GPU Initiated OpenSHMEM: Correct and Efficient Intra-Kernel Networking for dGPUs

Khaled Hamidouche
Advanced Micro Devices, Inc.
Khaled.Hamidouche@amd.com

Michael LeBeane
Advanced Micro Devices, Inc.
Michael.Lebeane@amd.com

Abstract

Current state-of-the-art in GPU networking utilizes a host-centric, kernel-boundary communication model that reduces performance and increases code complexity. To address these concerns, recent works have explored performing network operations from within a GPU kernel itself. However, these approaches typically involve the CPU in the critical path, which leads to high latency and inefficient utilization of network and/or GPU resources.

In this work, we introduce GPU Initiated OpenSHMEM (GIO), a new intra-kernel PGAS programming model and runtime that enables GPUs to communicate directly with a NIC without the intervention of the CPU. We accomplish this by exploring the GPU's coarse-grained memory model and correcting semantic mismatches when GPUs wish to directly interact with the network. GIO also reduces latency by relying on a novel template-based design to minimize the overhead of initiating a network operation. We illustrate that for structured applications like a Jacobi 2D stencil, GIO can improve application performance by up to 40% compared to traditional kernel-boundary networking. Furthermore, we demonstrate that on irregular applications like Sparse Triangular Solve (SpTS), GIO provides up to 44% improvement compared to existing intra-kernel networking schemes.

CCS Concepts • **Computer systems organization** → **Heterogeneous (hybrid) systems**; • **Networks** → **Programming interface**; • **Computing methodologies** → *Distributed programming languages*;

Keywords GPUs, Distributed programming models, RDMA networks

PPoPP '20, February 22–26, 2020, San Diego, CA, USA
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-6818-6/20/02...$15.00
https://doi.org/10.1145/3332466.3374544

1 Introduction

GPU-enabled clusters provide high levels of performance and power efficiency for many classes of data-parallel workloads [24]. High Performance Computing (HPC) in particular leverages the unique performance and power profile of modern GPUs to accelerate over 125 of the top 500 supercomputers [38].

HPC systems that employ GPUs typically do so across many compute nodes and use high-performance network adapters to communicate between them. Both the data path and pieces of the control path of remote GPU networking operations have been optimized using peer-to-peer data transfers from a GPU's discrete memory to the NIC [19] and using direct initiation of network operations by the GPU's front end [1, 35], respectively.

Unfortunately, traditional multi-node GPU systems restrict communications to kernel boundaries, which forces the programmer to think about communication separately from computation instead of embedding network runtime calls directly within the kernel itself. This restriction can lead to an increase in algorithm complexity and a decrease in programmer productivity. Additionally, the performance of inter-kernel networking strategies is often poor due to the high cost of starting and ending a GPU kernel. GPU kernel launch overheads have been shown to take upwards of 20μs [15]; this is an order of magnitude greater than modern network latencies of approximately 0.7μs [21].

Recently, researchers have explored methods to overcome the high overheads of ending a kernel by initiating network communication directly from the kernel itself [7, 8, 12–16, 28, 29, 31, 33, 37]. These and other research projects attempt to provide a Message Passing Interface (MPI) [22] or OpenSHMEM [6] programming model that is directly callable from the GPU shader code. Similarly, other works like NVSHMEM [34] have investigated porting the OpenSHMEM programming model to GPUs. However, they currently focus solely on a single node and do not investigate the interaction between GPUs and NICs.

While prior investigations propose different approaches that optimize various facets of GPU networking, they share two common drawbacks:

- **High Latency**: Initiating a network operation involves two main steps: 1) the creation and posting of the network command packet and 2) the ringing of the NIC's doorbell. Both of these steps are sequential memory operations which are not efficiently mappable to the GPU's Single Instruction, Multiple Thread (SIMT) execution style. This limitation is especially true for the creation of the network packet, which can increase latency by an order of magnitude over the latency of the network itself [7, 28].

- **Data Visibility and Ordering Issues**: GPUs use a relaxed memory consistency model [2], where data is visible and ordered with respect to agents external to the GPU only at kernel boundaries. As part of a kernel's teardown/launch procedure, the driver and GPU firmware issue fences and cache invalidations so that GPU-produced data can be accessed by external agents. To overcome this limitation for intra-kernel networking, most prior work relies on the CPU to initiate coherence actions to ensure correct data [37]. Some works do not use the CPU in this manner, but report intermittent data validation errors due to violating the memory consistency model of the GPU [26, 33]. The need for CPU intervention on the critical path of communication mitigates much of the performance improvements derived from enabling the GPU to communicate directly with the NIC.

In this work, we propose GPU Initiated OpenSHMEM (GIO), a GPU-centric PGAS programming model that elevates the GPU to a first class citizen in distributed systems. We investigate the root cause of a GPU's kernel-boundary memory visibility constraints and its mismatch with the semantics required for intra-kernel networking. We then propose practical system designs and techniques to overcome these limitations. GIO also proposes a novel runtime design and techniques for low-latency, GPU-initiated communication. The proposed GIO runtime enables tight integration of the GPU memory system with the NIC. To alleviate the latency issues of existing solutions [7, 28], our runtime introduces a new network packet templating approach to reduce the largely serial overhead of populating the network command packets from the GPU.

While some existing works, such as NVSHMEM [33, 34], proposed GPU support for OpenSHMEM on a single node, we believe removing the GPU-NIC interaction oversimplifies the challenges as GPU-to-GPU communications (via PCIe® or a proprietary GPU-centric network) do not have the same data ordering and visibility issues as GPU-to-NIC interactions. Techniques such as those provided by NVSHMEM can directly access the memory of other GPUs through a simple sequence of load and store operations with appropriate memory fences. Providing data to a NIC is significantly more challenging, as networks employ complex command queues and, as previously mentioned, ordering and visibility operations may not be available from within a GPU kernel for non-GPU devices. This paper investigates this challenging GPU-to-NIC interaction. To the extent of our knowledge, this is the first paper presenting a runtime designs that truly removes the CPU from the critical path and brings the GPU as first class-citizen without data ordering or visibility issues.

This paper presents the following contributions:

- Analyzing the GPU's coarse-grained memory consistency model and its mismatch with intra-kernel GPU networking requirements.
- Proposing system- and runtime-level designs to overcome such memory consistency limitations.
- Designing GPU initiated networking using a template-based approach to minimize message latency.
- Co-designing application kernels to demonstrate the impact and benefits of the GIO programming model and runtime.

2 Background

This section describes basic GPU and networking terminology critical to the understanding of GIO. Readers familiar with these concepts are encouraged to skip to Section 3.

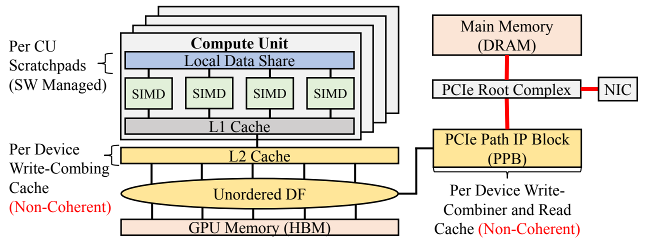


Figure 1. AMD's Graphics Core Next (GCN [3]) GPU architecture with a focus on memory and caches.

2.1 GPU Compute Architecture

Figure 1 illustrates the relevant components of a compute optimized GPU. We use AMD specific terminology for the purposes of this paper, but most concepts are directly applicable to Nvidia.

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 is connected to a private L1 cache and shared L2 cache, which are maintained by explicit cache management instructions. Groups of work-items are dispatched on the CUs in bundles known as wavefronts. These wavefronts are further bundled into work-groups, which are guaranteed to execute on the same CU and can therefore make use of fast, per-CU scratch-pad memory called the Local Data Share (LDS). More specific details concerning the GPU's memory subsystem and consistency model will be discussed in Section 4.1.

GPUs are programmed by writing code known as kernels. For the purposes of this paper, we will be presenting code written in the Heterogeneous-compute Interface for Portability (HIP) [5] language, which has a similar syntax to CUDA [25].

```
__host__ void
hostInit()
{
    // ❶ Initialize GIO Runtime
    gio_shmem_handle_t* gio_shmem_handle;
    gio_shmem_init(&gio_shmem_handle);

    // ❷ Allocate symmetric heap memory
    int size = sizeof(char) * ELEMENTS;
    char* src = gio_shmem_malloc(size);
    char* dst = gio_shmem_malloc(size);

    // ❸ Initiator/target launches kernel
    pe = gio_shmem_my_pe(gio_shmem_handle);
    if (pe == INITIATOR) {
        hipLaunchKernel(Ping, GRID_SZ,
            GRID_SZ / WG_SZ, 0, 0,
            gio_shmem_handle, src, dst);
    } else {
        // Launch pong kernel (not shown)
    }
}
```

(a) Initialization and host code.

```
__device__ void
devicePing(gio_shmem_handle_t *gio_shmem_handle
           char* src, char* dst)
{
    // ❹ Extract context from global handle
    __shared__ gio_shmem_ctx_t gio_shmem_ctx;
    gio_shmem_ctx_create(gio_shmem_handle,
        &gio_shmem_ctx);

    // ❺ Each WG pings target
    gio_shmem_put_nbi(gio_shmem_ctx,
        dst[hipBlockIdx_x],
        src[hipBlockIdx_x],
        sizeof(char), TARGET);

    // ❻ Wait on the network completion
    gio_shmem_quiet(gio_shmem_ctx);

    // ❼ Each WG waits for pong target
    gio_shmem_wait_until(
        dst[hipBlockIdx_x], 1);
    gio_shmem_ctx_destroy(gio_shmem_ctx);
}
```

(b) GPU ping to remote GPU (pong not shown).

Figure 2. GIO remote ping/pong example on the host and the GPU.

2.2 RDMA and OpenSHMEM

Remote Direct Memory Access (RDMA) technology bypasses the target CPU when performing network operations and is implemented in many high-performance networking protocols [11, 36]. Using InfiniBand terminology, each end-point interacts with the NIC using command queues. The Send-Queue (SQ) and Receive-Queue (RQ) are where users post command packets for the NIC to execute, and the Completion-Queue (CQ) is where the NIC posts the status of a completed operation. These structures are often collectively referred to as a Queue-Pair (QP).

RDMA is often used to implement one-sided communication semantics, such as those provided by OpenSHMEM [6]. OpenSHMEM is a Partitioned Global Address Space (PGAS) library specification that defines many one-sided operations, such as remote *Puts*() and *Gets*(), as well as synchronization primitives and collectives. The key data structure used in OpenSHMEM is called a symmetric heap. The symmetric heap is a memory pool where each memory allocation/deallocation call is a collective operation across all nodes, such that each node has the exact same variables at the same offset in its local heap. Therefore, each node can access another node's heap by using pointers to variables allocated on the symmetric heap. In this paper, we have designed GIO according to the semantics of the OpenSHMEM network programming standard.

3 GIO Programming Model

GIO implements an OpenSHMEM-based API that is exposed to the GPU programmer through a device side library. Each GIO API call (e.g., *Put()*, *Get()*, *Collective()*, etc.) takes the same arguments as a standard OpenSHMEM 1.4 [6] CPU implementation (e.g., source, destination, length, etc.)

The HIP programming model is used to implement GIO and provides a flexible, multi-level granularity corresponding to different collections of threads on the GPU; operations can be at grid-level, work-group level or even thread-level. For GIO, our design implements each operation as a work-group collective, which means that 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 (grid-level), and GIO would need to synchronize across work-groups, which is expensive and limits the ability for work-groups to overlap. However, if the application aggregates the message (sending from a contiguous buffer), one work-group can send the whole data as a single operation. Any smaller (thread-level), and the message size would most likely be too small to saturate the network link when streaming large messages.

Figure 2 illustrates a simple ping-pong benchmark between two GPUs. The pong step is omitted since it is similar to ping and offers no additional information regarding GIO's API. GIO's API is divided into both host and GPU components, which we will discuss in turn.

Figure 2a illustrates the host-side API, which is responsible for initializing the runtime and managing the memory and resources. First, the host initializes the runtime and creates a handle for the GPU ❶. This initialization step allocates a number of network resources (QPs and CQs) and establishes the connections. It also initiates the OpenSHMEM symmetric heap and saves this information in the global handle. It is important to note that the symmetric heap is allocated on GPU memory, similarly to what was done in the prior work by Hamidouche et al. [9]. More details on the design and information extracted and stored in the handle will be discussed in Section 5. Next, the host allocates a network accessible buffer on the symmetric heap allocated from GPU memory ❷. Finally, a GPU kernel is launched with the handle and the allocated buffer ❸.

Figure 2b illustrates the GPU-side API for the ping component of the ping-pong benchmark, which executes the whole application algorithm including both the computation and communication phases. The GPU first calls an initialization function with the host-provided handle ❹. This API creates a private communication context for each work-group. The context is allocated in LDS memory, since our API is implemented at a work-group level. This avoids accesses to global memory or increased register pressure to hold network state. The next two steps perform standard one-sided network calls to initiate a remote *Put*() on the target ❺ and waits for the completion of the network operation ❻. Finally the target waits for the ping that was just sent by the initiator ❼. Each work-group performs a separate ping operation on an independent buffer indexed by work-group ID.

4 Intra-Kernel Networking Data Consistency Issues

This section briefly describes the GPU's memory consistency model and the main hardware components that govern the GPU's interactions with external devices, such as a NIC. We then discuss the mismatch between the GPU's memory consistency model and the GPU-initiated networking vision. The discussion will largely be focused on the AMD GPU architecture described in Section 2. However, recent works have noted similar observations on Nvidia GPUs [1, 27].

4.1 GPU Memory Model and Relevant Hardware Components

The GPU operates under a relaxed memory model [10] that may be unfamiliar to many programmers accustomed to the stronger guarantees offered by most modern CPUs. Memory accesses on GPUs correspond to a scope, where the scope defines the level of visibility and ordering requirements of the GPU data (e.g., work-group, device, and system). To optimize performance, memory transaction implicitly operate at the most restrictive scope unless special instructions are used to change the scope of the memory access. Additionally, to make data produced at a more local scope visible to a further away scope, acquire/release markers must be inserted into the memory access stream. The *release* operation ensures that all previous memory operations have been made visible to the requested scope. The *acquire* operation ensures that we see the newest data for all memory operations below the synchronization point. These markers are compiled into cache maintenance operations and hardware fences, depending on the capabilities of the particular device and the scope of the marker.

In this paper, as we are interested in the visibility of data to other devices (NIC), so we focus on the scope farthest away from the GPU: system-scope. System-scope encompasses data visibility and ordering requirements for devices outside of the GPU, such as a NIC or host CPU. Unfortunately, current GPUs do not have mechanisms to manage system-scope from within a kernel. System-scope acquire/release markers map to kernel launch (acquire) and kernel finish (release) operations performed by CPU driver code and GPU device firmware.

Figure 1 depicts the GPU memory subsystem and highlights the main components that contribute to system-scope visibility and ordering. The three main components are:

- **GPU L2 Cache:** GPUs use a shared last level L2 cache between the different CUs. As this L2 cache is shared between all the SIMD engines, it is used as the coherency point for all operations at device-scope. The L2 cache is flushed exclusively at the end of a kernel. On some architectures, the GPU Command Processor (CP), the GPU front-end unit responsible for managing and scheduling the kernels, has the capability to flush the L2 cache. In most architectures there is no way to flush the L2 cache from the GPU shader code.
- **GPU PCIe Path IP Block (PPB):** As GPUs are typically PCIe devices, they require an IP Block that handles and manages all PCIe transactions to GPU memory. In order to maximize the performance of PCIe accesses, this block contains a cache for read operations and a write combiner for write operations. Similarly to the L2 cache, this read cache is not coherent and is only invalidated at kernel boundaries.
- **Unordered Data Fabric (DF):** To maximize throughput, GPU relaxed memory is designed with an unordered data fabric. GPUs can order their own memory requests by explicitly waiting for acknowledgements from the DF before issuing subsequent requests.

4.2 GPU Memory Model Implications for Intra-Kernel Networking

In this section, we will discuss the networking requirements for a GPU initiated networking programming model and the mismatch with the GPU memory consistency model.

GPU technologies like GPUDirect RDMA [19] and ROCm RDMA [4] optimize the data path between GPU memory and RDMA NICs by providing the NIC with the ability to directly read from/write to GPU memory. However, due to the GPU memory consistency restrictions imposed by the behavior of the components described in Section 4.1, network RDMA operations are limited to kernel boundaries. Correct data is NOT guaranteed for RDMA operations to/from GPU memory while a kernel is running [27].

In order to analyze the network requirements and their mismatch with the current GPU memory model, consider the two operations that the network performs on GPU memory (reads from GPU memory and writes to GPU memory) and their interactions with the system-scope hardware components presented in Section 4.1 .

When a NIC performs a PCIe read operation from GPU memory, it is routed to the GPU PPB Block. If the requested data is already in the PPB read cache (i.e., this address was already read by the CPU or NIC), the NIC will read the data from the PPB cache and does not access GPU memory. If the GPU has since produced updated data from the previous read, the NIC will have an inconsistent view of the data. To avoid this inconsistency, the PPB cache must be invalidated before the NIC starts reading data.

If, for the same read operation, the requested data is not in the PPB cache (or this cache was invalidated), the read request will be serviced directly from GPU memory. Unfortunately, it is still possible to read inconsistent data, as the most recent copy could be stuck in the GPU's non-coherent L2 cache. Hence, in addition to the PPB invalidation, the L2 cache must also be flushed before the NIC is allowed to read data from GPU memory.

For write requests to GPU memory, the PPB acts as a write-combiner (WC), which could re-order the write operations. Furthermore, the GPU DF itself has also the potential to re-order PCIe writes to GPU memory. These re-ordering points must be ordered when a programmer calls functions such as the $shmem\_fence()$ operation. Application writers commonly rely on $shmem\_fence()$ to order bulk data transfers with respect to a flag to notify the target that the transfer is complete. Hence a $shmem\_fence()$ operation on GIO requires a complete flush of the PPB write-combiner. Furthermore, the NIC itself needs to initiate this flush, as $shmem\_fence()$ is generally called by the initiator of the request.

5 GIO Architecture

In this section, we will discuss in detail the different designs and mechanisms that GIO uses to allow the GPU to directly interact with the NIC without the intervention of the CPU in the critical path. We first present the host runtime and system-level designs to address the memory consistency challenges described in Section 4.2. Then we discuss the GPU-side runtime architecture and the template approach to optimize performance and reduce the latency of preparing network packets by the GPU. Figure 3 highlights the key data structures that support GIO and where they exist in memory. We will refer to this figure as we explain how GIO works.

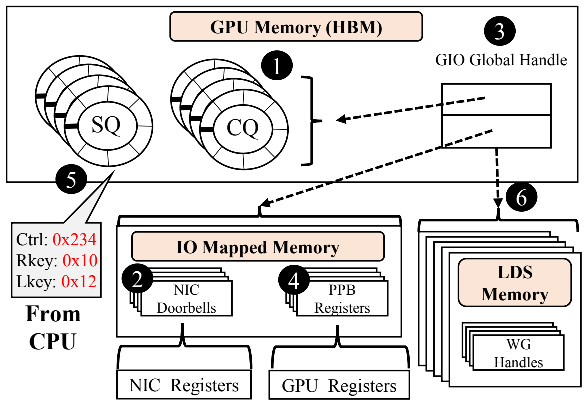


Figure 3. Illustration of important GIO data structures and where they are located in memory.

5.1 CPU-Side Runtime

The CPU handles initialization and management of network resources as well as network and GPU memory allocations. As shown in Figure 2, any GIO code will start by initializing the runtime via the host-side API call, $shmem\_init()$. During $shmem\_init()$, we perform the steps highlighted in Figure 3 and described here.

We first create the network resources (i.e., SQ and CQ) and establish connections between all of them. As GIO uses a work-group granularity, we create an SQ and a CQ for each work-group on the GPU. We rely on the InfiniBand Direct-Verbs [32] mechanism to directly interact with the NIC hardware and driver. We designed callbacks to allow the NIC driver to allocate these queues on GPU memory ❶. Next, we query the NIC driver for the addresses of the queues and the doorbell associated with each QP. We then expose these doorbells to the GPU by extracting the physical address and mapping them into the GPU's virtual memory subsystem ❷. This information is stored in a handle structure that we pass to the GPU-side runtime though the kernel launch ❸. Once the connections between the different QPs are established, the CPU allocates the symmetric heap on the GPUs and registers this memory with the NIC to allow direct RDMA accesses. Similarly, we map the physical address of the PPB's memory-mapped control register to both the GPU and NIC so that both devices may control the PPB ❹. Finally, we asynchronously prepare the network templates from the CPU and fill-up each queue as described in Section 5.3 ❺.

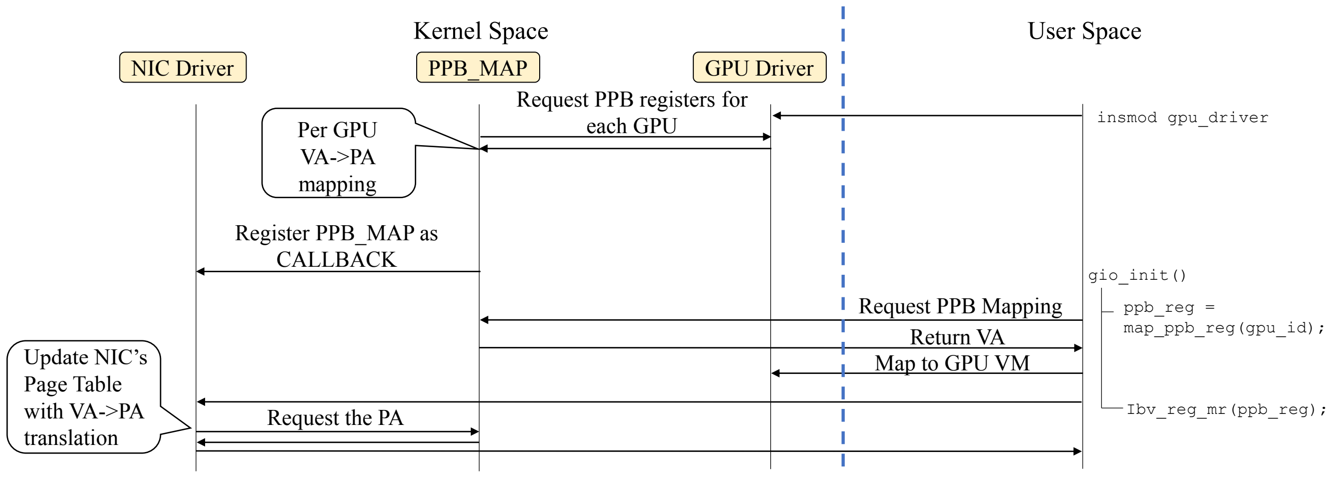


Figure 4. System-level interactions between GIO runtime, GPU driver and NIC driver.

5.1.1 Exposing and controlling the PPB

In order to expose the PPB's control registers to the GPU SIMD engines, we designed and augmented the GPU driver (*amdgpu driver*) with a novel feature that we refer to as PPB_MAP, which maps the PPB's Memory-Mapped I/O (MMIO) page to the GPU's virtual address space. This mapping is performed with assistance from the Linux kernel and the GPU driver, which creates a new entry in the GPU page table that maps to the same physical address. Furthermore, to avoid granting the user full access to all GPU MMIO registers, the GPU driver is updated to extract the PPB's registers to an empty page.

As we hinted at in Section 4.2, one-sided programming models require Write-after-Write (WaW) ordering, which requires the NIC to access the PPB's control registers. Usually, to allow a NIC to access memory, we need to register that memory with the NIC driver. This registration involves the NIC driver asking the Linux system to pin the memory region and provide the Physical Address (PA) associated with the Virtual Address (VA) of that region. However, as the PPB registers are MMIO pages, the registration will fail as the Linux memory-management system does not manage device MMIO pages.

To circumvent this limitation, the proposed PPB_MAP provides the PPB's PA to the driver by intercepting registration requests. This interception is implemented as a callback in the NIC driver via the PeerDirect interface [20]. Figure 4 highlights the steps of the interaction between the NIC driver, GPU driver, PPB_MAP, and GIO runtime. First, at boot/load time of the GPU driver, PPB_MAP function starts by exposing PPB's MMIO control registers to users and creates a map of the PPB's VA-to-PA translation for each GPU in the node. Next, we register PPB_MAP as a callback with the NIC driver. Once the PPB's MMIO space is registered with the NIC driver, the driver interacts with PPB_MAP to retrieve the PA. Then, the NIC driver inserts this VA-to-PA translation into the NIC's TLBs to allow it to access the PPB registers to perform $shmem\_fence()$, as discussed in Section 4.2. All of these steps are performed during module load or during the CPU-side GIO runtime initialization and are out of the critical path of execution.

5.1.2 Bypassing the GPU's L2 Cache

As discussed in Section 4.2, the GPU's L2 cache is not coherent with the rest of the system. Unfortunately, unlike the PPB, there are no MMIO registers that can be used to flush the L2 cache, and launching a new kernel solely for the purpose of flushing the L2 cache is too high of an overhead.

In order to satisfy the need to bypass the GPU's L2 cache for data that is accessed by the network for RDMA operations, the GPU driver is extended to provide the ability to allocate GPU memory with uncacheable pages (UC). For UC pages, stores and loads from the SIMDs will bypass the L2 and go directly to memory. This can potentially degrade the performance of the application, and we explore the impact of using uncached pages in Section 6.

Most of PPB_MAP and the UC page mapping feature is already available in the *amdgpu driver* starting with ROCm 2.5.

5.2 GPU-Side Runtime

In order to reduce memory access latency to the handle, during GPU runtime initialization each work-group copies the appropriate handle information from GPU global memory to LDS scratchpad memory as shown in ❻ of Figure 3. We then use this local handle as the OpenSHMEM context that we pass for each OpenSHMEM API call. In addition to the information about the network queues, NIC doorbells, and the PPB registers, the local context keeps track the queues' indices.

When a GPU work-group performs an OpenSHMEM call which translates to a network operation, it first locates the position of the next network command packet in its SQ and updates the packet with the dynamic information. Second, it will invalidate the PPB cache by performing a write operation to the PPB's control registers, which will flush the GPU's L2 cache. Finally, the work-group rings the NIC's doorbell associated with the SQ.

Similarly, when a work-group waits on a network operation to complete via the *shmem_quiet*() API, it locates the next entry in the CQ and polls on that memory waiting for the NIC to write the completion command.

As suggested in Section 4.2, in order to guarantee order at the remote GPU during a *shmem_fence*() operation, we implement *shmem_fence*() as an RDMA write operation to the remote GPU's PPB register to flush its PPB cache.

5.3 GIO Network Templating Design

As mentioned in Section 1, previous work on native GPU networking investigated porting the entire networking stack to a GPU [7, 28]. These designs possess a very high latency to initiate communication from the GPU. The reasons for the high latency are because preparing a network packet is predominantly a sequential series of memory store operations to fill up the different fields of a packet. These accesses are performed by a single thread of execution on the GPU. While GPUs are designed to hide memory latency to maximize throughput *across* threads, they struggle to hide the latency of a sequence of memory operations in a *single* thread.

To avoid this high latency, we propose a design that relies on templating and pre-posting the network packets from the CPU. The main idea derives from the fact that most of the information required to construct a network packet is static and known at runtime initialization; only a few pieces of information are dynamic such as the *src address*, *dst address*, and *size*. Our current design uses the CPU to prepare templates of the network packets containing the static information out of the critical path. This frees the GPU thread to simply update the dynamic packet information and ring the NIC's doorbell. With templating, the GPU thread now just performs a few stores, significantly reducing the latency to initiate a network operation.

6 Evaluation

In this section, we evaluate GIO performance and benefits on different workloads which exhibit different patterns and behaviors. First, we start with the standard network-based, micro-benchmark evaluation. Then, we explore a Jacobi 2D stencil application to evaluate the impact of GIO on structured communication patterns. Finally, we evaluate irregular communication patterns using a Sparse Triangular Solver (SpTS).

6.1 Experimental Setup

While the proposed concepts are generic and applicable for most GPU systems, our implementation uses the AMD ROCm GPU compute platform. Unfortunately, as the designs require driver and runtime patches, we are limited to a maximum of four nodes for our studies. Each node is comprised of dual socket 10-core CPUs with an AMD Radeon MI25 GPU running ROCm version 2.4. The nodes are connected with a Connect-X4 InfiniBand network running the Mellanox OFED 4.5 stack.

For our experiments, we define the following terminology to describe various baselines of interest:

GPU Initiated OpenSHMEM (GIO): The proposed GPU Initiated OpenSHMEM design where communication is performed directly within the kernel using network packet templating. It includes all of the features discussed in Section 5.

Existing Intra-Kernel Networking (IKN): Representative of existing intra-kernel networking approaches [8, 16, 31] that rely on CPU threads to perform network operations on behalf of the GPU. While IKN utilizes the same basic design as existing intra-kernel networking solutions, it does include the techniques described in Section 5 to ensure data correctness. It is important to note that our reference version of IKN delivers better performance (lower latency) than existing original approaches [7, 12–14, 28, 29].

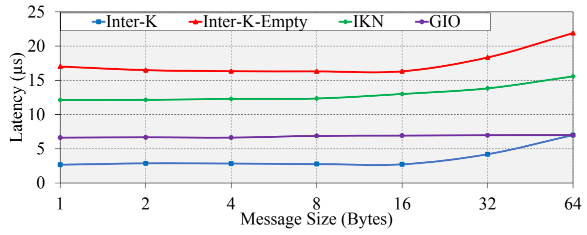
Inter-K: Traditional GPU networking mode where the CPU owns the network control path. Kernels are launched by the host to perform computation, and all networking is routed through CPU-centric MPI calls at kernel boundaries.

Inter-K-Overlap: Advanced GPU kernel-boundary networking model where communication is still performed at kernel boundaries, however, the code structure is enhanced to exploit computation/communication overlap using asynchronous MPI calls and multi-stream GPU techniques.

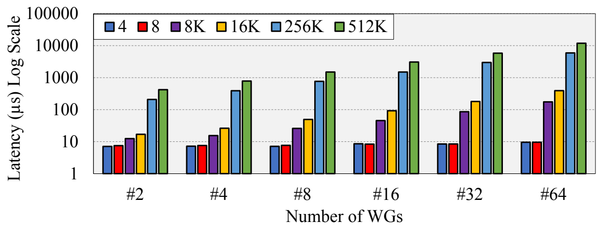
6.2 Microbenchmarks

In this section, we describe GIO performance with a network-centric microbenchmark that measures the latency of remote *Put*() operations (all implementations can achieve peak throughput, so bandwidth analysis is omitted). For the intra-kernel versions (GIO and IKN), we have redesigned the benchmark to be GPU-centric, where the network operations are performed inside the GPU kernel. The Inter-K version of the benchmark does not perform any computation (no kernel is launched). It is equivalent to the CPU version of the benchmark with the exception that the data is placed in GPU memory. Finally, a modified version of Inter-K that includes the overhead of starting an empty kernel is included in the evaluation (**Inter-K-Empty**).

Figure 5a shows the latency of a single work-group performing remote *Put*() network operations of varying payload sizes across the different versions (Inter-K, Inter-K-Empty, IKN, and GIO). As expected, compared to the CPU version



(a) Put latency sweep of payload size for a single WG.



(b) Put latency sweep of WGs with multiple message sizes.

Figure 5. Microbenchmark evaluation of GIO versus other baselines.

without any GPU (Inter-K), GIO is slightly behind in latency with 5.15µs compared to 2.8µs for 4 bytes. This is because a GPU thread is significantly slower than a CPU thread.

However, GIO provides 2*X* latency improvement compared to IKN. This comparison assesses the efficiency and the impact of 1) the template-based design and 2) the direct GPU-NIC interaction on reducing the latency of initiating the network operation. Similarly, compared to Inter-K-Empty, GIO shows up to 3*X* improvement in latency as it avoids the overhead of starting/ending a kernel for the sole purpose of performing network operations.

Figure 5b sweeps the *Put*() latency of GIO using multiple work-groups across multiple message sizes. In this example, each work-group is comprised of a single wavefront. From the figure, we see that the latency of an individual work-group is not significantly affected by the number of other work-groups resident on the GPU as long as the network is not saturated. We believe this is due to each work-group possessing independent network resources, such as QPs. For larger message sizes, the number of work-groups can impact the latency perceived by an individual work-group. This trend is because the network itself becomes the limiting factor, and increasing the number of work-groups that need to communicate with the network will eventually result in work-groups waiting for network resources.

6.3 Jacobi 2D Stencil

This section evaluates the performance of GIO 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 our example, a two-dimensional stencil is split in two dimensions over all participating nodes. The algorithm follows three main phases. First, the next value of the local stencil is calculated (either on the GPU or the host). Each element in the stencil updates its value based on the values of each of its 4 neighbors. 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. As Jacobi exhibits a regular communication pattern, the main goal of this section is to study the impact of GIO on structured applications.

In the GIO version of Jacobi, the host is no longer needed beyond data preparation. Since we can now perform network transfers from within a kernel, the main relaxation loop can be moved onto the GPU. Additionally, work-groups that perform a halo exchange on the edge of the stencil can automatically overlap with work-groups on the interior. Without intra-kernel networking, this overlap would need to be performed using an exterior and interior kernel each on its own stream. Explicit synchronization between the streams and MPI communication is also required; this increases the complexity of the algorithm and impacts the productivity of the programmer. The **Inter-K-Overlap** baseline is representative of the multi-stream overlap technique. Some results also include a **CPU** version that performs the whole Jacobi relaxation on the CPU using OpenMP with 20 threads per node. The main objective in including the CPU version in this evaluation is to validate that the problem size is big enough to require using GPUs.

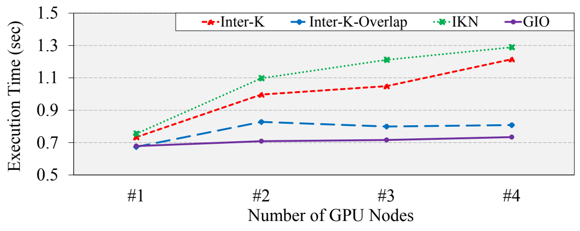


Figure 6. Weak scaling Jacobi Stencil for different GPU networking techniques with 2K * 2K problem size per GPU.

Figure 6 compares the execution time of the different Jacobi GPU networking versions on 1-4 nodes containing a single GPU each. This experiment illustrates weak-scaling where each GPU uses a fixed 2K * 2K elements problem size (each element is 8 bytes). The results indicate that GIO performs best and demonstrates linear performance scaling relative to the number of nodes. GIO outperforms the basic GPU Inter-K version by up to 40% with 4 GPU nodes. This performance uplift is because GIO was able to transparently overlap the computation and the communication without any explicit effort from the user. As suggested earlier, with a significant programming effort and a higher algorithm complexity, Inter-K-Overlap is able to overlap the communication time and achieve comparable scaling to the GIO version. Similarly, GIO also outperforms IKN, which exhibits a slight disadvantage compared to Inter-K due to the high overhead in CPU-GPU synchronization to perform the communication.

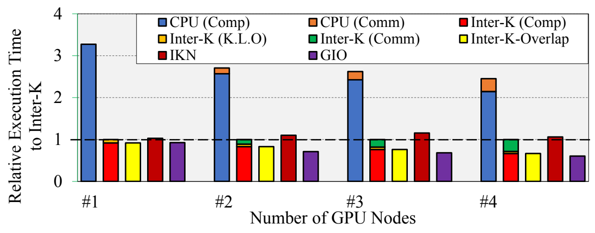


Figure 7. Jacobi Stencil performance breakdown with 2K * 2K problem size per GPU.

Figure 7 further decomposes the execution time to highlight where GIO performance benefits come from. For the GPU base version (Inter-K), we stack the execution time as computation (Comp), communication (Comm) and Kernel Launch Overhead (KLO). These components can be summed to equal total execution time since there is no communication/computation overlap in Inter-K. It illustrates the relative scaling results of Jacobi relaxation across all of our experimental configurations normalized to the Inter-K version (sum of Inter-K (comp), Inter-K (comm), Inter-K (K.L.O)). The CPU version performs the worst which indicates that the input size is big enough to offload to GPUs. Compared to Inter-K and Inter-K-Overlap versions, we can see that GIO benefits come from the computation/communication overlap. Furthermore, as GIO uses a single kernel, it was able to remove the overhead of launching a kernel that both Inter-K and Inter-K-Overlap incur.

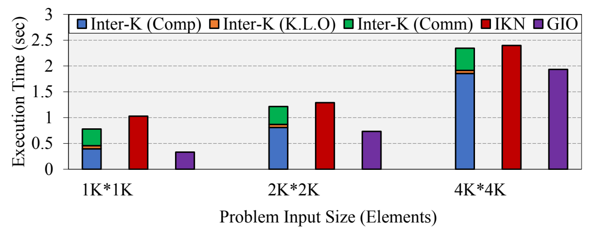


Figure 8. Performance of the different GPU networking techniques on various stencil sizes.

Figure 8 compares GIO to Inter-K and IKN on a 4 node cluster using different input problem sizes. We note four observations: 1) GIO outperforms both Inter-K and IKN versions for all input sizes as it hides both communication and kernel launch overheads. 2) Compared to Inter-K, GIO demonstrates 57%, 40% and 18% performance uplift for 1K*1K, 2K*2K and 4K*4K elements input sizes, respectively. Compared to IKN, GIO provides even more benefits with 3X, 45% and 20% improvements for 1K*1K, 2K*2K and 4K*4K input sizes, respectively. As expected the gain is reduced as the ratio of computation to communication increases with larger problem sizes. 3) The GIO runtime has a negligible overhead in GPU compute time as it exhibits similar performance to the Inter-K computation phase. This is an important observation as it illustrates that the overhead of the GIO GPU-side runtime does not significantly impact the compute portion of the application. 4) IKN performs the worst as its communication overhead is too high. However, its disadvantage reduces with increasing input size as communication becomes less of a contributing factor to the overall execution time.

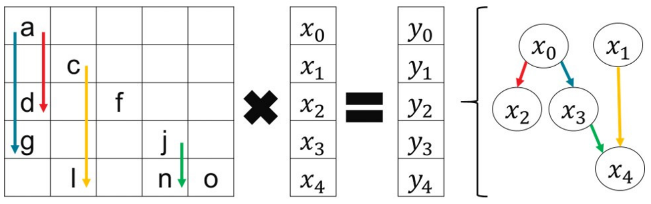


Figure 9. SpTS algorithm illustration where the colored arrows represent dependencies.

6.4 Sparse Triangular Solver (SpTS)

Our formulation of SpTS solves for the vector x in the equation $A\,x = y$ where matrix A is a sparse matrix and y is a dense vector. Both A and y are provided as inputs. The sparsity of A is the key property which makes it attractive for GPU execution. A sparse A matrix contains relatively few nonzero values, and thus has fewer dependencies between rows which provides significant opportunity for parallel solving of the rows.

To make the properties of the algorithm more concrete, Figure 9 illustrates a simple example of the dependencies in a common iteration of SpTS where the colored arrows represent dependencies that need to be resolved before an individual row can be computed. As we can see from the figure, rows 0 and 1 are completely independent of each other and can thus be solved in parallel. Once the results for x_0 are known, x_2 and x_3 also operate independently. Finally, x_4 can be solved once the values of x_3 and x_1 are computed.

Recently, a new class of GPU-aware algorithms for SpTS have been proposed [17]. These so-called *sync-free* algorithms operate in a pure data flow manner and dynamically discover dependencies as needed. Each wavefront on the GPU is assigned a row to solve, and wavefronts poll on values in memory to check if their dependencies have been satisfied. Wavefronts which have their dependencies satisfied compute x_n corresponding to their row and notify waiting wavefronts by using properly scoped memory operations. In the single GPU case, wavefronts in the same work-group

	WikiTalk	ASIC_320ks	nlpkkt160	Road_usa	Road_central
NNZ	3.072M	1.074M	118.9M	52.80M	31.02M
# Rows	2.394M	.321M	8.345M	23.95M	14.08M

Table 1. Characteristics of the input matrices for Sparse Triangular Solver (SpTS) Application

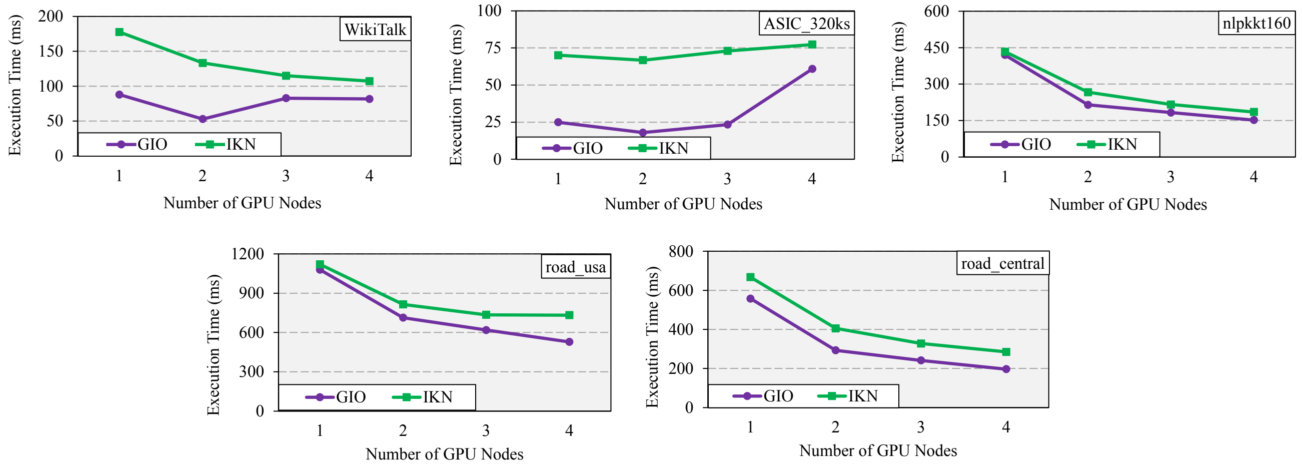


Figure 10. Performance evaluation with Sparse Triangular Solver (SpTS) Application.

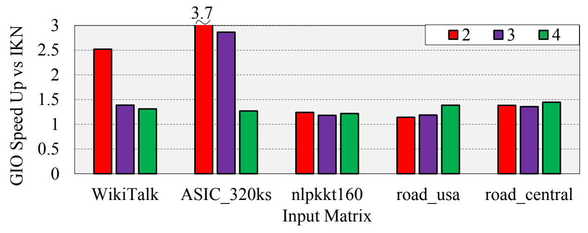


Figure 11. Speedup evaluation of GIO compared to IKN.

communicate values through LDS memory, and wavefronts on the same GPU but different work-groups communicate through the shared L2 cache (or GPU global memory).

Directly distributing the single-node sync-free algorithm is not effective with kernel-boundary communication. As the dependencies are unknown, execution is effectively serialized to a single GPU at a time. Using a block-cyclic data decomposition, each GPU would solve its portion of the matrix and send the results to the GPU that is assigned the next block in the matrix. This algorithm is expected to perform worse than a single node implementation.

Using an intra-kernel networking programming model such as GIO or IKN, the sync-free algorithm can be trivially extended across multiple nodes, since it is easy to extend the existing scoped communication model using PGAS semantics. Applications that exhibit irregular and dynamic communication patterns such as SpTS are the perfect match for intra-kernel communication models. In addition to performance, such a programming model maximizes productivity as the parallel algorithm complexity is a simple extension of the single GPU version.

For the SpTS evaluation, we use a variety of sparse matrices from different problem domains [17, 18, 23]. Table 1 highlights the basic information that affects the computation time such as the number of non-zero (NNZ) elements and the number of rows in the matrix.

For our analysis, we focus on the performance and scalability evaluation of GIO version as well as its comparison to the competing IKN version. Figure 10 shows the performance and scalability of GIO and IKN using 1, 2, 3 and 4 GPU nodes. The figure shows a graph for each input matrix. For small input matrices, as the communication time overtakes the computation, scalability is very limited and decreases for more than 2 nodes. However, for medium and large input matrices, GIO demonstrates excellent scalability with a maximum speedup of 2.8 on 4 GPU nodes (70% parallel efficiency). For instance, with the *road_central* matrix, GIO reduces the execution time from 557 *ms* on single node to 196 *ms* on 4 GPU nodes.

Compared to IKN, GIO demonstrates a significant boost in performance. Even on a single GPU, GIO outperforms IKN, which verifies the low overhead of the GIO runtime compared to IKN. Furthermore, as shown in Figure 11, for small matrices like *ASIC_320ks*, GIO shows up to 3.7X improvement on 2 nodes and more than 30% on 4 nodes. For large matrices, thanks to GIO's low latency network operations, the benefits are more significant as they increase with scale. For *road_central*, we show 38%, 35% and 44% improvement on 2, 3 and 4 GPU nodes, respectively.

	WikiTalk	ASIC_320ks	nlpkkt160	Road_usa	Road_central
L2 OFF (ms)	88.03	24.94	419.9	1078.9	557.7
L2 ON (ms)	87.14	23.56	380.7	1003.2	541.4
Overhead (%)	1	5.5	9.3	7	2.9

Table 2. Impact of disabling the GPU's L2 cache for communication buffers with SpTS.

One potential source of performance degradation for GIO is the need to bypass the GPU's L2 cache for network buffers, as discussed in Section 5.1.2. To evaluate the impact and overhead of bypassing the L2 cache, Table 2 shows the performance of the SpTS application with different input matrices on a single GPU when the L2 cache is enabled and disabled for the communication buffers. While the impact is minimal for this application and does not exceed a 9% overhead, we encourage next generation GPU hardware to provide finer-grained control and management of the L2 cache for a more natural GPU-NIC interaction.

7 Related Work

There are a number of works that support GPU networking through helper threads on the host CPU. GPUNet [12] provides a socket-based abstraction for GPUs. Both Distributed Computing for GPU Networks (DCGN) [37] and dCUDA [8] implement a device-side MPI RMA semantics for GPU kernels that relies on CPU threads to perform the network operations. Gravel [31] optimizes irregular GPU messaging applications by employing host-side coalescing of network operations. However, Gravel focuses solely on APUs.

Other recent works implement intra-kernel networking while avoiding CPU helper threads. GPU-TN [15] provides an intra-kernel networking scheme by using a mechanism based on Portals 4 triggered operations [36]. Similarly, ComP-Net [16] investigate an integrated programmable processor on the GPU to handle the communication. However, both these papers are simulation works that rely on hardware extensions to the NIC or GPU. GPU Global Address Space (GGAS) [28] implements intra-kernel networking by adding explicit hardware in the GPU to support a cluster-wide global address space. Oden et al. [30], GPUrdma [7], and Potluri et al. [33] all explore techniques to implement InfiniBand entirely on the GPU. Unfortunately, these works have challenges with performance [30] and data visibility [7, 33] related to the GPU's relaxed memory consistency model as mentioned in Section 1.

8 Conclusion

In this work, we proposed GIO, a new GPU-centric distributed programming model and runtime that uses intra-kernel networking to enable GPUs as first-class citizen on the network. Using a novel network packet templating approach, GIO improves the latency of GPU initiated communication. Furthermore, this paper analyzed the memory consistency challenges when sharing data between GPUs and NICs. We described the GIO programming model, runtime and system level designs to overcome these imitations and optimize GPU initiated networking. Overall, we show that GIO can improve application productivity and performance of both regular and irregular applications. For stencil-like applications, we showed up to 40% performance improvement. For irregular applications such as Sparse Triangular Solve (SpTS), we demonstrated that GIO was able to deliver up to 44% improvement compared to existing intra-kernel Networking designs. Furthermore, GIO achieves up-to 70% parallel efficiency versus a single node design.

GIO is an exciting first-step to elevate the GPU to a first-class citizen on the network. As such, we plan on continuing to study GIO as future work. One interesting area of study involves designing and porting more applications to use GIO. We also wish to further highlight the scalability of GIO on large scale systems.

Copyrights and Trademarks

References

[1] Elena Agostini, Davide Rossetti, and Sreeram Potluri. 2017. Offloading Communication Control Logic in GPU Accelerated Applications. In *Intl. Symp. on Cluster, Cloud and Grid Computing (CCGrid).*

[2] Johnathan Alsop, Marc S. Orr, Bradford M. Beckmann, and David A. Wood. 2016. Lazy release consistency for GPUs. In *Intl. Symp. on Microarchitecture (MICRO).*

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

[4] AMD. 2017. ROCn RDMA. https://github.com/rocmarchive/ROCnRDMA

[5] AMD. 2018. HIP: Heterogeneous-computing Interface for Portability. http://rocm-developer-tools.github.io/HIP/

[6] Matthew Baker, Swen Boehm, Aurelien Bouteiller, Barbara Chapman, Robert Cernohous, James Culhane, Tony Curtis, James Dinan, Mike Dubman, Karl Feind, Manjunath Gorentla Venkata, Max Grossman, Khaled Hamidouche, Jeff Hammond, Yossi Itigin, Bryant Lam, David Knaak, Jeff Kuehn, Jens Manser, Tiffany M. Mintz, David Ozog, Nicholas Park, Steve Poole, Wendy Poole, Swaroop Pophale, Sreeram Potluri, Howard Pritchard, Naveen Ravichandrasekaran, Michael Raymond, James Ross, Pavel Shamis, Sameer Shende, and Lauren Smith. 2018. OpenSHMEM Specification. http://openshmem.org/site/sites/default/site_files/OpenSHMEM-1.4.pdf

[7] 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.

[8] 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)*.

[9] 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 Computing* 58 (2016), 27–36.

[10] 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)*.

[11] InfiniBand Trade Association. 2000. InfiniBand Architecture Specification: Release 1.0.2. http://www.infinibandta.org/content/pages.php?pg=technology_download

[12] 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.

[13] Benjamin Klenk, Lena Oden, and Holger Froning. 2014. Analyzing Put/Get APIs for Thread-Collaborative Processors. In *Intl. Conf. on Parallel Processing (ICPP) Workshops*.

[14] 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)*.

[15] Michael LeBeane, Khaled Hamidouche, Brad Benton, Mauricio Breternitz, Steven K. Reinhardt, and Lizy K. John. 2017. GPU Triggered Networking for Intra-Kernel Communications. In *Intl. Conf. for High Performance Computing, Networking, Storage and Analysis (SC)*.

[16] Michael LeBeane, Khaled Hamidouche, Brad Benton, Mauricio Breternitz, Steven K. Reinhardt, and Lizy K. John. 2018. ComP-Net: Command Processor Networking for Efficient Intra-kernel Communications on GPUs. In *Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT'18)*.

[17] Weifeng Liu, Ang Li, Jonathan Hogg, Iain S. Duff, and Brian Vinter. 2016. A Synchronization-Free Algorithm for Parallel Sparse Triangular Solves. In *Intl. Conf. on Parallel Processing (Euro-Par)*.

[18] Weifeng Liu, Ang Li, Jonathan D. Hogg, Iain S. Duff, and Brian Vinter. 2017. Fast Synchronization-Free Algorithms for Parallel Sparse Triangular Solves with Multiple Right-Hand Sides (SpTRSM). In *Journal of Concurrency and Computation: Practice and Experience*.

[19] Mellanox. 2017. Mellanox OFED GPUDirect RDMA. http://www.mellanox.com/page/products_dyn?product_family=116

[20] Mellanox. 2018. How To Implement PeerDirect Client using MLNX_OFED. https://community.mellanox.com/s/article/howto-implement-peerdirect-client-using-mlnx-ofed

[21] Mellanox. 2018. InfiniBand Performance. http://www.mellanox.com/page/performance_infiniband

[22] MPI Forum. 2012. MPI: A Message-Passing Interface Standard. Ver. 3. www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf

[23] Maxim Naumov. 2011. Parallel Solution of Sparse Triangular Linear Systems in the Preconditioned Iterative Methods on the GPU. In *Technical Report NVR-2011-001, Nvidia*.

[24] Nvidia. 2017. GPU Applications. http://www.nvidia.com/object/gpu-applications-domain.html

[25] Nvidia. 2018. CUDA Toolkit 9.2. https://developer.nvidia.com/cuda-toolkit

[26] Nvidia. 2019. Developing a Linux Kernel Module using GPUDirect RDMA and CUDA APIs for Memory Ordering. https://docs.nvidia.com/cuda/gpudirect-rdma/index.html#sync-behavior

[27] Nvidia. 2019. GPUDirect RDMA. https://docs.nvidia.com/cuda/gpudirect-rdma/index.html#sync-behavior

[28] Lena Oden and Holger Froning. 2013. GGAS: Global GPU Address Spaces for Efficient Communication in Heterogeneous Clusters. In *Intl. Conf. on Cluster Computing (CLUSTER)*.

[29] 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.

[30] Lena Oden, Benjamin Klenk, and Holger Froning. 2014. Energy-Efficient Collective Reduce and Allreduce Operations on Distributed GPUs. In *Intl. Symp. on Cluster, Cloud and Grid Computing (CCGrid)*. 483–492.

[31] Marc S. Orr, Shuai Che, Bradford M. Beckmann, Mark Oskin, Steven K. Reinhardt, and David A. Wood. 2017. Gravel: Fine-Grain GPU-Initiated Network Messages. In *Intl. Conf. for High Performance Computing, Networking, Storage and Analysis (SC)*.

[32] Linux Man Pages. 2019. Diret Verbs. http://man7.org/linux/man-pages/man3/mlx5dv_init_obj.3.htmlS

[33] Sreeram Potluri, Anshuman Goswami, Davide Rossetti, C. J. Newburn, Manjunath G. Venkata, and Neena Imam. 2017. GPU-Centric Communication on NVIDIA GPU Clusters with InfiniBand: A Case Study with OpenSHMEM. In *Intl. Conf. on High Performance Computing (HiPC)*. 253–262.

[34] Sreeram Potluri, Davide Rossetti, Becker Donald, Poole Duncan, Venkata Manjunath, Hernandez Oscar, Shamis Pavel, Lopez M. Graham, Baker Mathew, and Poole Wendy. 2015. Exploring OpenSHMEM Model to Program GPU-based Extreme-Scale Systems. In *Workshop on OpenSHMEM and related techonlogies*.

[35] Davide Rossetti. 2015. GPUDirect Async. http://on-demand.gputechconf.com/gtc/2015/presentation/S5412-Davide-Rossetti.pdf

[36] Sandia National Laboratories. 2017. The Portals 4.1 Network Programming Interface. http://www.cs.sandia.gov/Portals/portals41.pdf

[37] Jeff A. Stuart and John D. Owens. 2009. Message Passing on Data-parallel Architectures. In *Intl. Symp. on Parallel Distributed Processing (IPDPS)*.

[38] TOP500.org. 2019. Highlights - June 2019. https://www.top500.org/lists/2019/06/