Exhibiting GPU-Centric Communications

A Survey on Communication Schemes for Distributed HPC and ML Applications

NAVEEN NAMASHIVAYAM, University of Minnesota, USA

Compute nodes on modern heterogeneous supercomputing systems comprise CPUs, GPUs, and high-speed network interconnects (NICs). Parallelization is identified as a technique for effectively utilizing these systems to execute scalable simulation and deep learning workloads. The resulting inter-process communication from the distributed execution of these parallel workloads is one of the key factors contributing to its performance bottleneck. Most programming models and runtime systems enabling the communication requirements on these systems support GPU-aware communication schemes that move the GPU-attached communication buffers in the application directly from the GPU to the NIC without staging through the host memory. A CPU thread is required to orchestrate the communication operations even with support for such GPU-awareness. This survey discusses various available *GPU-centric communication* schemes that move the control path of the communication operations from the CPU to the GPU. This work presents the need for the new communication schemes, various GPU and NIC capabilities required to implement the schemes, and the potential use-cases addressed. Based on these discussions, challenges involved in supporting the exhibited GPU-centric communication schemes are discussed.

1 Introduction

To accommodate efficient inter-process communication in modern heterogeneous supercomputing systems [1–7] comprised of CPUs and GPUs, the distributed simulation and deep learning applications executed on these large-scale systems that span across different application domains such as molecular dynamics [8–10], genome analysis [11], quantum field theory [12–14], weather modeling [15], solid and fluid mechanics [16, 17], material modeling [18], astrophysics [19], and artificial neural network training [20–22] use *GPU-aware communication* libraries. GPU-aware libraries [23–28] support performing inter-process communication operations [29, 30] involving GPU-attached memory buffers without having the application to stage them through CPU-attached memory. Remote direct memory access (RDMA) [31] and GPU vendor-specific peer-to-peer [32] data transfer mechanisms are used to implement GPU-aware inter-node and intra-node inter-process data movement operations on the scale-out [33–35] and scale-up [36, 37] interconnects available in the system, respectively.

Even with the state-of-the-art GPU-awareness in the communication stack, CPU threads are still required to orchestrate data-moving communication and inter-process synchronization operations.

Author's Contact Information: Naveen Namashivayam, University of Minnesota, USA.

CPU threads are required to synchronize with the GPU and NIC to manage this orchestration. While the programming models and runtimes support multiple effective options (polling, interrupt, or callbacks) in implementing the required synchronization, it is unnecessary for the CPU to get involved in the communication path. The reliance of an CPU thread to orchestrate the communication in these existing state-of-the-art GPU-aware communication schemes can impact (1) compute/communication overlap, (2) communication latency, and (3) GPU autonomy. The current communication model necessitates a rigid workflow and impacts the application's performance [38–43].

Modern GPUs and NICs support various capabilities, such as support for persistent GPU kernels [44], communication operations with deferred execution semantics [45], NIC-enabled memory ordering [46], and programmable processors on the NIC [47–50], making it possible to eliminate the CPU from orchestrating the communication operations. This work surveys the various available communication schemes to support GPU orchestrating the communication operations. The new communication schemes observed in this work enable efficient communication/compute overlap, improved communication latency and GPU autonomy.

1.1 Contributions

As described in this work, *GPU-centric communication* represents a communication operation that allows the GPU to directly manage the data movement operation involving a GPU-attached memory buffer. Three different GPU-centric communication schemes are explored in this work - (1) stream triggered communication, (2) kernel triggered communication, and (3) kernel initiated communication. Apart from describing the various available GPU-centric communication options, this work briefly discusses the hardware and software features required to implement them and the potential challenges involved. To our knowledge, this is the first survey to comprehensively describe the various available GPU-centric communication schemes.

1.2 Related Surveys

Other surveys in GPU-based communication focus on parallelism requirements from the applications [51–53], scalable system architectures [54, 55], and potential programming model semantics [56, 57] for enabling GPU-centric communication. Two works [58, 59] in particular attempts to explore the different properties of GPU-centric communication operations but focuses on a subset of the available options. Existing surveys focus on a subset of the available GPU-centric communication operations or implementation strategies. This work comprehensively analyzes the existing state-of-the-art communication scheme and a complete list of available GPU-centric options by decoupling the communication schemes from the implementation and programming model semantics.

1.3 Scope

This work provides a comprehensive analysis of the various GPU-centric communication schemes. It is organized as follows:

- Section 2 provides the background on the communication operations described in this work.
- Section 3 describes the currently employed baseline and state-of-the-art communication schemes used for inter-process communication.
- Section 4 exhibits on the various available GPU-centric communication schemes.
- Section 5 demonstrates the various factors impacting the communication schemes exhibited in this work and the critical software and hardware capabilities required for implementing them.

- Section 6 describes the various communication patterns addressed by the GPU-centric communication schemes, and
- Section 7 extrapolates potential challenges involved in implementing the discussed schemes and provide concluding remarks in Section 8.

2 Background

This section describes the communication operations classified in this work. An arbitrary HPC system used in this work consists of the following components, as shown in Fig. 1: (1) heterogeneous compute nodes with a host processor (CPU), accelerator units (like GPUs), and memory systems, (2) intra-node or scale-up network connecting the various compute units inside a compute node, and (3) inter-node or scale-out network connecting the various compute nodes in the system.

The host and accelerator units in the compute nodes have a shared or independent memory subsystem based on the compute node architecture. This section briefly describes the compute node architecture to provide more details on the resulting communication operations executed from the nodes. Multiple contemporary compute node architectures are discussed in Section 5.1.

Similarly, based on the system architecture, the scale-up and scale-out network interconnects can be part of the same or different networks. The topology of the inter-node and intra-node networks (fat-tree, dragonfly) is irrelevant to the communication discussion exhibited in this work.
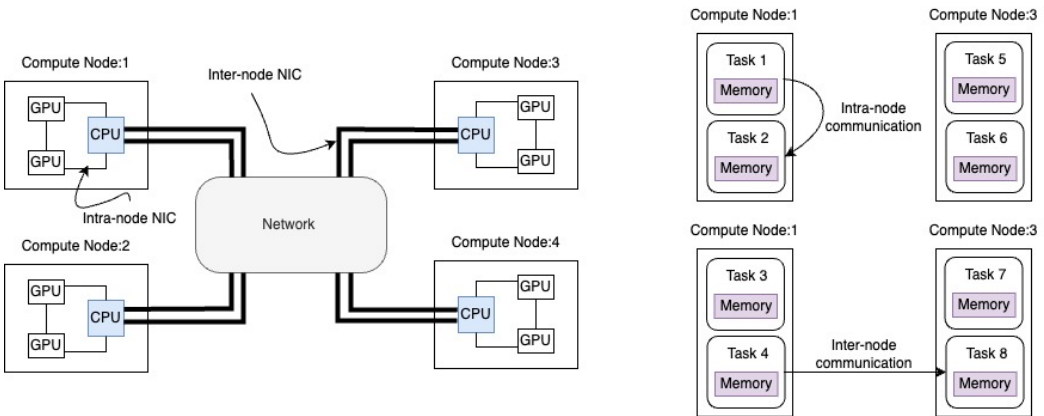

Fig. 1. Representing a traditional HPC heterogeneous system architecture with four compute nodes connected across a network. The heterogeneous compute nodes represent a host CPU attached to two GPU devices. Eight tasks are created for the distributed application, each placed on the same compute node.

Distributed memory parallelism with the SPMD (Single Program Multiple Data) programming style is the key focus for the communication operations addressed in this work. This model for parallelism demonstrates the following characteristics: (1) a set of tasks associated with a distributed application, (2) tasks parts of a distributed application use their local memory during computation, and (3) can reside on the same compute node or distributed across multiple nodes. Tasks exchange data during communication. The communication operations can be synchronous (`send` and `receive`) or asynchronous (`put` and `get`) based on the distributed programming model and the runtime systems used in the application. Fig. 1 shows a simple SPMD-style application with 8 tasks distributed across different compute nodes in the system.

2.1 Data and Control Path

Communication operations in GPU-attached memory regions discussed in this work typically comprise *data paths* and *control paths*. Data paths refer to those operations that involve moving data between the host-attached and the GPU-attached memory regions. These data movement operations can occur within the same compute node or between different compute nodes across a high-speed network. Control paths correspond to coordination operations that occur between (1) the application running on the host process, (2) the application compute kernels running on the GPU, and (4) the NIC.

This analysis focuses on the semantics describing the control path of the data movement operations when the memory buffer in the operation resides on the accelerator memory domain (specifically GPU-attached memory). This work classifies the various communication schemes used to perform the data movement operations involving the GPU-attached memory buffers in the distributed application. Section 2.2 and Section 2.3 provide a background on the intra-node and inter-node communication performed on the scale-up and scale-out interconnect, respectively. It shows the various steps involved in performing the data movement operation.

2.2 Communication Across Scale-up Interconnect

A scale-up interconnect connects the various components within a compute node. It usually supports an extensive network bandwidth. The scale of the scale-up network can vary from a small number of components, such as 4/8 GPUs (as in a fat-node architecture seems in most HPC systems), to 256 GPUs (as in Nvidia POD architecture) based on the type of the node architecture. Components within a single compute node connected using the scale-up network are usually of the same shared memory domain with or without cache coherency. This allows the communication runtime and programming models supporting the required communication operations to implement the required data movement *load* and *store* operations.

2.3 Communication Across Scale-Out Interconnect

Most scale-out interconnects connect the various compute nodes on the system. These scale-out interconnects are usually linked to the compute units in each compute node through a PCIe interface, and there can be multiple network endpoints linked per compute node called *multi-rail* systems. For example, Fig. 1 represents two network endpoints per compute node. Scale-out interconnects connecting the various compute nodes usually support less network bandwidth when compared to the scale-up interconnect due to the scale of the network that can sometimes extend beyond 8K nodes, as seen in many leadership supercomputers[1, 3, 5, 6].

Unlike the communication operation supported in scale-up networks using load and store operations, communication in scale-out networks requires complicated packet transfer. Fig. 2 represents the steps in performing the data movement operation in a scale-out network. As shown in Fig. 2, the data is moved from the source to the target process. The source and target process are in a separate compute node, with the data expected to be transferred through the scale-out network.

In this example, the data is initially available in the source process. The source process that runs on a host process (like a CPU thread or core) initiates the communication operation. The communication operation in this context can be an RDMA operation, where the NIC associated with the host process retrieves the data from the CPU-attached memory without the host or the operating system's involvement and moves the data to the target process memory.

As a first step in this data movement operation, the host associated with the source process creates a network command and enqueues it into a command queue (CQ). The network command
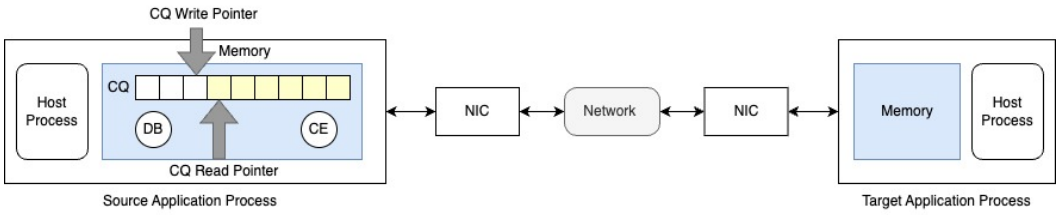
Fig. 2. Representing a simple data movement from a source to target process in a scale-out network.

is a structure that contains the metadata for the NIC to execute the required communication operation without the host or OS involvement. The metadata in the network command could include properties such as the target process information and the location and size of the memory buffer on the source process that is expected to be transferred.

CQ is a memory location on the source process monitored by both the host and the NIC using write and read pointers. The host process uses a write pointer to identify the location in the CQ and enqueue the network command.

Once the network command is enqueued into the CQ, the host process triggers the doorbell (DB) in the network to enable the NIC attached to the source to execute the communication operation. Triggering the DB moves the read pointer in the CQ for the NIC to identify the newly enqueued network command. Similarly, it moves the write pointer to enable enqueuing new network commands.

Once the data is delivered, an acknowledgment for the delivered communication operation is returned to the source process as an ack. The received ack updates a completion event (CE) in the source process. CT can be used to track the message completions. CE can be a simple monotonously increasing counter update or a full-completion event enqueued into a completion queue.

Most scale-out interconnects provide support for network transfers as described above. For brevity, the target side delivery of the message and the connection establishment between the source and the target process are not described in this work. With the intra-node and inter-node communication executed on scale-up and scale-out networks introduced, the rest of this work classifies the various available communication schemes for performing the data movement operations involving the GPU-attached memory buffers. This includes a description of the data path involved in the operation and the owner (CPU vs. GPU) of the control path of the communication operation. Section 3 and Section 4 provide a description of these various communication schemes.

3 Basic Communication Schemes

This section describes the basic communication schemes supporting the communication operations involving GPU-attached memory buffers across different programming models and runtime systems. The control path of the operation in the communication schemes discussed in this section is managed by the host process (like a CPU thread or a core).

3.1 Non GPU-Aware Communication

A non-GPU-aware communication scheme involves a data movement operation with GPU-attached memory buffers. This scheme requires the host process to orchestrate the communication and manage the operation's control path and datapath from the GPU-attached memory. It requires the data from the GPU-attached memory to be staged into the CPU-attached system memory managed by the host. Fig. 3 represents the various steps involved in the non-GPU-aware communication scheme.
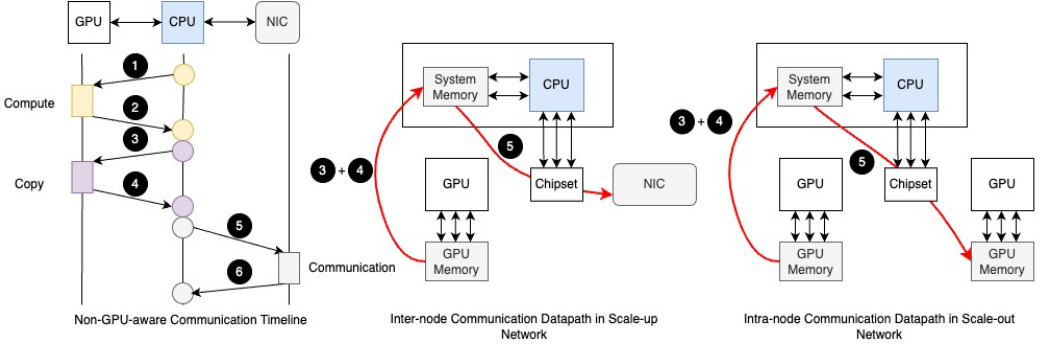
Fig. 3. Representing non-GPU-aware communication scheme.

From the application perspective, as represented by the timeline chart in Fig. 3, a non-GPU-aware communication scheme has 6 (❶ - ❻) steps. This does not include initiating the GPU-attached buffers before executing the compute operation. Step ❶ represents the host process launching the compute kernel to be executed on the GPU, and step ❷ represents the host waiting for the completion of the previously initiated compute kernel on the GPU. While the compute kernel is being executed, the host process is shown to wait for its completion. The host is blocked from performing other operations between ❶ and ❷ while the compute kernel is being executed.

Once the compute kernel is completed, the data associated with the compute kernel will still reside in the GPU-attached memory. Before initiating a non-GPU-aware communication operation, the host must stage the data from the GPU-attached memory to the host-managed CPU-attached memory. This is performed using a copy operation as shown in ❸ and ❹. The host process waits on the completion of the copy operation and is blocked from performing other operations. With the data now staged into the CPU-attached memory, the host process initiates the communication operation in step ❺ and waits for its completion in step ❻. The core semantics of the non-GPU-aware communication scheme is the following:

(1) The host process involved in the communication initiates the data movement operation.
(2) It is not possible to directly move the data from the GPU-attached memory. Instead, the host process orchestrating the communication operation is expected to stage the source buffer from the GPU-attached memory into CPU-attached system memory before initiating the data movement operation. And,
(3) The staging of the source buffer into the CPU-attached system memory (❸ and ❹) is required for both intra-node and inter-node operation on the scale-up and scale-out networks, respectively.

While the non-GPU-aware communication scheme introduces the GPU-based communication operation involving GPU-attached memory buffers, the baseline GPU-aware communication is the state-of-the-art implementation supported across different programming models and runtime systems. The baseline GPU-aware communication is described in Section 3.2.

3.2 Baseline GPU-Aware Communication

A GPU-aware communication scheme [60–64] extends the non-GPU-aware communication scheme described in section 3.1. In this communication scheme, staging the GPU-attached memory buffer into the host-managed system memory is not needed. Fig. 4 represents the various steps involved in the GPU-aware communication scheme.
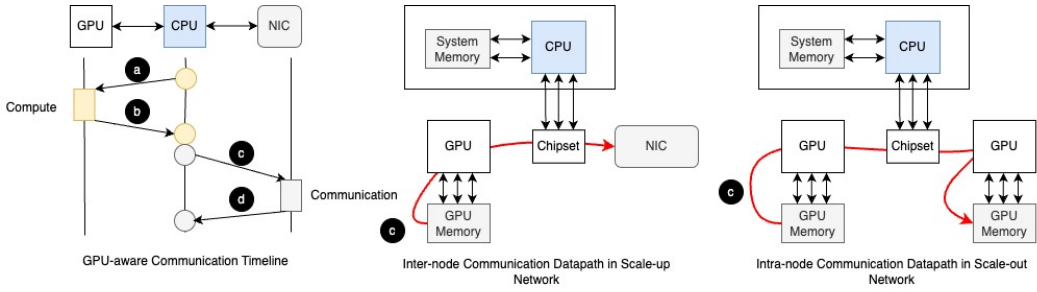
Fig. 4. Representing GPU-aware communication scheme.

As shown in Fig. 4, the need for staging the source buffer from the GPU-attached memory to the CPU-attached memory is eliminated. Irrespective of the intra-node and inter-node communication operation, GPU-aware communication from the application perspective allows the data to be directly transferred to the target process using a zero-copy communication model. Steps (a) and (b) in Fig. 4 represent the host process launching the compute kernel and waiting for completion. With the compute kernel completed, data associated with the compute kernel that residess on the GPU-attached memory is directly used for the communication operation. This is shown in step (c) and step (d). The core semantics of the GPU-aware communication scheme is the following:

(1) The host process involved in the communication initiates the data movement operation.
(2) The datapath in the communication scheme does not necessitate staging the GPU-attached buffer into the system-managed CPU-attached memory. Hence, the data associated with the compute kernel resides in the GPU-attached memory and is directly used for communication. And,
(3) The source buffer can be communicated without staging into the host-managed system memory ((c) and (d)) for both intra-node and inter-node operation on the scale-up and scale-out networks, respectively.

The basic non-GPU-aware and GPU-aware communication scheme described in sections 3.1 and 3.2 represents the basic communication schemes where the host process orchestrates the communication operation, and the control path of the operation is managed by the host process involved in the communication operation. There are multiple runtime libraries and programming models that support these communication schemes. The GPU-centric communication schemes discussed in section 4 represent the advanced communication schemes where the control path of the communication operation is moved from the CPU to the GPU.

4 GPU-Centric Communication Schemes

As mentioned in Section 3, a GPU-centric communication scheme involves the GPU managing the control path of the communication operation. This section describes three different GPU-centric communication schemes.

4.1 Stream Triggered Communication

Stream Triggered (ST) communication scheme [65–68] enables a GPU-aware application to offload the control paths of the communication operations to the underlying implementation and hardware components. This specifically includes the *GPU Stream*.

4.1.1   GPU Streams. A GPU stream [69] is a queue of device operations. GPU compute kernel concurrency is achieved by creating multiple concurrent streams. Operations issued on a stream typically run asynchronously with respect to the CPU and operations enqueued in other GPU streams. Operations in a given stream are guaranteed to be executed in FIFO order. In this work, the GPU component that provides these execution guarantees to schedule and control the execution of the enqueued operation is referred to as the *GPU Stream Execution Controller* (GPU SEC). Depending on the GPU vendor, GPU SEC can be a software, hardware, or kernel component associated with the GPU. ST scheme enables offloading the control path of the operation involving the GPU-attached memory buffers into the GPU SEC.

4.1.2   ST Description. A parallel application using the ST scheme continues to manage compute kernels on the GPU via existing mechanisms. In addition, the ST scheme allows an application process running on the CPU to define a set of ST communication operations. These communication operations can be scheduled for execution at a later point in time. More importantly, in addition to offering a deferred execution model, ST enables the GPU to get closely involved in the control paths of the communication operations.

Fig. 5 illustrates a sequence of events involved in a parallel application using ST inter-process communication and synchronization operations. An application process running on the CPU enqueues ⓐ GPU kernel K1 to the GPU stream, ⓑ triggered ST-based communication operations to the NIC, ⓒ the corresponding trigger event to the GPU stream, and ⓓ GPU kernel K2 to the same stream. The CPU returns immediately after the operations are enqueued and is not blocked on the completion of the enqueued operations. It is the GPU SEC responsibility to ① launch K1 and ② wait for its completion. Once K1 completes, ③ GPU SEC triggers the execution of previously enqueued communication operations and ④ waits for these operations to finish. Next, ⑤ the GPU SEC launches K2.
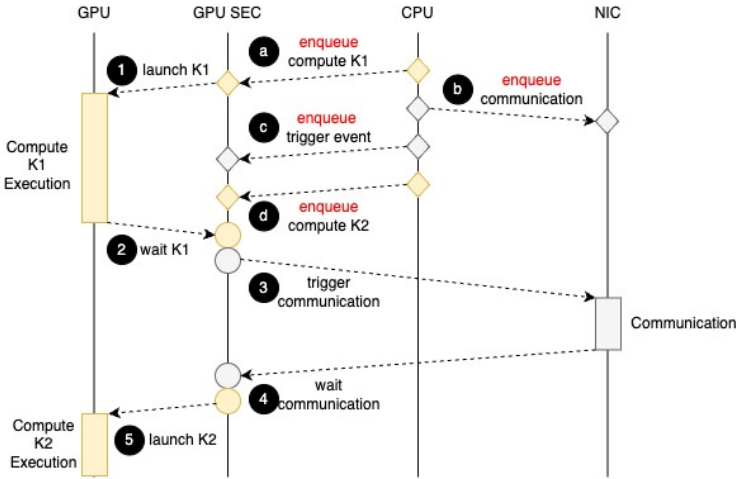

Fig. 5. Representing a GPU-aware application using ST.

The key semantics of the ST scheme includes the following:

(1) The CPU offloads the control path of the communication operation to the GPU SEC.
(2) While the CPU initiates the communication operation, the GPU SEC triggers the execution of the previously enqueued communication operation. And,

(3) Communication involving the GPU-attached memory buffers are executed at GPU kernel boundaries.

With the ST communication scheme, an application process running on the CPU enqueues operations to the NIC command queue (as described in Section 2.3) and the GPU stream, but does not get directly involved in the control paths of communication operations, subsequent kernel launch, and tear-down operations. The CPU does not directly wait for communication operations to complete. The GPU manages the control paths and eliminates potential synchronization points in the application.

4.2 Kernel Triggered Communication

Kernel Triggered (KT) communication scheme is an extension to the ST scheme. Offloading the communication control path from the CPU to GPU SEC in ST allows the CPU to exit the communication operation without involved in the communication control path. While offloading the communication control path to the GPU is useful, communication in ST still happens at compute kernel boundaries. KT allows the communication to be performed from within the compute kernel. KT offloads the communication control path to the GPU thread or thread-block based on the granularity of the communication operation.

Fig. 6. Representing various events in a GPU-aware application using KT.

Fig. 6 shows the various steps involved in the KT communication scheme. This example shows three communication operations (a , b , and c) getting enqueued into the NIC. The host process associated with the application enqueues the communication operation along with enqueuing the compute kernel *K*. Once both the communication and compute operations are enqueued, the host process returns immediately. It is good to note that the communication operations are enqueued before the compute kernel. With the compute kernel executed, previously enqueued communication operations can be triggered within the compute kernel. Steps A and B show the GPU triggering the execution of operation a and b respectively. Similarly, communication operation c can be

considered as a persistent operation getting triggered for execution multiple times ((C1), (C2), and (C3)) from within the compute kernel. The key semantics of the KT scheme includes the following:

(1) The CPU offloads the control path of the communication operation to the GPU.
(2) While the CPU initiates the communication operation, the GPU triggers executing the previously enqueued communication operation.
(3) Communication involving the GPU-attached memory buffers is executed within a GPU kernel and not across GPU kernel boundaries. And,
(4) KT supports managing persistent communication operation.

4.3 Kernel Initiated Communication

Kernel Initiated (KI) communication enables the GPU to initiate and trigger the execution of the communication operation associated with a GPU-attached memory buffers. Similar to the KT scheme, there are different granularities in initiating the communication operation like the GPU thread-level and GPU thread-block level operations. For this section, the communication granularity is not considered.

4.3.1 KI Description. KI communication scheme [70–75] involves the host process part of the application to initiate the compute kernel and exit from any further processing. With the GPU executing the enqueued compute kernel, it can initiate and execute communication operations from within the compute kernels. In this scheme, the GPU thread/thread-block initiates the communication operation by preparing the network packets necessary for performing the inter-node operation or creating an inter-process shared memory-based communication across the GPU/CPU components in the compute node. The KI scheme provides complete GPU autonomy when performing communication operations.
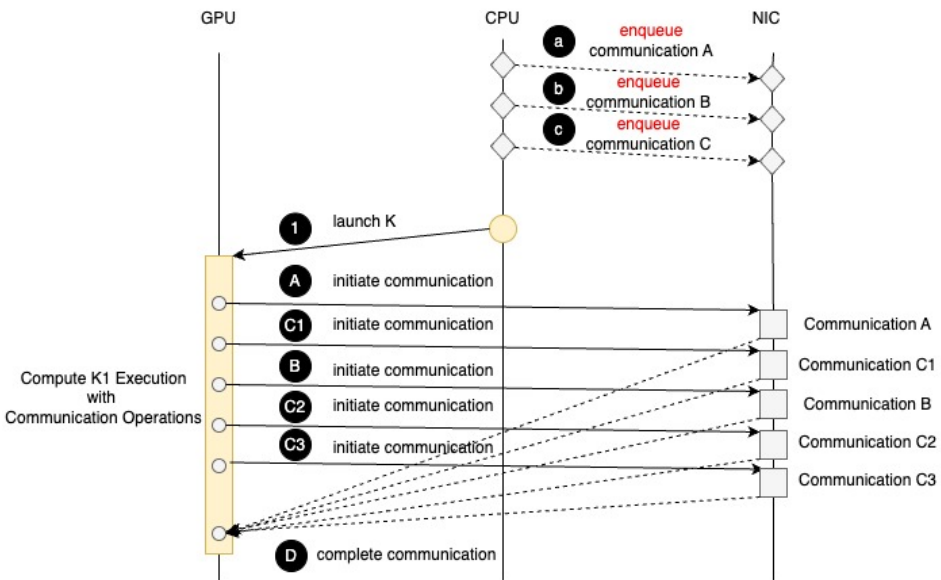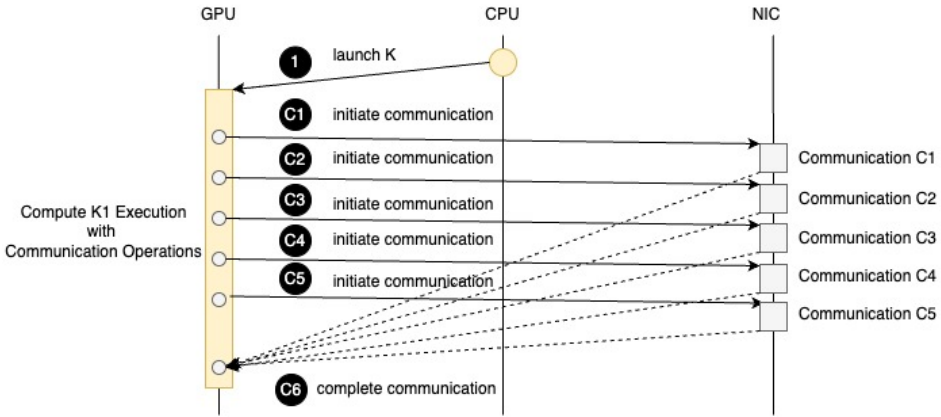

Fig. 7. Representing various events in a GPU-aware application using KI.

Fig. 7 shows the various steps involved in KI communication scheme. As shown in Fig. 7, (1) shows the host process involved in the application launches the compute kernel (*K*) into the GPU execution stream and returns immediately. Within the compute kernel, the GPU thread or thread-blocks can initiate and execute the communication operations ((C1) - (C5)) without involving the CPU in the communication control path. GPU manages both the communication execution and synchronization, determining the completion of the executed operations. In this example, a

course-grain completion is shown as step ⬤C6. The core semantics of the KI scheme involves the following:

(1) KI communication operations are initiated and executed by the GPU.
(2) The GPU manages the communication control path and orchestrates data movement without CPU assistance.
(3) Communication operations initiated by the GPU are immediately executed and are not deferred execution operations. And,
(4) Communication is performed within the compute kernel. It is not necessary to tear-down the compute kernels to perform the communication operation.

4.4 Comparing Communication Schemes

With various basic and GPU-centric communication schemes exibited in section 3 and section 4, this section compares the various communication traits across these different communication schemes. Table 1 provides the result of the analysis.

	NGA	GA	ST	KT	KI
Orchestrator	CPU	CPU	GPU SEC	GPU	GPU
Initiator	CPU	CPU	CPU	CPU	GPU
Executor	CPU	CPU	GPU SEC	GPU	GPU
Execution point	Kernel Boundary	Kernel Boundary	Kernel Boundary	Within Kernel	Within Kernel
Direct copy	No	Yes	Yes	Yes	Yes
Communication Pattern	N/A	N/A	Fixed	Fixed	Dynamic

Table 1. Table comparing different communication schemes.

As shown in Table 1, the different exibited communication schemes are compared against the different communication traits such as the component responsible for initiating, executing and orchestrating the communication operation, the execution point in the application, need for staging the data, and communication pattern suitable for the schemes. With the different communication schemes introduced in section 4, the various factors impacting the implementation of the different communication schemes and the communication patterns addressed by the communication schemes are discussed in section 5.

5 Factors Impacting GPU Communication Schemes

This section analyzes the contemporary heterogeneous system architecture landscape and the most common communication pattern involving GPU-attached memory buffers executed on these systems. The goal is to expose the criticality of the GPUs in modern supercomputing systems and the different types of network interconnects used to link the various components in the compute nodes that could impact the performance and functionality of the required data movement operations.

5.1 Heterogeneous System Landscape

The major components of a heterogeneous HPC compute node as introduced in section 2 include (a) a host processor (like a central processing unit (CPU)), (b) an accelerator (like a graphics processing unit (GPU)), and (c) a network interconnect (NIC) connecting the various components of the compute node and across the network connecting different compute nodes in the system. Some prominent heterogeneous node architectures from the Top500 [76] list are used for this analysis.

Fig. 8 represents the different node architectures used for this discussion. The network inter-connect usage in these architectures is tightly linked to support the communication schemes discussed in this work. In some modern architectures, the discrete CPUs and GPUs are replaced with specialized processors like an APU or a *superchip*. In brief, an APU is a CPU built-in with a GPU on a single die. While a superchip is similar in design with an APU, the CPU and GPU are interconnected using an high-speed network interconnect in this design.

Examples of different components include Intel Xeon [77], AMD Milan [78], and AMD Gen-ova [79] for CPUs, AMD MI250X, Nvidia A100 [80], and Intel Max Series GPUs code-named *Ponte Vecchio* [81] for GPUs, AMD MI300A [82] for APUs, Nvidia Grace-Hopper [83] for superchips, and Nvidia Infiniband [34], HPE Slingshot [33], AMD Infinity Fabric [37], Nvidia NVlink [36], and Eth-ernet for network interconnects. Fig. 8 provides a high-level representation of five heterogeneous node architectures. For brevity, vendor details and compute capabilities of each component in the architecture are not mentioned in Fig. 8.
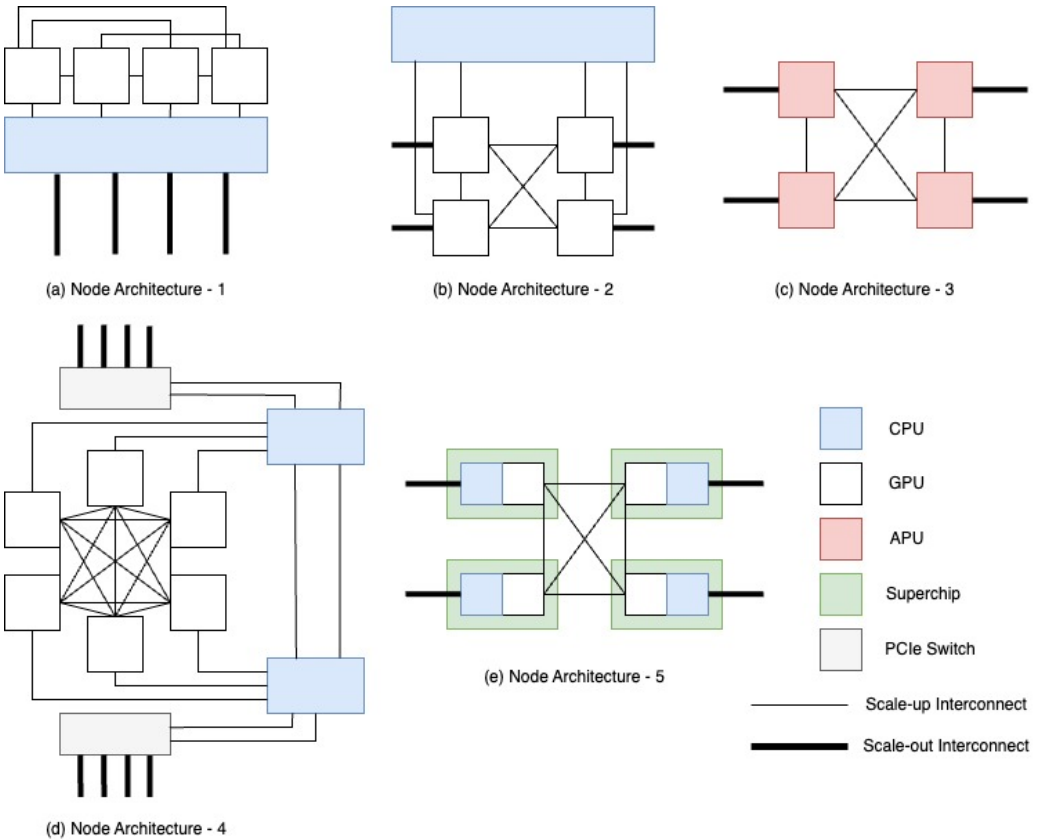
Fig. 8. Modern heterogeneous system node architectures with network links.

The following are the core observations across these different architectures and their impact in supporting the different GPU-centric communication schemes used for the data movement operations in these systems.

5.1.1 ***Number of GPUs****.* Most node architectures exhibited in Fig. 8 support multiple accelerators (GPUs or APUs) per compute node. With respect to GPU-centric communication, this is critical if

the data movement operation within a node is managed by a high-speed and high-bandwidth scale-up NIC like NVlink and Infinity Fabric. The programming models and runtime systems supporting the GPU-centric communication schemes are expected to support an independent communication middleware in the software stack exploiting the shared memory domain supported by the scale-up NIC.

5.1.2 **CPU to GPU Ratio**. The ratio of the GPUs to the CPUs across most of the architectures is high (4 : 1 or 6 : 2), Except `Node Architecture-3` and `Node Architecture-5` which represents 1 : 1 GPU:CPU ratio due to the inherent APU and superchip design. While it is ideal to offload the GPU-centric communication operations to the underlying NIC, supporting these ideal implementation designs across all system architectures is not possible. In NICs where the offload capability is not supported, one or more processes (core or thread) from the host are used as an asynchronous progress thread to implement the data movement and synchronization operations. CPU to GPU ratio determines the resources available to support such emulation design for implementing GPU-centric communication schemes.

5.1.3 **Number of NICs**. There are two different networks in the system: a scale-up network connecting the various components within the node and a scale-out network connecting the node with other nodes in the system. Effectively utilizing the available networks is critical for performing the data movement operations involving the GPU-attached memory buffers.

5.1.4 **Scale-out NIC Connection**. The compute node component on which the scale-out NIC is attached is not uniform. In some architectures, the NIC is directly attached to the GPU (`Node Architecture-2` and `Node Architecture-3`), while there are architectures where the NIC is attached to the CPU (`Node Architecture-1` and `Node Architecture-5`) or a PCIe switch (`Node Architecture-4`). The latency of the small payload communication operations is usually influenced by this design.

5.1.5 **GPU to NIC Ratio**. Most node architectures support a 1 : 1 GPU to NIC ratio. This allows the user libraries to support an application process per GPU. When more than one NIC is available per GPU, the user libraries are expected to support multiple NICs per process or force applications to adopt a multiple-process per GPU job distribution. The GPU-to-NIC ratio determines the type of process configuration associated with the GPU-centric communication schemes.

5.1.6 **Impact of PCIe Switching**. While most scale-up NICs are directly connected to a PCIe root complex, specifically in the `Node Architecture-4`, the NICs are connected to a PCIe switch. This design decision can impact the various features supported by the NIC, such as peer mapping the NIC resources in the GPU memory domain, and, in turn, impact the GPU-centric communication schemes. For example, features supporting network address translation impacting the RDMA communication and network caching of memory registered with the NIC to perform the RDMA operation can be impacted by this design.

Overall, it is essential to understand the criticality of the compute node architecture determines the host, accelerator, and NICs that link all the components within the compute node. The compute node architecture determines the proposed GPU-centric communication scheme implementation options. In this section, we briefly discussed the various compute node architecture features that could impact the performance and functionality of the GPU-centric communication schemes. Section 5.2 introduces the various GPU and NIC capabilities required to implement the exhibited GPU-centric communication schemes and section 6 classifies the various communication patterns the different communication schemes address.

5.2 Implementation Requirements

This section discusses the essential features required for implementing the communication schemes described in section 4. While multiple factors could impact the performance and functionality of the communication operations using GPU-attached memory buffers, features described in this section can be considered critical. It is important to note that some of the terminologies used in the section are specific to a particular GPU vendor (Nvidia), but most known GPU vendors (specifically AMD) have support for all the below-mentioned features.

5.2.1 ***GPUDirect Peer to Peer****.* Peer-to-peer communication allows the communication runtime (like Cray MPI) to implement zero-copy for many, but not necessarily all, data transfers across PEs sharing the same shared memory domain (intra-node communication). This feature is also referred to as GPUDirect P2P. GPUDirect Peer to Peer is supported natively by most known GPU vendors in their device drivers. This feature enables GPU-to-GPU copies, loads, and stores directly over the memory fabric (PCIe, NVLink, Infinity Fabric).

Multiple middleware libraries enable GPUDirect P2P. Inter-Process Communication (IPC) is a capability supported by both CUDA [84] and HIP [85] runtimes that allows GPUDirect P2P for on-node data transfers. For fine-grained communication control, AMD offers Heterogenous System Architecture (HSA) [86], a low-level library for performing IPC.

5.2.2 ***GPUDirect RDMA****.* GPUDirect RDMA is a technology that allows a direct path for data exchange between the GPU and a third-party peer device (like scale-out interconnect) using standard features of PCI Express. GPUDirect RDMA is used to perform zero-copy transfers of the GPU-attached memory buffers, when possible, from the GPU to the NIC without staging through a CPU-attached memory buffer. Most known device vendors provide support for GPUDirect RDMA and the scale-out interconnect attached to the GPU can perform data transfers on the GPU-attached memory buffers without staging through a host CPU-attached memory buffer.

Other intra-node memory copy libraries like GDRCopy [87] for Nvidia GPU devices are also capable of using the GPUDirect RDMA to perform effective data transfers between the GPU and the CPU. GDRCopy creates a CPU mapping of the GPU memory and uses it to perform low-latency data transfers between the GPU and CPU.

GPUDirect RDMA is a critical feature required for performing as many data transfers as possible across different compute nodes and within the same compute node.

5.2.3 ***GPUDirect Async****.* A common semantic for initiating a network data transfer involves initiating the communication to enqueue a network command entry into a predefined queue (called a command queue) and notifying the NIC about the enqueue operation by ringing a doorbell. This can trigger the NIC dequeue and execute the operation defined by the process in the network command entry.

GPUDirect Async is a new feature supported across different devices to allow the GPU to directly trigger and poll for completion of the communication operations queued to a command queue managed by the NIC. This feature depends on the ability to create GPU peer mappings of the NIC BAR space, which allows the NIC MMIO registers to be mapped into the GPU memory space so the GPU threads (for KT and KI) or the GPU SEC (for ST) can control the communication operations.

GPUDirect Async is a critical feature for supporting data transfers across different compute nodes using advanced GPU-centric communication schemes like ST, KT, and KI. HPE Slingshot NIC and Nvidia Infiniband can support the GPUDirect Async feature if the underlying GPUs provide it.

5.2.4 ***Triggered Communication Operations****.* Triggered communication operations are network data transfer operations with deferred execution semantics. While a traditional data transfer

involves the NIC immediately executing the operation as soon as the process enqueuing the data transfer rings the doorbell, notifying the enqueued network command entry, a triggered operation defers the execution of the operation until an associated condition to the network command entry is satisfied.

The conditions associated with a triggered operation include a network counter matching a posted trigger threshold. When the counter reaches the expected threshold value determined by the application, the associated data movement operation is executed.

5.2.5 **Network Descriptor Templating**. As briefly introduced in section 2.3, posting a network communication operation involves the source process generating the data transfer to enqueue a network command entry into a command queue monitored by the NIC. The placement of the command queue and the creator of the command entry are critical to enable the kernel-initiated communication scheme to be supported. Since the GPU thread or the thread-block is involved in generating the command entry, the cost of this operation is heavy as the operation is sequential.

Command entry templating allows the source process to maintain a predefined descriptor that can be filled quickly and then copied into the command queue relatively cheaply. This allows for an efficient implementation of the kernel-initiated communication scheme. Scale-out interconnects like HPE Slingshot NIC have different options to support this expected feature in enabling kernel-initiated communication.

5.2.6 **Network Interconnect On-Demand Paging**. Network memory registration (MR) is a mechanism that allows an application to describe a virtually contiguous memory location to the network adapter. The MR process pins the memory pages to avoid getting swapped and maintain the associated physical-to-virtual memory mapping. It is a key to executing RDMA operations. It creates a *key* (label) to the registered memory buffer by setting specific permissions. The key details can be transferred to a remote process in the application to perform remote RDMA operation.

While the MR process is critical for RDMA, it is usually considered a major inhibitor to RDMA adoption where the programming model or runtime systems do not naturally support the MR process in the exposed semantics.

Pinning the memory as part of the MR process can be critical for implementing GPU-centric communication operations using GPU-attached memory buffers. It can restrict the use of advanced GPU-attached memory types, such as managed memory [88] and unified virtual memory [89]. These advanced GPU memory types require swapping pages between the CPU and GPU to allow efficient programmability. Registering these memory types for GPU-centric communication is tricky.

On-Demand-Paging (ODP) is a technique that eases memory registration. Applications do not need to pin down the underlying physical pages of the address space and track the validity of the mappings. Modern NICs like Inifiband and HPE Slingshot can support ODP efficiently and ease the implementation of GPU-centric communication operations.

This section briefly discussed the critical GPU and NIC capabilities that are key for efficiently implementing GPU-centric communication schemes exhibited in section 4. Section 5.3 provides a list of libraries and runtime systems supporting the GPU-centric communication schemes using the GPU and NIC capabilities discussed in this section.

5.3 GPU-centric Communication Libraries

This section briefly discusses the various communication libraries and runtime systems supporting different GPU-centric communication schemes. For brevity, this list does not include the various GPU-aware and non-GPU-aware communication libraries along with middlewares (like libfabric [90], ibverbs [91], and UCX [92]) exposing the various GPU-centric communication semantics.

Name	ST	KT	KI	Operations			Devices		
				P2P	RMA	COLL	AMD	Nvidia	Intel
NCCL	✓			✓		✓		✓	
RCCL	✓			✓		✓	✓		
OneCCL	✓					✓			✓
NVSHMEM	✓		✓	✓				✓	
ROC_SHMEM	✓		✓	✓			✓		
Intel SHMEM			✓	✓					✓
HPE Cray MPI	✓	✓		✓	✓		✓	✓	
MPICH	✓			✓			✓	✓	✓
Libmp	✓	✓	✓	✓	✓			✓	

Table 2. Table representing the different user-level communication libraries and runtime systems supporting GPU-centric communication schemes.

Table 2 provides a comprehensive list of various user-level communication libraries and runtime systems supporting GPU-centric communication schemes. As shown in Table 2, the communication libraries and runtime systems supporting the GPU-centric communication schemes are supported across different GPU devices provided by vendors such as AMD, Intel, and Nvidia. These libraries expose the GPU-centric communication schemes through various operations supporting different communication protocols, such as: (1) *P2P* refers to the synchronous point-to-point communication model exposing message passing operations like *send* and *receive*, (2) *RMA* refers to remote memory access-based one-sided communication model supported by partitioned global address space (PGAS) based style of programming supporting operations like *puts* and *gets*, and collective communication operations like *allreduce*, *broadcast*, *reduce-scatter*, and *allgather* performed by a group of processes.

Comparing and contrasting the suitability of different operations supporting GPU-centric communication schemes in these libraries listed in Table 2 and the availability of these libraries across different network interconnects (like Nvidia Infiniband and HPE Slingshot) is beyond the scope of this work. In brief, this section shows an early adaptability of the various GPU-centric communication schemes exhibited in this work. Also, it shows the various operations (P2P, RMA, and collectives) used to expose the GPU-centric communication schemes to GPU-aware applications in different domains such as HPC and ML. Section 6 classifies the various communication patterns addressed by the different communication schemes introduced in section 4.

6 Communication Pattern

This section describes the various communication patterns addressed by the different GPU-centric schemes. The different communication patterns are broadly grouped into three categories: (1) nearest-neighbor communication, (2) data-driven communication, and (3) persistent collective communication. This section does not describe the API syntax and semantics required to implement the GPU-centric communication schemes. Instead, it provides high-level data movement requirements of some prominent HPC and ML applications that could adopt GPU-centric communication schemes.

6.1 Nearest-neighbor Communication

Near-neighbor communication appears in many contemporary HPC simulation applications. It represents one of the most essential communication patterns, primarily using two-sided point-to-point communication operations like *send* and *receive* operations supported by MPI [29]. The

communication data in this pattern is usually packed at the source process and sent to the target process, where the received data is unpacked before consumption. There is usually, at most, one message in the communication pairs for each phase, and there can be multiple neighbors per process in each phase.

With GPU-attached memory buffers associated with the communication, a GPU compute kernel performs the packing and unpacking operations, and the communication operations are performed at GPU kernel boundaries. Some prominent examples of this communication pattern in HPC simulation frameworks include adaptive mesh refinement frameworks like BoxLib [93], Navier-Stokes CFD solvers like Nekbone [16], and fusion simulation applications like Princeton Gyrokinetic Toroidal Code (GTC-P) [94].

6.2 Data-driven Communication

Large-scale data analytics problems like sorting require finding meaningful patterns in data sets. Effective solutions for addressing such problems generate fine-grained data movement patterns between unpredictable sets of processes. These resulting irregular communication patterns are different from the relatively fixed communication patterns observed in traditional simulation workloads [16, 93, 94] (as discussed in section 6.1) that exploits the structural regularity and the data locality in the problem set.

While message passing [29] is established as a *de facto* programming style for addressing the communication requirements for the HPC simulation workloads, Partitioned Global Address Space (PGAS) [30] style of programming is considered a highly effective [95–101] alternate for addressing the communication requirements of data-driven workloads. Enabling GPU-centric communication schemes with PGAS-style of programming allows enabling the data-driven communication operations to exploit using these advanced communication operations involving GPU-attached memory buffers.

6.3 Collective Communication

While sections 6.1 and 6.2 described contrasting communication patterns (regular vs. irregular), both represented a pairwise exchange model where the associated operations are between a pair of processes. This section describes the communication operations with GPU-attached memory buffers performed by a group of processes.

While collective communication operations are widely used on HPC and ML workloads [102], this section provides examples of their use in distributed ML training and inference applications. Distributed ML workloads employ different parallelism [51] strategies, such as tensor parallelism, data parallelism, layer parallelism, and hybrid parallelism, that generate data movement operations involving wide use of collective communication operations. Examples of such operations include *allreduce*, *reduce-scatter*, *allgather*, and *broadcast*. The type of collective communication depends on the parallelism strategy. Enabling GPU-centric communication schemes as collective communication operations can enable the effective execution of distributed ML workloads.

6.4 Communication Pattern Comparison

Section 6 introduced three application groups representing widely used communication patterns across HPC and ML applications employing GPU-attached memory buffers in the data movement operations. Table 3 provides a rough representation of the various traits of communication patterns discussed in this section. Based on our understanding, the potential GPU-centric schemes associated with each communication pattern are briefly mentioned in the table. However, details on the message payload size and the frequency of the operation are not discussed as these are application-dependent details.

	Nearest-Neighbor	**Data-driven**	**Collective**
Pattern	Regular	Irregular / Random	Regular
Execution	Kernel boundaries	Persistent Kernels	Kernel boundaries
Pairs	Pairwise	Pairwise	Group-of-processes
Protocols	Point-to-point	RMA	Collectives
Style	Message Passing	PGAS	NA
Examples	HPC Simulation	Data Analytics	Distributed ML
Potential Schemes	ST	KI	ST / KI

Table 3. Comparing communication patterns addressed by GPU-centric communication schemes.

Section 6 introduced three widely used communication patterns in HPC and ML workloads that can potentially employ GPU-centric communication schemes. While the communication pattern described in this section represents some critical use cases, it does not cover all the known use cases. This section can be considered a high-level representation of the potential use cases that can eventually be addressed by the GPU-centric communication schemes exhibited in section 4. Section 7 describes the various challenges in implementing the discussed schemes.

7   Challenges and Open Questions

This section outlines the central challenges and open questions in the field of GPU-centric communication to inspire future research.

(1) **Efficient compute node architecture**. While some of the features factored into the compute node design, such as the GPU-to-NIC ratio, GPU-to-CPU link (PCIe vs. proprietary link like NVlink and Infinity fabric), link connection (CPU vs. GPU), number of NICs per node, and number of GPUs per node are discussed in section 5.1, how to indentify an efficient compute node architecture for exposing GPU-centric communication? This requires further analysis of the various compute node components impacting the performance of GPU-centric communication.

(2) **Communication protocol selection**. While the requirements of the GPU-aware applications provide details on the preferred communication protocol (message passing-based synchronous vs. one-sided RMA-based asynchronous), it is not clear on the supportability of the GPU-centric communication schemes in the existing communication protocols. What is the effective communication protocol for exposing different discussed GPU-centric communication schemes?

(3) **Standardization.** The adaptability of the GPU-centric communication schemes depends on the potential to standardize the operations based on these schemes into specifications such as MPI [29] or OpenSHMEM [103]. Vendor-specific libraries exposing these communication schemes hinder portability and broad adoption. How to standardize the syntax and semantics of the operations exposing the GPU-centric communication? Is there specific programming model to target for standardization?

(4) **NIC feature requirements**. Hardware co-design with respect to the network interconnect design is critical for enabling a performant implementation of the data movement operation representing the GPU-centric communication scheme. How to co-design NIC features with the operations exposing GPU-centric communication schemes?

(5) **GPU device requirements**. Similar to previous open question, how to co-design GPU architectures with the data movement operations exposing GPU-centric communication schemes?

(6) **Software layer support**. With the GPU-centric communication support relatively recent, communication middlewares (like Libfabric, UCX, and verbs) required to expose the NIC hardware features are not designed to support these new schemes. Is it possible to integrate these new schemes into the existing middleware libraries or do we need separate libraries for these advanced GPU-aware schemes?

(7) **Application adaptability**. Most applications are designed to be GPU-aware. However, adapting to the new GPU-centric communication schemes might require extensive code change (like moving from explicit GPU kernels to long- running persistent GPU kernels). How can applications be created to use the communication operations supporting GPU-centric communication schemes?

8 Concluding Remarks

With GPUs identified as a prominent compute node component in modern heterogeneous supercomputing systems, it is imperative to determine efficient communication operations associated with GPU-attached memory buffers. The existing state-of-the-art approach, GPU-aware communication, requires a CPU to orchestrate the data movement operations associated with the GPU-attached memory buffers. GPU-centric communication is a relatively new approach that offloads the control path of the communication operation from the CPU to the GPU. This work exhibits the various available GPU-centric communication schemes. We discuss the need for these new communication schemes, factors impacting the implementation of these schemes, and potential communication patterns addressed by these new schemes. With multiple standards committees (like MPI and OpenSHMEM) and user communities interested in introducing the data movement operations exposing GPU-centric communication schemes, this work provides a detailed description of these communication schemes. It could inspire future research to address the known open challenges.

References

[1] Scott Atchley, Christopher Zimmer, John Lange, David Bernholdt, Veronica Melesse Vergara, Thomas Beck, Michael Brim, Reuben Budiardja, Sunita Chandrasekaran, Markus Eisenbach, Thomas Evans, Matthew Ezell, Nicholas Frontiere, Antigoni Georgiadou, Joe Glenski, Philipp Grete, Steven Hamilton, John Holmen, Axel Huebl, Daniel Jacobson, Wayne Joubert, Kim Mcmahon, Elia Merzari, Stan Moore, Andrew Myers, Stephen Nichols, Sarp Oral, Thomas Papatheodore, Danny Perez, David M. Rogers, Evan Schneider, Jean-Luc Vay, and P. K. Yeung. Frontier: Exploring Exascale. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2023.

[2] David Schneider. The Exascale Era is Upon Us: The Frontier supercomputer may be the first to reach 1,000,000,000,000,000,000 operations per second. *IEEE Spectrum*, 2022.

[3] National Energy Research Scientific Computing Center (NERSC). Perlmutter: High Performance Computing Optimized for Science. https://perlmutter.carrd.co, 2022.

[4] George S. Markomanolis, Aksel Alpay, Jeffrey Young, Michael Klemm, Nicholas Malaya, Aniello Esposito, Jussi Heikonen, Sergei Bastrakov, Alexander Debus, Thomas Kluge, Klaus Steiniger, Jan Stephan, Rene Widera, and Michael Bussmann. Evaluating GPU Programming Models for the LUMI Supercomputer. In *Supercomputing Frontiers: 7th Asian Conference, SCFA 2022, Singapore, March 1–3, 2022, Proceedings*, 2022.

[5] Daniele De Sensi, Lorenzo Pichetti, Flavio Vella, Tiziano De Matteis, Zebin Ren, Luigi Fusco, Matteo Turisini, Daniele Cesarini, Kurt Lust, Animesh Trivedi, Duncan Roweth, Filippo Spiga, Salvatore Di Girolamo, and Torsten Hoefler. Exploring GPU-to-GPU Communication: Insights into Supercomputer Interconnects. 2024.

[6] Argonne Leadership Computing Facility (ALCF). Aurora Exascale Supercomputer. https://www.anl.gov/article/aurora-supercomputer-heralds-a-new-era-of-scientific-innovation, 2023.

[7] Lawrence Livermore National Laboratory (LLNL). El Capitan: Preparing for NNSA's first exascale machine. https://asc.llnl.gov/exascale/el-capitan, 2024.

[8] A. P. Thompson, H. M. Aktulga, R. Berger, D. S. Bolintineanu, W. M. Brown, P. S. Crozier, P. J. in 't Veld, A. Kohlmeyer, S. G. Moore, T. D. Nguyen, R. Shan, M. J. Stevens, J. Tranchida, C. Trott, and S. J. Plimpton. LAMMPS - a flexible simulation tool for particle-based materials modeling at the atomic, meso, and continuum scales. 2022.

[9] Jeongnim Kim, Andrew D Baczewski, Anouar Beaudet, Todd Dand Benali, M Chandler Bennett, Mark A Berrill, Nick S Blunt, Edgar Josué Landinez Borda, David M Casula, Micheleand Ceperley, Simone Chiesa, Bryan K Clark, Raymond C

Clay, Kris T Delaney, Mark Dewing, Kenneth P Esler, Hongxia Hao, Olle Heinonen, Jaron T Kent, Paul R Cand Krogel, Ilkka Kylänpää, Ying Wai Li, M Graham Lopez, Ye Luo, Fionn D Malone, Richard M Martin, Amrita Mathuriya, Jeremy McMinis, Cody A Melton, Lubos Mitas, Miguel A Morales, William D Neuscamman, Ericand Parker, Sergio D Pineda Flores, Nichols A Romero, Brenda M Rubenstein, Jacqueline A R Shea, Hyeondeok Shin, Luke Shulenburger, Joshua P Tillack, Andreas F andTownsend, Norm M Tubman, Jordan E Van Der Goetz, Brett andVincent, D ChangMo Yang, Yubo Yang, Shuai Zhang, and Luning Zhao. QMCPACK: an open sourceab initioquantum monte carlo package for the electronic structure of atoms, molecules and solids. 2018.

[10] Mark James Abraham, Teemu Murtola, Roland Schulz, Szilárd Páll, Jeremy C. Smith, Berk Hess, and Erik Lindahl. Gromacs: High performance molecular simulations through multi-level parallelism from laptops to supercomputers. 2015.

[11] Jack Deslippe, Georgy Samsonidze, David A. Strubbe, Manish Jain, Marvin L. Cohen, and Steven G. Louie. Berkeleygw: A massively parallel computer package for the calculation of the quasiparticle and optical properties of materials and nanostructures. 2012.

[12] Steven Gottlieb and Sonali Tamhankar. Benchmarking MILC code with OpenMP and MPI. 2001.

[13] M. A. Clark, Bálint Joó, Alexei Strelchenko, Michael Cheng, Arjun Gambhir, and Richard Brower. Accelerating lattice qcd multigrid on gpus using fine-grained parallelization. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2016.

[14] R. Babich, M. A. Clark, B. Joó, G. Shi, R. C. Brower, and S. Gottlieb. Scaling lattice qcd beyond 100 gpus. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, 2011.

[15] Amrita Mathuriya, Deborah Bard, Peter Mendygral, Lawrence Meadows, James Arnemann, Lei Shao, Siyu He, Tuomas Kärnä, Diana Moise, Simon J. Pennycook, Kristyn Maschhoff, Jason Sewall, Nalini Kumar, Shirley Ho, Michael F. Ringenburg, Prabhat, and Victor Lee. Cosmoflow: using deep learning to learn the universe at scale. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, 2019.

[16] N. Chalmers, A. Mishra, D. McDougall, and T. Warburton. HipBone: a performance-portable gpu-accelerated c++ version of the nekbone benchmark. https://github.com/paranumal/hipBone, 2022.

[17] OLCF-6 Benchmark M-PSDNS Code. Minimalist Pseudo-Spectral DNS. https://www.olcf.ornl.gov/wp-content/uploads/OLCF-6_M-PSDNS_description-1.pdf, 2024.

[18] Paolo Giannozzi, Stefano Baroni, Nicola Bonini, Matteo Calandra, Roberto Car, Carlo Cavazzoni, Davide Ceresoli, Guido L Chiarotti, Matteo Cococcioni, Ismaila Dabo, Andrea Dal Corso, Stefano de Gironcoli, Stefano Fabris, Guido Fratesi, Ralph Gebauer, Uwe Gerstmann, Christos Gougoussis, Anton Kokalj, Michele Lazzeri, Layla Martin-Samos, Nicola Marzari, Francesco Mauri, Riccardo Mazzarello, Stefano Paolini, Alfredo Pasquarello, Lorenzo Paulatto, Carlo Sbraccia, Sandro Scandolo, Gabriele Sclauzero, Ari P Seitsonen, Alexander Smogunov, Paolo Umari, and Renata M Wentzcovitch. QUANTUM ESPRESSO: a modular and open-source software project for quantum simulations of materials. 2009.

[19] NERSC-10 Workflow benchmark suite. MTime Ordered Astrophysics Scalable Tools (TOAST) software framework. https://gitlab.com/NERSC/N10-benchmarks/toast3, 2022.

[20] NERSC-10 Workflow benchmark suite. DeepCAM AI benchmark. https://gitlab.com/NERSC/N10-benchmarks/deepcam, 2020.

[21] Brian Van Essen, Hyojin Kim, Roger Pearce, Kofi Boakye, and Barry Chen. Lbann: livermore big artificial neural network hpc toolkit. In *Proceedings of the Workshop on Machine Learning in High-Performance Computing Environments (MLHPC-2015)*, 2015.

[22] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language Models are Few-Shot Learners. In *Advances in Neural Information Processing Systems*, 2020.

[23] N. Ravi T. White L. Kaplan K. Kandalla, K. McMahon and M. Pagel. Designing the HPE Cray Message Passing Toolkit Software Stack for HPE Cray EX Supercomputers. In *Cray User Group (CUG), CUG2023*, 2023.

[24] Dhabaleswar Kumar Panda, Hari Subramoni, Ching-Hsiang Chu, and Mohammadreza Bayatpour. The MVAPICH project: Transforming research into high-performance MPI library for HPC community. 2021.

[25] W Gropp, E Lusk, N Doss, and A Skjellum. MPICH. Portable Implementation of the Standard Message Passing Interface. 1992.

[26] Richard L. Graham, Timothy S. Woodall, and Jeffrey M. Squyres. Open MPI: a flexible high performance MPI. In *Proceedings of the 6th International Conference on Parallel Processing and Applied Mathematics*, PPAM'05, 2005.

[27] Nvidia. NCCL:Optimized primitives for collective multi-GPU communication. https://github.com/NVIDIA/nccl.

[28] AMD. ROCm Communication Collectives Library (RCCL). https://github.com/ROCm/rccl.

[29] Message Passing Forum. MPI: A Message-Passing Interface Standard. Technical report, 1994.

[30] S. Amarasinghe, D. Campbell, W. Carlson, A. Chien, W. Dally, E. Elnohazy, M. Hall, R. Harrison, W. Harrod, and K. Hill. Exascale software study: Software challenges in extreme scale systems. DARPA IPTO, Air Force Research Labs, Tech. Rep, 2009.

[31] IB Verbs RDMA programming guide. https://docs.nvidia.com/networking/display/RDMAAwareProgrammingv17/RDMA+Aware+Networks+Programming+User+Manual+v1.7, 2023.

[32] Nvidia GPUDirect family. https://developer.nvidia.com/gpudirect, 2023.

[33] Daniele De Sensi, Salvatore Di Girolamo, Kim H. McMahon, Duncan Roweth, and Torsten Hoefler. An In-Depth Analysis of the Slingshot Interconnect. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2020.

[34] InfiniBand Trade Association et al. InfiniBand Architecture specification, volume 1, release 1.0, 2003.

[35] Sourav Chakraborty, Shulei Xu, Hari Subramoni, and Dhabaleswar Panda. Designing Scalable and High-Performance MPI Libraries on Amazon Elastic Fabric Adapter. In *2019 IEEE Symposium on High-Performance Interconnects (HOTI)*, 2019.

[36] Ang Li, Shuaiwen Leon Song, Jieyang Chen, Jiajia Li, Xu Liu, Nathan R. Tallent, and Kevin J. Barker. Evaluating Modern GPU Interconnect: PCIe, NVLink, NV-SLI, NVSwitch and GPUDirect. 2020.

[37] Gabin Schieffer, Ruimin Shi, Stefano Markidis, Andreas Herten, Jennifer Faj, and Ivy Peng. Understanding Data Movement in AMD Multi-GPU Systems with Infinity Fabric. 2024.

[38] Takashi Shimokawabe, Takayuki Aoki, Tomohiro Takaki, Toshio Endo, Akinori Yamanaka, Naoya Maruyama, Akira Nukada, and Satoshi Matsuoka. Peta-scale phase-field simulation for dendritic solidification on the TSUBAME 2.0 supercomputer. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, 2011.

[39] Quentin Anthony, Benjamin Michalowicz, Jacob Hatef, Lang Xu, Mustafa Abduljabbai, Aamir Shafi, Hari Subramoni, and Dhabaleswar K. Panda. Demystifying the Communication Characteristics for Distributed Transformer Models . In *2024 IEEE Symposium on High-Performance Interconnects (HOTI)*, 2024.

[40] Aashaka Shah, Vijay Chidambaram, Meghan Cowan, Saeed Maleki, Madan Musuvathi, Todd Mytkowicz, Jacob Nelson, Olli Saarikivi, and Rachee Singh. TACCL: Guiding collective algorithm synthesis using communication sketches. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, 2023.

[41] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan Ports, and Peter Richtarik. Scaling Distributed Machine Learning with In-Network Aggregation. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, 2021.

[42] Wei Deng, Junwei Pan, Tian Zhou, Deguang Kong, Aaron Flores, and Guang Lin. DeepLight: Deep Lightweight Feature Interactions for Accelerating CTR Predictions in Ad Serving. In *Proceedings of the 14th ACM International Conference on Web Search and Data Mining*, 2021.

[43] Nadeen Gebara, Manya Ghobadi, and Paolo Costa. In-network Aggregation for Shared Machine Learning Clusters. In *Proceedings of Machine Learning and Systems*, 2021.

[44] Lingqi Zhang, Mohamed Wahib, Peng Chen, Jintao Meng, Xiao Wang, and Satoshi Matsuoka. Persistent Kernels for Iterative Memory-bound GPU Applications. 2022.

[45] Brian W. Barrett, Ron Brightwell, K. Scott Hemmert, Kyle B. Wheeler, and Keith D. Underwood. Using Triggered Operations to Offload Rendezvous Messages. In *Proceedings of the 18th European MPI Users' Group Conference on Recent Advances in the Message Passing Interface*, EuroMPI'11, 2011.

[46] NVIDIA GPUDirect libgdsync. 2020.

[47] B. Ramesh N. Contini J. Yao S. Xu A. Shafi H. Subramoni D. Panda K. Suresh, B. Michalowicz. A Novel Framework for Efficient Offloading of Communication Operations to Bluefield SmartNICs . 2023.

[48] A. Shafi H. Subramoni D. Panda A. Jain, N. Alnaasan. Accelerating CPU-based Distributed DNN Training on Modern HPC Clusters using BlueField-2 DPUs . 2021.

[49] H. Subramoni J. Hashmi D. Panda M. Bayatpour, N. Sarkauskas. BluesMPI: Efficient MPI Non-blocking Alltoall Offloading Designs on Modern BlueField Smart NICs . 2021.

[50] N. Contini B. Ramesh M. Abduljabbar A. Shafi H. Subramoni D. Panda K. Suresh, B. Michalowicz. Using BlueField-3 SmartNICs to Offload Vector Operations in Krylov Subspace Methods . 2024.

[51] Tal Ben-Nun and Torsten Hoefler. Demystifying Parallel and Distributed Deep Learning: An In-depth Concurrency Analysis. 2019.

[52] Daniel Nichols, Siddharth Singh, Shu-Huai Lin, and Abhinav Bhatele. A Survey and Empirical Evaluation of Parallel Deep Learning Frameworks. 2022.

[53] Yoshua Bengio. Deep Learning of Representations: Looking Forward. In *Proceedings of the First International Conference on Statistical Language and Speech Processing*, SLSP'13, 2013.

[54] Benjamin Klenk. Communication architectures for scalable GPU-centric computing systems, 2018.

[55] Massimo Girondi, Mariano Scazzariello, Gerald Q. Maguire, and Dejan Kostić. Toward GPU-centric Networking on Commodity Hardware. In *Proceedings of the 7th International Workshop on Edge Systems, Analytics and Networking*, EdgeSys '24, 2024.

[56] Patrick G. Bridges, Anthony Skjellum, Evan D. Suggs, Derek Schafer, and Purushotham V. Bangalore. Understanding GPU Triggering APIs for MPI+X Communication. In *Recent Advances in the Message Passing Interface: 31st European MPI Users' Group Meeting*, 2024.

[57] Taylor Groves, Ben Brock, Yuxin Chen, Khaled Z. Ibrahim, Lenny Oliker, Nicholas J. Wright, Samuel Williams, and Katherine Yelick. Performance Trade-offs in GPU Communication: A Study of Host and Device-initiated Approaches. In *2020 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, 2020.

[58] GPU-Centric Communication Schemes: When CPUs Take a Back Seat. https://parcorelab.ku.edu.tr/wp-content/uploads/2024/01/Ismayil_Ismayilov_MSc_Thesis.pdf.

[59] Didem Unat, Ilyas Turimbetov, Mohammed Kefah Taha Issa, Doğan Sağbili, Flavio Vella, Daniele De Sensi, and Ismayil Ismayilov. The Landscape of GPU-Centric Communication. 2024.

[60] Hao Wang, Sreeram Potluri, Miao Luo, Ashish Kumar Singh, Sayantan Sur, and Dhabaleswar Kumar Panda. MVAPICH2-GPU: optimized GPU to GPU communication for InfiniBand clusters. 2011.

[61] Sreeram Potluri, Khaled Hamidouche, Akshay Venkatesh, Devendar Bureddy, and Dhabaleswar K. Panda. Efficient Inter-node MPI Communication Using GPUDirect RDMA for InfiniBand Clusters with NVIDIA GPUs. In *2013 42nd International Conference on Parallel Processing*, 2013.

[62] D. Bureddy A. Singh C. Rosales D. Panda S. Potluri, H. Wang. Optimizing MPI Communication on Multi-GPU Systems using CUDA Inter-Process Communication . 2012.

[63] S. Potluri K. Hamidouche J. Zhang D. Panda R. Shi, X. Lu. HAND: A Hybrid Approach to Accelerate Non-contiguous Data Movement using MPI Datatypes on GPU Clusters . 2014.

[64] H. Wang H. Subramoni D. Panda S. Potluri, D. Bureddy. Extending OpenSHMEM for GPU Computing . 2013.

[65] Naveen Namashivayam, Krishna Kandalla, Trey White, Nick Radcliffe, Larry Kaplan, and Mark Pagel. Exploring GPU Stream-Aware Message Passing using Triggered Operations. 2022.

[66] Naveen Namashivayam, Krishna Kandalla, James B White III au2, Larry Kaplan, and Mark Pagel. Exploring Fully Offloaded GPU Stream-Aware Message Passing. 2023.

[67] Hui Zhou, Ken Raffenetti, Yanfei Guo, and Rajeev Thakur. MPIX Stream: An Explicit Solution to Hybrid MPI+X Programming. In *Proceedings of the 29th European MPI Users' Group Meeting*, EuroMPI/USA '22, 2022.

[68] Joseph Schuchart and Edgar Gabriel. Stream Support in MPI Without the Churn. In *Recent Advances in the Message Passing Interface: 31st European MPI Users' Group Meeting, EuroMPI 2024*, 2024.

[69] NVIDIA CUDA C/C++ Streams and Concurrency. 2013.

[70] Chung-Hsing Hsu, Neena Imam, Akhil Langer, Sreeram Potluri, and Chris J. Newburn. An Initial Assessment of NVSHMEM for High Performance Computing. In *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2020.

[71] Khaled Hamidouche and Michael LeBeane. GPU Initiated OpenSHMEM: correct and efficient intra-kernel networking for dGPUs. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '20, 2020.

[72] Lena Oden, Holger Fröning, and Franz-Joseph Pfreundt. Infiniband-Verbs on GPU: A Case Study of Controlling an Infiniband Network Device from the GPU. In *2014 IEEE International Parallel Distributed Processing Symposium Workshops*, 2014.

[73] Sreeram Potluri, Anshuman Goswami, Davide Rossetti, C.J. Newburn, Manjunath Gorentla Venkata, and Neena Imam. GPU-Centric Communication on NVIDIA GPU Clusters with InfiniBand: A Case Study with OpenSHMEM. In *2017 IEEE 24th International Conference on High Performance Computing (HiPC)*, 2017.

[74] Alex Brooks, Philip Marshall, David Ozog, Md. Wasi ur Rahman, Lawrence Stewart, and Rithwik Tom. Intel(r) shmem: Gpu-initiated openshmem using sycl. 2024.

[75] Elena Agostini, Davide Rossetti, and Sreeram Potluri. Offloading Communication Control Logic in GPU Accelerated Applications. In *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, 2017.

[76] Jack Dongarra and Piotr Luszczek. TOP500. In *Encyclopedia of Parallel Computing*, 2011.

[77] Nevine Nassif, Ashley O. Munch, Carleton L. Molnar, Gerald Pasdast, Sitaraman V. Iyer, Zibing Yang, Oscar Mendoza, Mark Huddart, Srikrishnan Venkataraman, Sireesha Kandula, Rafi Marom, Alexandra M. Kern, Bill Bowhill, David R. Mulvihill, Srikanth Nimmagadda, Varma Kalidindi, Jonathan Krause, Mohammad M. Haq, Roopali Sharma, and Kevin Duda. Sapphire Rapids: The Next-Generation Intel Xeon Scalable Processor. In *2022 IEEE International Solid-State Circuits Conference (ISSCC)*, 2022.

[78] Tsai-Wei Wu, Stephen Lien Harrell, Geoffrey Lentner, Alex Younts, Sam Weekly, Zoey Mertes, Amiya Maji, Preston Smith, and Xiao Zhu. Defining Performance of Scientific Application Workloads on the AMD Milan Platform. In

Practice and Experience in Advanced Research Computing 2021: Evolution Across All Dimensions, PEARC '21, 2021.

[79] Timothy Prickett Morgan. Why AMD "Genoa" EPYC Server CPUs take the Heavyweight Title. https://www.nextplatform.com/2022/11/10/amd-genoa-epyc-server-cpus-take-the-heavyweight-title/.

[80] Jack Choquette, Edward Lee, Ronny Krashinsky, Vishnu Balan, and Brucek Khailany. The A100 Datacenter GPU and Ampere Architecture. In *2021 IEEE International Solid-State Circuits Conference (ISSCC)*, 2021.

[81] Hong Jiang. Intel's Ponte Vecchio GPU : Architecture, Systems & Software. In *2022 IEEE Hot Chips 34 Symposium (HCS)*, 2022.

[82] Alan Smith and Vamsi Alla. AMD Instinct MI300X Generative AI Accelerator and Platform Architecture . In *2024 IEEE Hot Chips 36 Symposium (HCS)*, 2024.

[83] Luigi Fusco, Mikhail Khalilov, Marcin Chrapek, Giridhar Chukkapalli, Thomas Schulthess, and Torsten Hoefler. Understanding Data Movement in Tightly Coupled Heterogeneous Systems: A Case Study with the Grace Hopper Superchip. 2024.

[84] Massimiliano Fatica. CUDA Toolkit and Libraries. In *2008 IEEE Hot Chips 20 Symposium (HCS)*, 2008.

[85] AMD. HIP Programming Guide. https://rocmdocs.amd.com/en/latest/index.html, 2024.

[86] HSA Foundation. HSA Runtime AMD. https://github.com/HSAFoundation/HSA-Runtime-AMD, 2017.

[87] Nvidia. GDRCopy. https://github.com/NVIDIA/gdrcopy.

[88] AMD. AMD Managed Memory. https://rocm.docs.amd.com/en/latest/conceptual/gpu-memory.html#managed-memory.

[89] AMD. AMD Unified Memory. https://rocm.docs.amd.com/projects/HIP/en/docs-6.2.0/how-to/unified_memory.html.

[90] Paul Grun, Sean Hefty, Sayantan Sur, David Goodell, Robert D. Russell, Howard Pritchard, and Jeffrey M. Squyres. A Brief Introduction to the OpenFabrics Interfaces - A New Network API for Maximizing High Performance Application Efficiency. In *Proceedings of the 2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*, HOTI '15, 2015.

[91] P. MacArthur, Q. Liu, R. D. Russell, F. Mizero, M. Veeraraghavan, and J. M. Dennis. An Integrated Tutorial on InfiniBand, Verbs, and MPI. 2017.

[92] Pavel Shamis, Manjunath Gorentla Venkata, M. Graham Lopez, Matthew B. Baker, Oscar Hernandez, Yossi Itigin, Mike Dubman, Gilad Shainer, Richard L. Graham, Liran Liss, Yiftah Shahar, Sreeram Potluri, Davide Rossetti, Donald Becker, Duncan Poole, Christopher Lamb, Sameer Kumar, Craig Stunkel, George Bosilca, and Aurelien Bouteiller. UCX: An Open Source Framework for HPC Network APIs and Beyond. In *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*, 2015.

[93] Weiqun Zhang, Ann Almgren, Vince Beckner, John Bell, Johannes Blaschke, Cy Chan, Marcus Day, Brian Friesen, Kevin Gott, Daniel Graves, Max Katz, Andrew Myers, Tan Nguyen, Andrew Nonaka, Michele Rosso, Samuel Williams, and Michael Zingale. AMReX: a framework for block-structured adaptive mesh refinement.

[94] Bei Wang, Stephane Ethier, William Tang, Timothy Williams, Khaled Z. Ibrahim, Kamesh Madduri, Samuel Williams, and Leonid Oliker. Kinetic turbulence simulations at extreme scale on leadership-class systems. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, 2013.

[95] Evangelos Georganas, Rob Egan, Steven Hofmeyr, Eugene Goltsman, Bill Arndt, Andrew Tritt, Aydin Buluç, Leonid Oliker, and Katherine Yelick. Extreme scale de novo metagenome assembly. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, SC '18, 2018.

[96] Scott French, Yili Zheng, Barbara Romanowicz, and Katherine Yelick. Parallel Hessian Assembly for Seismic Waveform Inversion Using Global Updates. In *2015 IEEE International Parallel and Distributed Processing Symposium*, 2015.

[97] Hongzhang Shan, Samuel Williams, Yili Zheng, Amir Kamil, and Katherine Yelick. Implementing High-Performance Geometric Multigrid Solver with Naturally Grained Messages. In *Proceedings of the 2015 9th International Conference on Partitioned Global Address Space Programming Models*, PGAS '15, 2015.

[98] Alok Tripathy, Katherine Yelick, and Aydin Buluc. Distributed Matrix-Based Sampling for Graph Neural Network Training. 2024.

[99] Steven Hofmeyr, Rob Egan, Evangelos Georganas, Alex Copeland, Robert Riley, Alicia Clum, Emiley Eloe-Fadrosh, Simon Roux, Eugene Goltsman, Aydin Buluç, Daniel Rokhsar, Leonid Oliker, and Katherine Yelick. Terabase-scale metagenome coassembly with MetaHipMer. 2020.

[100] Evangelos Georganas, Marquita Ellis, Rob Egan, Steven Hofmeyr, Aydin Buluç, Brandon Cook, Leonid Oliker, and Katherine Yelick. MerBench: PGAS Benchmarks for High Performance Genome Assembly. In *Proceedings of the Second Annual PGAS Applications Workshop*, 2017.

[101] Mathias Jacquelin, Yili Zheng, Esmond Ng, and Katherine Yelick. An Asynchronous Task-based Fan-Both Sparse Cholesky Solver. 2016.

[102] Kishore Punniyamurthy, Bradford Beckmann, and Khaled Hamidouche. GPU-initiated Fine-grained Overlap of Collective Communication with Computation, 2023.

[103] Barbara Chapman, Tony Curtis, Swaroop Pophale, Stephen Poole, Jeff Kuehn, Chuck Koelbel, and Lauren Smith. Introducing openshmem: Shmem for the pgas community. In *Proceedings of the Fourth Conference on Partitioned*

Global Address Space Programming Model, PGAS '10, 2010.