Communication Library for the IBM RS/6000 SP," in *Proc. of the First Merged Intl. Parallel Processing Symp. (IPPS) and Symp. on Parallel and Distributed Processing (SPDP)*, 1998, pp. 260–266.

[93] M. Strengert, C. Müller, C. Dachsbacher, and T. Ertl, "CUDASA: Compute Unified Device and Systems Architecture," in *Procs. of the 8th Eurographics Conf. on Parallel Graphics and Visualization (EGPGV)*, 2008, pp. 49–56.

[94] J. A. Stuart and J. D. Owens, "Message Passing on Data-Parallel Architectures," in *Intl. Symp. on Parallel Distributed Processing (IPDPS)*,

2009, pp. 1–12.

[95] TACC. (2015) Stampede Supercomputer User Guide. [Online]. Available: https://portal.tacc.utexas.edu/user-guides/stampede

[96] R. Thakur, R. Rabenseifner, and W. Gropp, "Optimization of Collective Communication Operations in MPICH," *Intl. Journal of High Performance Computing Applications*, vol. 19, no. 1, pp. 49–66, 2005.

[97] TOP500.org. (2018) Highlights - June 2018. [Online]. Available: https://www.top500.org/lists/2018/06/highlights/

[98] K. D. Underwood, J. Coffman, R. Larsen, K. S. Hemmert, B. W. Barrett, R. Brightwell, and M. Levenhagen, "Enabling Flexible Collective Communication Offload with Triggered Operations," in *Symp. on High Performance Interconnects (Hot Interconnects)*, 2011.

[99] M. Valiev, E. Bylaska, N. Govind, K. Kowalski, T. Straatsma, H. V. Dam, D. Wang, J. Nieplocha, E. Apra, T. Windus, and W. de Jong, "NWChem: A Comprehensive and Scalable Open-source Solution for Large Scale Molecular Simulations," *Computer Physics Communications*, vol. 181, no. 9, pp. 1477–1489, Sep 2010.

[100] W. Wen, C. Xu, F. Yan, C. Wu, Y. Wang, Y. Chen, and H. Li, "TernGrad: Ternary Gradients to Reduce Communication in Distributed Deep Learning," in *31st Annual Conf. on Neural Information Processing Systems (NIPS)*, 2017.

[101] J. J. Willcock, T. Hoefler, N. G. Edmonds, and A. Lumsdaine, "AM++: A Generalized Active Message Framework," in *Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT)*, 2010, pp. 401–410.

[102] S. J. E. Wilton and N. P. Jouppi, "CACTI: An Enhanced Cache Access and Cycle Time Model," *IEEE Journal of Solid-State Circuits*, vol. 31, no. 5, pp. 677–688, May 1996.

[103] Y. Zheng, A. Kamil, M. B. Driscoll, H. Shan, and K. Yelick, "UPC++: A PGAS extension for C++," in *Intl. Parallel and Distributed Processing Symp. (IPDPS)*, 2014, pp. 1105–1114.