



GPU Triggered Networking for Intra-Kernel Communications

Michael LeBeane

The University of Texas at Austin

Advanced Micro Devices, Inc.

mlebeane@utexas.edu

Khaled Hamidouche

Advanced Micro Devices, Inc.

Khaled.Hamidouche@amd.com

Brad Benton

Advanced Micro Devices, Inc.

Brad.Benton@amd.com

Mauricio Breternitz*

Instituto Universitario de Lisboa

mbjrz@iscte.pt

Steven K. Reinhardt*

Microsoft Corporation

stever@microsoft.com

Lizy K. John

The University of Texas at Austin

ljohn@ece.utexas.edu

ABSTRACT

GPUs are widespread across clusters of compute nodes due to their attractive performance for data parallel codes. However, communicating between GPUs across the cluster is cumbersome when compared to CPU networking implementations. A number of recent works have enabled GPUs to more naturally access the network, but suffer from performance problems, require hidden CPU helper threads, or restrict communications to kernel boundaries.

In this paper, we propose GPU Triggered Networking, a novel, GPU-centric networking approach which leverages the best of CPUs and GPUs. In this model, CPUs create and stage network messages and GPUs trigger the network interface when data is ready to send. GPU Triggered Networking decouples these two operations, thereby removing the CPU from the critical path. We illustrate how this approach can provide up to 25% speedup compared to standard GPU networking across microbenchmarks, a Jacobi stencil, an important MPI collective operation, and machine-learning workloads.

CCS CONCEPTS

- Computer systems organization → Heterogeneous (hybrid) systems;
- Hardware → Networking hardware;

KEYWORDS

GPUs, RDMA networks, NIC hardware

ACM Reference format:

Michael LeBeane, Khaled Hamidouche, Brad Benton, Mauricio Breternitz, Steven K. Reinhardt, and Lizy K. John. 2017. GPU Triggered Networking for Intra-Kernel Communications. In *Proceedings of SC17, Denver, CO, USA, November 12–17, 2017*, 12 pages.
DOI: 10.1145/3126908.3126950

*The work in this paper was performed while the author was employed by AMD.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SC17, Denver, CO, USA

© 2017 ACM. 978-1-4503-5114-0/17/11...\$15.00

DOI: 10.1145/3126908.3126950

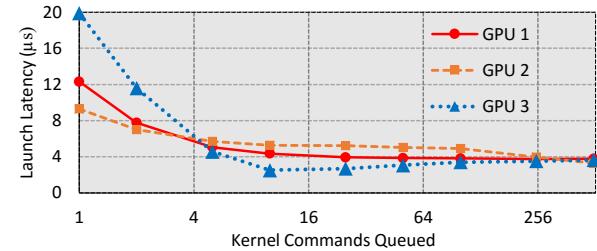


Figure 1: Study of kernel launch latencies on modern GPUs. While there is significant variance depending on how many kernels are presented to the scheduler at once, even the best case takes 3–4 μ s. These high latencies discourage the use of kernel-boundary networking for fine-grained or frequent communication.¹

1 INTRODUCTION

With the impending end of Moore’s Law and Dennard scaling [9], the computing industry has turned to accelerators to continue pushing the performance and power trends of the last 25 years. Chief among proposed accelerator architectures are GPUs, which have already found a comfortable home in both datacenters and high-performance computing (HPC) ecosystems. Over 70 of the top 500 supercomputers [39] and many of the supercomputers on the Green 500 list [38] utilize GPUs to maximize performance per watt on data parallel workloads. Additionally, most major cloud computing providers, such as Amazon EC2 [3], offer GPU-enabled nodes as part of their service portfolio. GPUs are, and will continue to be, an important part of many computing infrastructures.

Despite widespread deployment across clusters both large and small, networks of GPUs are currently programmed in a cumbersome coprocessor style. While many clusters employ peer-to-peer capabilities for direct data movement between NICs and GPUs [25], the control plane is routed through the host CPU. Such data movement operations are only available through CPU runtime and driver calls, which restricts communication to only occur on GPU kernel boundaries [10, 25, 35]. We will refer to this style of kernel-boundary GPU networking as **Host-Driven Networking** (HDN), since the host directs the networking operations at kernel boundaries.

The most recent iteration of HDN technology, GPUDirect Async (GDS) [33], goes so far as to allow the GPU to initiate pre-registered network messages by ringing a doorbell on the NIC. In the GDS

¹Product names omitted to avoid direct cross-vendor comparisons.

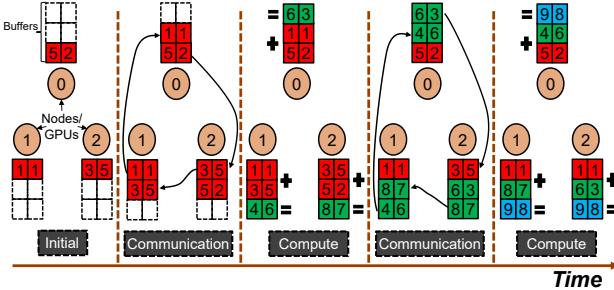


Figure 2: Allreduce algorithm on a cluster of GPUs organized as a simple ring. As the number of participating GPUs increases with a fixed problem size, kernel launch overheads between the communication and computation phases will become more pronounced.

model, the CPU posts a network operation and interleaves network initiation between kernel invocations inside of NVIDIA® CUDA™ [28] streams. The GPU front-end scheduler evaluates the stream and rings a doorbell on the NIC when a kernel has completed. While this is certainly a step in the right direction, GDS still restricts communication to kernel boundaries. On modern GPU architectures, launching and tearing down a kernel context adds significant latency, even when the scheduling logic is performed locally on the GPU using its hardware scheduler.

Figure 1 explores these launch latencies on modern GPUs from multiple vendors and different form factors. Our experiments quantify the overheads associated with the GPUs’ hardware scheduling logic when 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 $3\mu\text{s}$ - $20\mu\text{s}$. Even in the best case, the large dispatch overheads discourage fine-grained or frequent messaging using kernel boundary solutions, and require 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 negate the efforts of network interconnect providers, who have successfully reduced wire latencies to less than 100ns per hop [20].

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 2, which is an important primitive in some distributed, GPU-accelerated machine learning [1] applications. In Allreduce, each GPU needs a piecewise combination of the vector present on every other GPU. Data gets copied from one node to the next node during the communication step which is followed by a user-specified binary operation 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 make GPUs attractive.

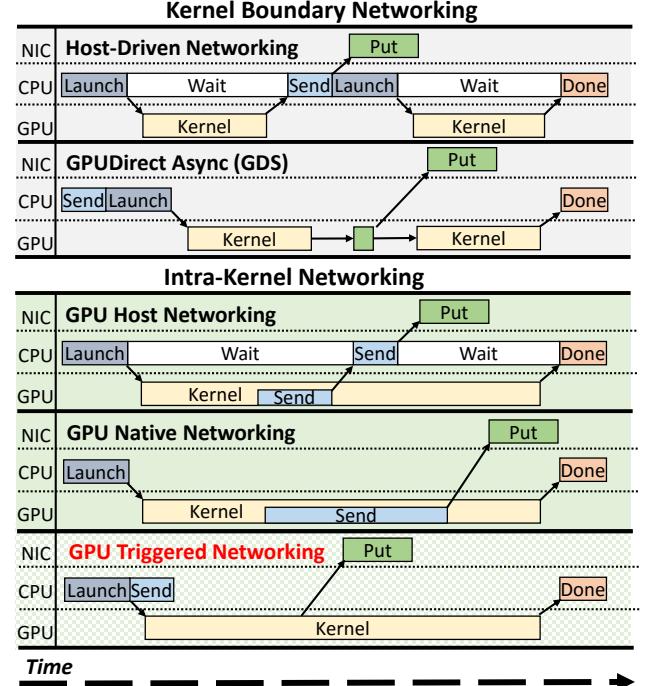


Figure 3: Overview of the control flow of different networking strategies on the GPU. GPU Triggered Networking (GPU-TN) utilizes the CPU to construct a command packet for the GPU to initiate, bypassing an expensive control flow switch on the critical path. GPU-TN supports flexible, intra-kernel networking using triggered operation semantics on the NIC. Note that the time spent is not drawn to scale.

In contrast to kernel boundary networking, some prospective research communication models propose initiating network messages from within a kernel itself. These GPU-centric networking models allow for more autonomy on the GPU, avoid the steep kernel initiation and teardown costs discussed previously, and provide a more natural interface for programmers to express fine-grained communication patterns.

Current intra-kernel networking mechanisms can largely be broken down into two classes depending on their design. The first approach, which we will refer to as **GPU Native Networking**, constructs a networking stack on the GPU itself [8, 22, 23, 30, 31]. 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. While some works show promise with running a network stack on the GPU [22, 23], others illustrate poor performance [31] or suffer correctness issues [8] related to the GPU’s relaxed memory model.

The second GPU-centric approach, which we will refer to as **GPU Host Networking**, defines a lightweight, GPU-optimized interface between the GPU and the CPU [13, 21, 36]. 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

Table 1: Qualitative comparison of discussed GPU networking strategies. The accompanying text describes the overheads.

	GPU Triggered	Intra-Kernel	GPU Overhead	CPU Overhead
Host-Driven Networking (HDN) [10, 25, 35]	No	No	Kernel Boundary	Network Stack
GPU Native Networking [8, 22, 23, 30, 31]	Yes	Yes	Network Stack	NA
GPU Host Networking [13, 21, 26, 36]	No	Yes	CPU/GPU Queues	Service Threads, Network Stack
GPU Direct Async (GDS) [33]	Yes	No	Kernel Boundary, Trigger	Partial Network Stack
GPU Triggered Networking (GPU-TN)	Yes	Yes	Trigger	Partial Network Stack

it off to the NIC. While it is possible to achieve high bandwidth in this approach, it does incur a latency penalty by requiring the CPU to construct messages. Additionally, one or more helper threads are required on the CPU to process messages on the GPU’s behalf. Introducing helper threads ties up CPU resources that could otherwise be used for useful computation.

In this paper, we introduce a new flavor of intra-kernel GPU communication that we call **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 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. An overview of the control flow of all discussed GPU networking frameworks, including GPU-TN, is presented in Figure 3. 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 adaptor.
- **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.

A qualitative comparison of GPU-TN with alternative networking approaches is presented in Table 1.

GPU-TN is inspired by triggered operations [34], 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* [11], which optimizes communication in systems where each thread contributes a small portion of data to the total message.

This paper explores the design and evaluation of GPU-TN. We describe 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. We also describe how GPU-TN’s intra-kernel API offers a high degree of flexibility for the kernel programmer. Finally, we evaluate 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. We show that 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.

2 BACKGROUND

This section provides a brief overview of the technologies underlying this work. Specifically, we provide an introduction to relevant GPU architectural features and high-performance networking concepts.

2.1 GPGPU Overview

General-purpose computing on graphics processing units (GPGPUs) involves offloading structured, data parallel sections of workloads to GPUs. In this work, we will use the OpenCL™ [12] programming terminology for GPGPUs, however, all GPU structures and concepts discussed in the paper also have an equivalent representation in CUDA [28].

2.1.1 GPU Architecture. GPUs are composed of a collection of *compute units* (CUs), which in turn are composed of groups of single instruction, multiple data units. Groups of GPU threads, each individually known as a *work-item*, are scheduled on a compute

unit in a thread bundle called a *work-group*. Work-items executing in the same work-group can efficiently synchronize and communicate with each other using an application controlled, shared storage space called the *Local Data Share*. The subgroup of threads dispatched at the same time on a compute unit is known as a *wavefront*. Threads in a wavefront operate in lockstep; if a work-item in a wavefront branches in a different direction than another work-item, then the wavefront is said to diverge and is executed twice with an execution mask used to ignore the unwanted results. The key to achieving good performance on a GPU is to avoid divergence and provide enough work to hide latencies and achieve good throughput.

2.1.2 GPU 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 and 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 read by the GPU's front-end scheduling unit.

2.2 High-Performance Networking

High-performance networking protocols, such as InfiniBand™ [17], iWARP [19], Omni-Path [20], and RDMA over Converged Ethernet (RoCE) [18], can operate using Remote Direct Memory Access (RDMA) technology to completely avoid the target CPU when performing network operations. RDMA technologies are particularly useful for implementing one-sided communication semantics. In one-sided communications, the programmer or runtime exposes a region of memory to a remote node, which can then be directly accessed using get or put operations, which are analogous to loads and stores on a single node. The CPU at the target node is unaware of the operations occurring on the exposed data; synchronization is separate from data movement in traditional one-sided semantics. This paradigm stands in contrast to the more traditional send/receive, two-sided communication popularized by earlier versions of the Message Passing Interface (MPI) [27], which provides both data movement and synchronization in the same calls. In this paper, GPU-TN implements one-sided communication semantics, as their simplicity makes them a natural fit for GPUs.

3 THE GPU-TN ARCHITECTURE

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

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

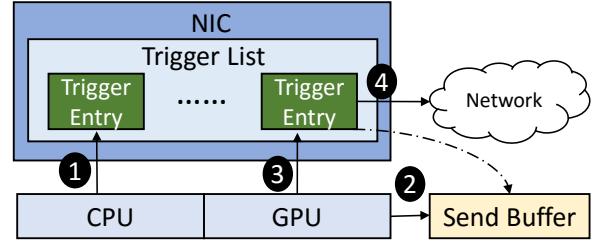


Figure 4: Overview of a GPU triggered operation in GPU-TN. The CPU initializes the network operation, which is triggered by the GPU from within a kernel when the message is ready to be sent.

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.

3.1 Overview

Figure 4 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 ①. The CPU is responsible for creating the network operation using the triggered operations API (see Section 4) 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. We will refer to this address as the *trigger address*. During kernel execution, the GPU will populate the send buffer with data to send to another node ②. After the send buffer is populated, the GPU notifies the NIC that the triggered put operation is ready by performing a posted write operation to the memory-mapped trigger address, supplying the tag of the message that it wants to initiate ③. 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 ④.

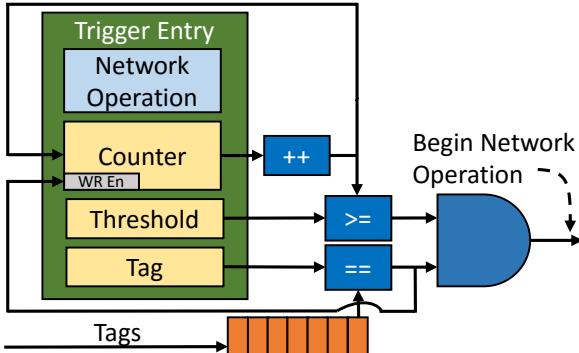


Figure 5: Tag matching behavior of trigger entries. GPU provided tags are matched to a CPU registered trigger entry. The network operation is ready when the counter reaches the threshold.

3.2 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 tag, 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 3.1.

3.3 NIC Hardware Extensions

GPU-TN requires minimal modifications to a standard RDMA network interface to support these new semantics. The simplest implementation would store the trigger list in main memory, and allow a cache on the NIC 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.

If the NIC is implemented using a programmable microprocessor, the logic-level changes required for GPU-TN would be simple to

```

...  

// 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: Pseudocode illustrating the responsibilities of the host CPU in GPU-TN. The host CPU constructs a network packet on behalf of the GPU, but is not involved in the network transfer on the critical path.

add in software. If the NIC is implemented using custom logic, Figure 5 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. 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 [6], 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. Our prototype implementation evaluated in Section 5 requires no more than 16 simultaneous active trigger operations, which allows us to adopt the associative lookup optimization with small hardware overhead.

3.4 GPU-TN and 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 5), it could prove limiting for some more dynamic applications.

For the applications explored in this work, we were 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 NICs trigger address, the GPU could contribute more fields dynamically, such

```

__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
    ...
}

__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
    ...
}

__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
    ...
}

```

(a) Work-item-level networking.
(b) Work-group-level networking.
(c) Kernel-level networking.

Figure 7: GPU kernel pseudocode illustrating how to trigger network transfers through GPU-TN for different granularities.

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 tradeoff performance and flexibility. However, we leave a detailed treatment and analysis of a more dynamic implementation of GPU-TN as future work.

4 PROGRAMMING MODEL

GPU-TN provides a low-level programming interface suitable for runtime library developers to implement highly optimized networking code. In this section, we will describe the host-facing API for registering triggered operations with the NIC. We will also describe a number of sample GPU kernels illustrating the flexibility of the GPU-TN programming model.

4.1 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 shows the essential host-side steps in GPU-TN. First, the network communications runtime performs general network initialization and allocates the send buffer ①. Then, the host code registers a number of operations with the NIC, providing a threshold and unique tag-based identifier for every operation ②. The NIC runtime library allocates a trigger entry for this operation. Next, the memory-mapped triggered address is extracted from the networking runtime so that it can be provided to the GPU ③. This trigger address is then passed as a kernel argument when the kernel is launched, along with one or more tags ④. 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 ⑤.

One key feature of GPU-TN is that steps ④ and ② 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 3.2.

4.2 Kernel API

Intra-kernel networking offers numerous benefits over traditional kernel-boundary communication [8, 21–23, 30, 31, 36]. 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 programmers and runtime developers. In this section, we illustrate how GPU-TN can be supported at multiple granularities. Figure 7 provides example kernels for each granularity using an OpenCL-like pseudocode syntax.

4.2.1 Work-item/Work-group-Level. In Figures 7a and 7b, 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.

4.2.2 Kernel-Level. Figure 7c 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.

4.2.3 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.

4.2.4 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. In GPU-TN, we simply expose an additional global variable for each trigger operation that is set by the NIC on message completion. While this is not shown in our simple examples in Figure 7, the GPU threads can query this location to determine completion status of individual network operations without the complexity of monitoring a network completion queue.

4.2.5 Target-Side Completion. GPU-TN implements a one-sided communication style described in Section 2, which fits very naturally with the hardware capabilities of the GPU [14]. Complex tag matching and deep runtime stacks present in two-sided communication paradigms like MPI introduce software complexity that is difficult to efficiently implement 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 partitioned global address space (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.

4.2.6 GPU-TN and the GPU’s Scoped Memory Model. Modern GPUs operate using a relaxed memory model, where explicit fences and scope specifiers need to be provided by the programmer in order to ensure correct visibility and ordering of memory accesses by different agents in the system [16]. By default, languages like OpenCL only provide visibility within a workgroup. System scope accesses from within a GPU kernel are particularly difficult, and require

Table 2: GPU-TN simulation configuration.

CPU and Memory Configuration	
Type	8 Wide OOO, 4GHz, 8 cores
I,D-Cache	64K, 2-way, 2 cycles
L2-Cache	2MB, 8-way, 4 cycles
L3-Cache	16MB, 16-way, 20 cycles
System Memory	DDR4, 8 Channels, 2133MHz
GPU Configuration	
Type	1 GHz, 24 Compute Units
D-Cache	16kB, 64B line, 16-way, 25 cycles
I-Cache	32kB, 64B line, 8-way, 25 cycles
L2-Cache	768kB, 64B line, 16-way, 150 cycles
Kernel Latencies	1.5 μ s launch / 1.5 μ s teardown
Network Configuration	
Latency	100ns Link, 100ns Switch
Bandwidth	100Gbps
Topology	Star (single switch)

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. However, we believe that this is a temporary constraint, and that future GPUs will implement more system scope operations in the future.

The examples in Figure 7 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 to ensure that the GPU sees updates from the NIC itself.

5 EVALUATION

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

5.1 Experimental Setup

To evaluate GPU-TN, we use the open-source gem5 simulator [5] including the AMD public GPU compute model [4]. Table 2 shows the specific configuration for the major components of our infrastructure. We configure our system to resemble a compute node containing a CPU, GPU, and NIC. Our GPU model is configured as part of a high-performance SoC, where the CPU and GPU share system memory and are coherent. This avoids memory copies between GPU and CPU address spaces and minimizes offload times from the CPU to the GPU. While we have selected this configuration for our analysis, GPU-TN can still be applied in a more traditional discrete GPU architecture, provided they support the advanced system synchronization operations discussed in Section 4.2.6.

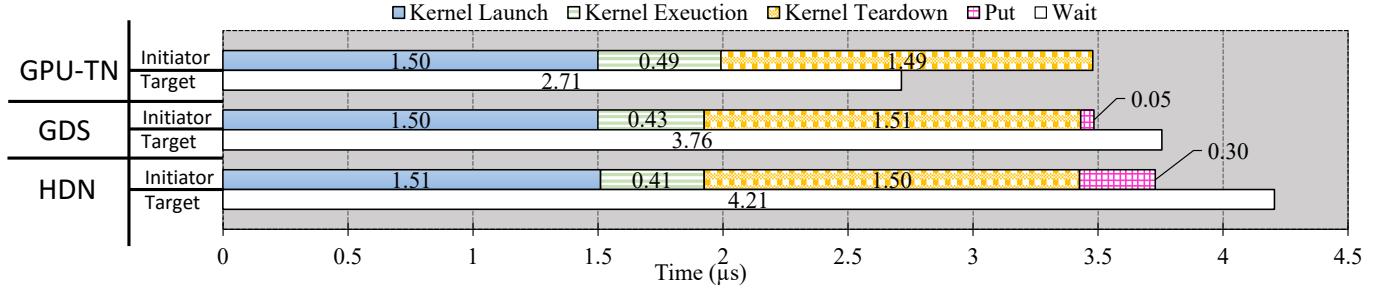


Figure 8: GPU-TN vs HDN vs GDS latency decomposition from a small microbenchmark. GPU-TN achieves approximately 35% performance improvement over an HDN approach and 25% improvement over GDS.

Our infrastructure simulates multi-node configurations with a simple switch and wire delay model. The NIC model implements the Portals 4 [34] network programming specification with custom GPU-TN functions implemented using an API similar to existing Portals 4 triggered operations.

The removal of GPU kernel boundary latencies to send network messages is a key motivating factor behind GPU-TN. We calibrated our simulation infrastructure to model some of the more optimistic numbers derived from our experiments in Figure 1. The performance results presented for GPU-TN are based on $3\mu\text{s}$ 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 indicates that the performance uplift of GPU-TN could be even higher than the results reported in this section.

In our experiments, we compare four different networking strategies that we will refer to as CPU, HDN, GDS, and GPU-TN. These configurations are defined as follows:

- **CPU:** All computation and communication is done 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.
- **GDS:** The GDS baseline approximates the behavior of GPUDirect Async [33] kernel boundary communication in our 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.

5.1.1 Comparison to GPU Host/Native Networking. For our results, we do not explicitly compare against GPU Host/Native Networking approaches as defined in Section 1. This is largely due to the fact that we are unaware of any open source implementations of these approaches that are compatible with our simulation environment, and implementing our own approaches from scratch is a considerable effort. However, in this section, we provide a brief qualitative discussion of how we expect GPU Host/Native Networking approaches to compare with GPU-TN.

GPU Native Networking runs the entire networking stack locally on the GPU. We expect 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.

GPU Host Networking uses dedicated polling threads on the host to service messages on behalf of the GPU. The key advantage of GPU-TN over these approaches is that it can provide the same 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.

5.2 Latency Analysis

In this section, we analyze 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 8 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 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.

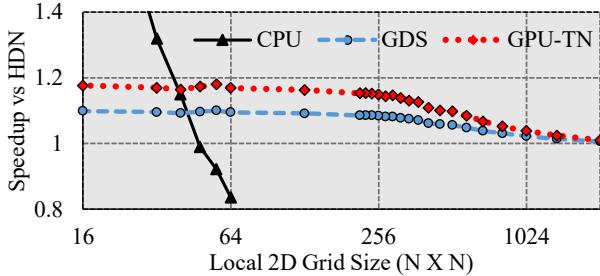


Figure 9: GPU-TN performance on a single iteration of a 2D Jacobi Relaxation computation over different $N \times N$ grid sizes. Strong scaling Jacobi would move further to the left on the graph. GPU-TN offers up to 10% improvement vs GDS and 20% improvement vs HDN over a range of medium-sized grids.

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, we do wish to note 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 IO bus.

When comparing the GDS implementation against GPU-TN, we see two distinct differences. 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 phenomena 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 uplift over GDS, and approximately 35% improvement over HDN.

5.3 2D Jacobi Relaxation

This section evaluates the performance of GPU-TN over a 2D Jacobi relaxation problem [24] with various input sizes. Jacobi relaxation is a method used to determine solutions of a diagonally dominant system of linear equations. From a computational perspective, it takes the form of an iterative stencil. During each iteration, a series of operations are performed on a local data set, followed by a halo exchange of neighboring data. 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. This particular implementation of Jacobi does not exploit overlap.

We implemented the Jacobi relaxation over our 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

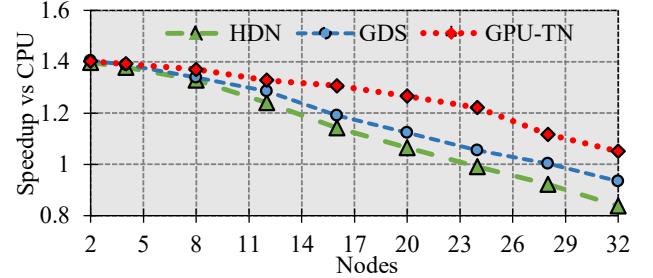


Figure 10: GPU-TN strong scaling performance evaluation on an 8MB MPI Allreduce collective operation. GPU-TN scales better than HDN/GDS as the number of nodes participating in the collective operation increases.

MPI send/receives after every round. 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 time, and GPU-TN uses a single kernel for the entire duration of the program.

Figure 9 illustrates the results of the Jacobi relaxation on our 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, we see that GPU-TN achieves approximately 10% improvement over GDS, and approximately 20% improvement over HDN on medium problem sizes. CPU results are included in the figure to ensure that the range of problem sizes GPU-TN offers benefits on do not fall outside what is useful to offload onto a GPU.

5.4 Collective Operations

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 [7, 29]. In this section, we use GPU-TN to implement the Allreduce collective operation in MPI, and demonstrate GPU-TN’s ability to accelerate the performance of workloads written in a popular machine learning framework, Microsoft’s Cognitive Toolkit [1].

5.4.1 Allreduce. Allreduce is an MPI collective operation that combines the contents of all participating nodes’ buffers using some arithmetic operation, as illustrated in Figure 2. At the end of an Allreduce operation, all participating nodes contain contributions from every other node. There are a number of topology specific implementations that optimize the number of messages and operations performed on all the nodes; for our case study, we will use a simple ring communications pattern.

We implement GPU-TN in the libNBC [15] 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

Table 3: CNTK workload description. The %Blocked heading refers to total time spent blocked on an Allreduce operation, and Reductions refers to the total number of reduction calls.

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

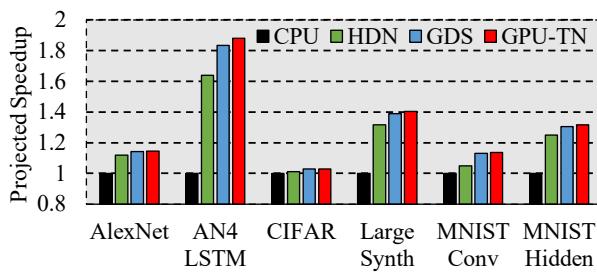


Figure 11: GPU-TN performance across six deep learning workloads on a cluster of 8 nodes. GPU-TN performs up to 20% better than HDN and 5% better than GDS on select machine learning workloads.

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 [40].

We implement the Allreduce algorithm on CPU, GDS, GPU-TN, and HDN systems. The implementations of CPU, GDS, and HDN are similar to what was described in the Jacobi benchmark evaluation. 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. Our implementation triggers the network operation at the granularity of a work-group; this allows for easy software pipelining of the computation and network transfer.

Figure 10 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 to provide speedup into 32 nodes and beyond.

5.4.2 Deep Learning. Deep 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 operation has been shown to be a significant bottleneck in deep learning workloads, especially those operating in the synchronous training mode. For our studies, we have selected six machine learning workloads from a variety of application domains. A brief description of each is presented in Table 3.

Figure 11 shows how GPU-TN can be used to accelerate training of neural networks on Microsoft's Cognitive Toolkit [1] deep learning platform on a cluster of 8 nodes. To perform our study, we ran a variety of deep learning workloads on the Stampede supercomputer [37] and measured the frequency, time, and data size of the various Allreduce calls. Using these numbers, we were able to use results from our simulator to project the application-level speedup from applying GPU-TN to these workloads. Since these deep learning workloads were operating in the synchronous training mode, there are no computation/communication overlap effects to worry about when performing the projections.

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. GPU-TN provides the most benefit in scenarios where there are a large number of small-to-medium-sized collective operations.

6 RELATED WORK

This section describes related work involving optimizing communication for GPUs and other accelerators. We attempt to classify the prior work using the taxonomy introduced in Section 1.

Host-Driven Networking: A few works attempt to optimize GPU networking while retaining the classic GPU-as-a-coprocessor style of programming interface. Zippy [10] and Compute Unified Device and Systems Architecture (CUDASA) [35] 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. NCCL [29] is a communication library that performs efficient collective operations across multiple GPUs in one or many nodes.

The most popular commercial GPU networking approach is an HDN solution called GPUDirect RDMA [25], which allows high-performance RDMA NICs to directly access GPU local memory through a PCIe Base Address Register (BAR) window without intermediate data copies in system memory. CUDA-aware OpenSHMEM [14] takes advantage of GPUDirect RDMA features to optimize data movement for one sided communications. The most recent version of this technology, GPUDirect Async [33], is used

as a comparison point to this work and is discussed extensively throughout the paper.

GPU Host Networking: Some projects attempt to support GPU networking through helper threads on the host CPU. FLAT [26] allows for the automatic generation of CPU MPI codes from GPU kernels using custom compiler extensions. Distributed Computing for GPU Networks (DCGN) [36] 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 [21] provides a socket-based abstraction for the GPU, and also uses CPU helper threads to perform the actual communication. dCUDA [13] implements a GPU networking programming model that attempts to hide long latency GPU network events across the cluster. These approaches differ from GPU-TN in that one or more dedicated, critical path service thread(s) are required to manage communication on the CPU.

GPU Native Networking: Some recent work implements GPU centric networking while trying to avoid CPU helper threads. GPU Global Address Space (GGAS) [30] explores 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.* explore implementing InfiniBand entirely on a GPU runtime, with mixed results [31]. However, additional work by the same research group illustrates much more favorable performance [22, 23]. GPUrdma [8] also implements InfiniBand directly on the GPU, although limitations of current GPU hardware can cause correctness problems under high load. NVSHMEM provides an OpenSHMEM-like interface to perform one-sided communication from within a kernel, but is currently limited to a single node [32]. Concurrently with our work, Agostini et. al [2] described several implementations of GPUDirect Async, one of which offers similar intra-kernel networking semantics to GPUrdma and GPU-TN.

Triggered Operations: Our work makes use of triggered network operations to improve the networking performance of GPUs. Triggered operations were introduced in the Portals 4 network programming API [34] as a way to build efficient sequences of operations that can be progressed by the NIC. In Portals 4, the host CPU provides the NIC with a network operation to perform, and a threshold indicating how many events must occur before the operation is initiated. A triggered operation is initiated when a lightweight event counter reaches the specified threshold. Triggered operations have been shown to be effective for implementing collective operations [40], which generally involve a number of consecutive steps that need to occur in response to various network events.

7 CONCLUSION

First-order GPU networking support is essential for multi-node GPU applications. As GPUs become more tightly integrated into a node’s compute ecosystem, strong networking paradigms will likely become essential for good performance.

In this paper, we 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. We illustrate that 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. We look forward to exploring more applications that can benefit from the high-performance, intra-kernel networking that can be provided by GPU-TN acceleration in future work.

ACKNOWLEDGEMENTS AND ATTRIBUTIONS

We would like to thank the anonymous reviewers for their detailed feedback, which undoubtedly improved the quality of this work.

The authors acknowledge the Texas Advanced Computing Center (TACC) at The University of Texas at Austin and Advanced Micro Devices Inc. for providing resources that have contributed to the research results reported within this paper. Mauricio Breternitz is partially funded by Marie Curie IRIS (ref. 610986, FP7-PEOPLE-2013-IAPP). Lizy K. John is partially funded by National Science Foundation grant CCF-1337393. Any opinions, findings, conclusions, or recommendations do not necessarily reflect the views of these funding agencies.

AMD, the AMD Arrow logo, and combinations thereof are trademarks of Advanced Micro Devices, Inc. OpenCL is a trademark of Apple Inc. used by permission by Khronos. Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies.

REFERENCES

- [1] Amit Agarwal, Eldar Akchurin, Chris Basoglu, Guoguo Chen, Scott Cyphers, Jasha Droppo, Adam Eversole, Brian Guenter, Mark Hillebrand, T. Ryan Hoen, Xuedong Huang, Zhiheng Huang, Vladimir Ivanov, Alexey Kamenev, Philipp Kranen, Oleksii Kuchaiev, Wolfgang Manousek, Ayner May, Bhaskar Mitra, Olivier Nano, Gaizka Navarro, Alexey Orlov, Hari Parthasarathi, Baolin Peng, Marko Radmilac, Alexey Reznichenko, Frank Seide, Michael L. Seltzer, Malcolm Slaney, Andreas Stolcke, Huaming Wang, Yongqiang Wang, Kaisheng Yao, Dong Yu, Yu Zhang, and Geoffrey Zweig. 2014. *An Introduction to Computational Networks and the Computational Network Toolkit*. Technical Report. Microsoft. <https://www.microsoft.com/en-us/research/wp-content/uploads/2014/08/CNTKBook-20160217.pdf>
- [2] Elena Agostini, Davida Rossetti, and Sreeram Poturi. 2017. Offloading communication control logic in GPU accelerated applications. In *Intl. Symp. on Cluster, Cloud and Grid Computing (CCGrid)*. DOI : <https://doi.org/10.1109/CCGRID.2017.29>
- [3] Amazon. 2016. Amazon EC2 Cloud Computing. <https://aws.amazon.com/ec2>
- [4] AMD. 2015. The AMD gem5 APU Simulator: Modeling Heterogeneous Systems in gem5. http://gem5.org/GPU_Models
- [5] Nathan Binkert, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, David A. Wood, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, and Tushar Krishna. 2011. The gem5 simulator. *ACM SIGARCH Computer Architecture News* 39, 2 (2011), 1. DOI : <https://doi.org/10.1145/2024716.2024718>

- [6] Bull. 2017. BXI: Bull eXascale Interconnect. <https://atos.net/en/products/high-performance-computing-hpc/bxi-bull-exascale-interconnect>
- [7] Ching-Hsiang Chu, Khaled Hamidouche, Akshay Venkatesh, Ammar Ahmad Awan, and Dhabaleswar K. Panda. 2016. CUDA Kernel Based Collective Reduction Operations on Large-scale GPU Clusters. In *Intl. Symp. on Cluster, Cloud and Grid Computing (CCGrid)*. DOI : <https://doi.org/10.1109/CCGrid.2016.111>
- [8] Feras Daoud, Amir Watad, and Mark Silberstein. 2016. GPUrDMA: GPU-side Library for High Performance Networking from GPU Kernels. In *Intl. Workshop on Runtime and Operating Systems for Supercomputers (ROSS)*. 6:1–6:8. DOI : <https://doi.org/10.1145/2931088.2931091>
- [9] Hadi Esmaeilzadeh, Emily Blehm, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. 2011. Dark silicon and the end of multicore scaling. In *Intl. Symp. on Computer Architecture (ISCA)*. 365–376. DOI : <https://doi.org/10.1145/2000064.2000108>
- [10] Zhe Fan, Feng Qiu, and Arie E. Kaufman. 2008. Zippy: A Framework for Computation and Visualization on a GPU Cluster. *Computer Graphics Forum* 27, 2 (2008), 341–350. DOI : <https://doi.org/10.1111/j.1467-8659.2008.01131.x>
- [11] Ryan E Grant, Anthony Skjellum, and V Purushotham. 2015. Lightweight threading with MPI using Persistent Communications Semantics. In *Workshop on Exascale MPI (ExaMPI)*.
- [12] Khronos Group. 2017. OpenCL. <https://www.khronos.org/opencl/>
- [13] Tobias Gysi, Jeremia Bär, and Torsten Hoefler. 2016. dCUDA: Hardware Supported Overlap of Computation and Communication. In *Intl. Conf. for High Performance Computing, Networking, Storage and Analysis (SC) (SC '16)*. Article 52, 12 pages. DOI : <https://doi.org/10.1109/sc.2016.51>
- [14] Khaled Hamidouche, Akshay Venkatesh, Ammar Ahmad Awan, Hari Subramoni, Ching-Hsiang Chu, and Dhabaleswar K. Panda. 2016. CUDA-Aware OpenSHMEM: Extensions and Designs for High Performance OpenSHMEM on GPU Clusters. *Parallel Comput.* 58 (2016), 27–36. DOI : <https://doi.org/10.1016/j.parco.2016.05.003>
- [15] Torsten Hoefler and Andrew Lumsdaine. 2006. *Design, Implementation, and Usage of LibNBC*. Technical Report. Indiana University Bloomington. <http://www.cs.indiana.edu/cgi-bin/techreports/TRNNN.cgi?trnum=TR637>
- [16] Derek R. Hower, Blake A. Hechtman, Bradford M. Beckmann, Benedict R. Gaster, Mark D. Hill, Steven K. Reinhardt, and David A. Wood. 2014. Heterogeneous-race-free Memory Models. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.
- [17] InfiniBand Trade Association. 2000. InfiniBand Architecture Specification: Release 1.0.2. http://www.infinibandta.org/content/pages.php?pg=technology_download
- [18] InfiniBand Trade Association. 2014. RDMA over Converged Ethernet v2. <https://cwg.infinibandta.org/document/dl/7781>
- [19] Intel. 2010. Internet Wide Area RDMA Protocol (iWARP). <http://www.intel.com/content/dam/doc/technology-brief/iwarp-brief.pdf>
- [20] Intel. 2015. Omni-Path Fabric 100 Series. <https://fabricbuilders.intel.com/>
- [21] Sangman Kim, Seonggu Huh, Yige Hu, Xinya Zhang, Emmett Witchel, Amir Wated, and Mark Silberstein. 2014. GPUnet: Networking Abstractions for GPU Programs. In *USENIX Conf. on Operating Systems Design and Implementation (OSDI)*. 201–216. DOI : <https://doi.org/10.1145/2963098>
- [22] Benjamin Klenk, Lena Oden, and Holger Froning. 2014. Analyzing Put/Get APIs for Thread-Collaborative Processors. In *Intl. Conf. on Parallel Processing (ICPP)* Workshops. DOI : <https://doi.org/10.1109/ICPPW.2014.61>
- [23] Benjamin Klenk, Lena Oden, and Holger Froning. 2015. 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)*. DOI : <https://doi.org/10.1109/ISPASS.2015.7095817>
- [24] Jim Lambers. 2010. Jacobi Methods. <http://web.stanford.edu/class/cme335/lecture7.pdf>
- [25] Mellanox. 2017. Mellanox OFED GPUDirect RDMA. http://www.mellanox.com/page/products_dyn?product_family=116
- [26] Takefumi Miyoshi, Hidetsugu Irie, Keigo Shima, Hiroki Honda, Masaaki Kondo, and Tsutomu Yoshihaga. 2012. FLAT: A GPU Programming Framework to Provide Embedded MPI. In *Workshop on General Purpose Processing with Graphics Processing Units (GPGPU)*. 20–29. DOI : <https://doi.org/10.1145/2159430.2159433>
- [27] MPI Forum. 2012. MPI: A Message-Passing Interface Standard. Ver. 3. www mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf
- [28] Nvidia. 2016. CUDA Toolkit 8.0. <https://developer.nvidia.com/cuda-toolkit>
- [29] Nvidia. 2016. Fast Multi-GPU collectives with NCCL. <https://devblogs.nvidia.com/parallelforall/fast-multi-gpu-collectives-nccl/>
- [30] Lena Oden and Holger Froning. 2013. GGAS: Global GPU address spaces for efficient communication in heterogeneous clusters. In *Intl. Conf. on Cluster Computing (CLUSTER)*. 1–8. DOI : <https://doi.org/10.1109/cluster.2013.6702638>
- [31] Lena Oden, Holger Froning, and Franz-Joseph Pfreundt. 2014. Infiniband-Verbs on GPU: A Case Study of Controlling an Infiniband Network Device from the GPU. In *Intl. Conf. on Parallel Distributed Processing Symposium Workshops (IPDPSW)*. 976–983. DOI : <https://doi.org/10.1109/ipdpsw.2014.111>
- [32] Sreram Potluri, Nathan Luehr, and Nikolay Sakharnykh. 2016. Simplifying Multi-GPU Communication with NVSHMEM. <http://on-demand.gtc.gputechconf.com/gtc-quicklink/7D7mU>
- [33] Davide Rossetti. 2015. GPUDirect Async. <http://on-demand.gputechconf.com/gtc/2015/presentation/S5412-Davide-Rossetti.pdf>
- [34] Sandia National Laboratories. 2014. The Portals 4.0.2 Network Programming Interface. <http://www.cs.sandia.gov/Portals/portals402.pdf>
- [35] Magnus Strengert, Christoph Müller, Carsten Dachsbaier, and Thomas Ertl. 2008. CUDASA: Compute Unified Device and Systems Architecture. In *Eurographics Conf. on Parallel Graphics and Visualization (EGPGV)*. DOI : <https://doi.org/10.2312/EGPGV/EGPGV08/049-056>
- [36] Jeff A. Stuart and John D. Owens. 2009. Message passing on data-parallel architectures. In *Intl. Symp. on Parallel Distributed Processing (IPDPS)*. 1–12. DOI : <https://doi.org/10.1109/ipdps.2009.5161065>
- [37] TACC. 2015. Stampede Supercomputer User Guide. <https://portal.tacc.utexas.edu/user-guides/stampede>
- [38] TOP500.org. 2016. Green 500. <http://www.top500.org/green500>
- [39] TOP500.org. 2017. Highlights - June 2017. <https://www.top500.org/lists/2017/06/highlights/>
- [40] Keith D. Underwood, Jerrie Coffman, Roy Larsen, K. Scott Hemmert, Brian W. Barrett, Ron Brightwell, and Michael Levenhagen. 2011. Enabling Flexible Collective Communication Offload with Triggered Operations. In *Symp. on High Performance Interconnects (Hot Interconnects)*. DOI : <https://doi.org/10.1109/HOTI.2011.15>