Designing GPU-aware OpenSHMEM for HPE Cray EX and XD Systems

Danielle Sikich
Hewlett Packard Enterprise
USA

Md Rahman
Hewlett Packard Enterprise
USA

Naveen Namashivayam
Hewlett Packard Enterprise
USA

Elliot Ronaghan
Hewlett Packard Enterprise
USA

William Okuno
Hewlett Packard Enterprise
USA

Nathan Wichmann
Hewlett Packard Enterprise
USA

Abstract

OpenSHMEM is a Partitioned Global Address Space (PGAS)-based library interface specification, developed through a collaborative standardization effort among numerous implementers and users of the SHMEM programming model. The current OpenSHMEM specification lacks GPU awareness and does not support the management of data movement operations involving GPU-attached memory buffers. Nonetheless, OpenSHMEM users are actively investigating mechanisms to enable the execution of data-driven workloads on heterogeneous system architectures.

HPE Cray OpenSHMEMX is a prominent implementation of the standard OpenSHMEM programming model, supporting a wide range of vendor platforms. This work presents the design and development of a GPU-aware and vendor-agnostic OpenSHMEM library, using HPE Cray OpenSHMEMX as a reference implementation across multiple HPE-supported system architectures. The study includes a comprehensive performance characterization of the implementation on various heterogeneous platforms and examines the necessity of advanced GPU-centric communication mechanisms to facilitate GPU-autonomous communication. The results of this work contribute to the evolution of the OpenSHMEM programming model by laying the groundwork for standardizing vendor-independent GPU support within the specification.

Keywords

OpenSHMEM, PGAS, MPI, RDMA, RMA, GPU, IPC, NCCL, RCCL, NVSHMEM, heterogeneous, GPU-aware, communication

1 Introduction

Large-scale data analytics and machine learning applications—such as sorting [10, 16, 38, 63], graph neural networks(GNN) [11], and histogramming—require finding meaningful patterns within large datasets. Analyzing these massive datasets requires state-of-the-art systems equipped with advanced processing and storage capabilities. Efficient solutions for such problems [31, 33, 34, 40, 44, 72, 79] often generate fine-grained, irregular data movement patterns among dynamic and unpredictable sets of processes. These irregular communication patterns contrast with the relatively static and structured communication observed in traditional high-performance computing (HPC) simulation workloads [2, 9, 15, 19, 24, 35, 46, 49, 78], which leverage inherent regularity and data locality.

While message passing (MPI) [29] remains the *de facto* programming model for simulation-based workloads, the Partitioned Global Address Space (PGAS) [4] model has emerged as a compelling alternative for data-driven applications. Multiple languages [13, 14, 58] and libraries [1, 3, 17, 39, 62, 73] support the PGAS-style of programming, enabling global memory spaces logically partitioned across processing elements and accessed asynchronously using RDMA-based [76] operations such as *put*, *get*, and atomic updates.

OpenSHMEM is a PGAS-based library interface specification. It is a culmination of a standardization effort among many implementers and users of the SHMEM programming model. OpenSHMEM addresses the communication requirements of data-centric workloads and serves as a communication substrate for higher-level PGAS models, including Unified Parallel C (UPC) [14] and Coarray Fortran [56, 58].

Historically, these data-driven workloads have primarily been deployed on CPU-based homogeneous supercomputing systems. However, the increasing adoption of heterogeneous [8, 48, 57, 69] architectures—comprising CPUs, GPUs, and high-speed interconnects [23, 27, 43]—has prompted SHMEM users to explore their potential for improved performance in data-driven applications.

Enabling these explorations necessitates a GPU-aware communication library. The current OpenSHMEM specification lacks GPU-awareness, meaning it cannot natively manage data movement involving GPU-attached memory buffers. In such cases, applications must stage data through CPU-attached buffers prior to initiating communication, significantly underutilizing the bandwidth capabilities of modern high-speed interconnects.

Several non-standard, vendor-specific implementations have emerged to provide GPU support in SHMEM-like environments. Notable examples include NVSHMEM (NVIDIA) [41], ROC_SHMEM (AMD) [36], and Intel SHMEM [12]. These implementations not only offer basic GPU-awareness but also introduce advanced GPU-centric communication capabilities such as kernel- and stream-triggered [5, 53, 54, 59, 68, 84] operations, enabling communication directly from within GPU kernels [37]. However, these vendor-specific approaches present two key challenges:

(1) **Limited Portability**: Proprietary implementations restrict application portability due to inconsistent feature sets, and
(2) **Lack of Standardization**: The role and requirements of advanced GPU-centric communication mechanisms are not yet fully understood, impeding efforts to integrate them into the OpenSHMEM standard.

This work aims to address both challenges by leveraging HPE Cray OpenSHMEMX [52], a premier implementation of the OpenSHMEM

programming model, widely adopted by SHMEM users for data-driven workloads. HPE Cray OpenSHMEMX introduces support for vendor-agnostic GPU-awareness and is actively evolving to incorporate advanced GPU-centric communication mechanisms that enable autonomous GPU-initiated operations.

1.1 Contributions

The primary contributions of this work are as follows:

(1) **Design of a Vendor Agnostic GPU-Aware OpenSHMEM**. This work presents the design and implementation of a GPU-aware, device-agnostic extension to the OpenSHMEM programming model, using HPE Cray OpenSHMEMX as the reference. It details the necessary API modifications and enhancements required to support seamless integration with GPU-attached memory buffers.
(2) **Performance Characterization Across Heterogeneous Architectures**. The performance of standard OpenSHMEM data movement operations is evaluated using GPU-attached memory across multiple system architectures. These include platforms featuring NVIDIA Grace-Hopper [32, 69], AMD MI250X [6, 8, 67], and AMD MI300A [48, 74] GPUs. The analysis highlights performance implications and architecture-specific considerations.
(3) **Investigation of Advanced GPU-Centric Communication Requirements**. The communication patterns of various data-driven workloads are examined to identify the limitations of current models and demonstrate the need for enhanced GPU-centric communication mechanisms. The study motivates the introduction of features that enable greater GPU autonomy, such as stream-triggered and kernel-initiated communication operations.

This work is organized as follows: Section 2 provides an overview of the OpenSHMEM programming model. Section 3 outlines the motivation for introducing GPU-awareness into the OpenSHMEM specification. Section 4 details the proposed enhancements to support GPU-aware communication, while Section 5 describes their implementation within the HPE Cray OpenSHMEMX library. Section 6 presents a performance characterization of the implementation using microbenchmarks, and Section 7 evaluates its applicability at the application level. Section 8 discusses the limitations of the proposed basic GPU-awareness support and highlights the need for advancing the model with more sophisticated GPU-centric communication schemes. Section 9 reviews related efforts in this area, and Section 10 concludes the paper.

2 Background

This section provides a brief overview on the OpenSHMEM programming model. Specifically, this section describes the various features supported by the programming model and a high-level description on its usability.

2.1 Programming Model Overview

OpenSHMEM is a library interface specification that supports the Partitioned Global Address Space (PGAS) programming model. It represents the result of a collaborative standardization effort among various stakeholders in the SHMEM programming community. The model enables the creation of a globally partitioned, symmetric memory space that is remotely addressable across all participating processes within an application. Each process can independently perform data movement operations on this shared symmetric memory, utilizing one-sided asynchronous remote read, write, and atomic operations.

A key feature of the OpenSHMEM programming model is the decoupling of data transfer from memory ordering semantics. This separation facilitates fine-grained communication, enabling low-latency, high-bandwidth data transfers with minimal software overhead while fully leveraging the capabilities of underlying high-performance network hardware supporting RDMA-based communication. Designed with input from multiple vendors and national research laboratories, OpenSHMEM specifically targets scientific applications characterized by irregular and dynamic communication patterns, where the sources and targets of data movement are not known *a priori*.

2.2 Memory Management

This section provides a high-level overview of the memory model supported by the OpenSHMEM programming model. Figure. 1 illustrates the OpenSHMEM memory model. An OpenSHMEM program consists of two types of data objects: **private data objects**, which are local to each process(PE), and **remotely accessible data objects**, which can be accessed by all participating processes (PEs).

Private data objects reside in the local memory of each PE and are only accessible by the owning PE. These objects are not visible to or accessible by other PEs using OpenSHMEM APIs.

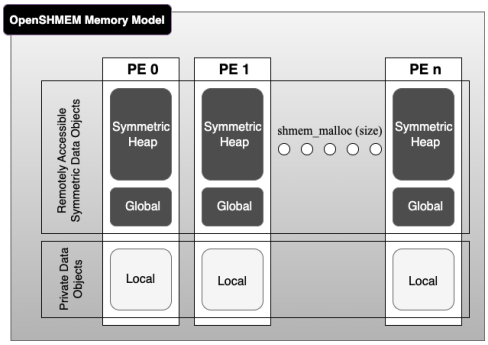


Figure 1: Illustrating OpenSHMEM Memory Model.

In contrast, remotely accessible data objects—referred to as *symmetric data objects (SDOs)* [51, 81]—can be accessed by any PE using the standard OpenSHMEM communication routines. SDOs must have identical name, type, and size across all PEs in the program. There are two categories of SDOs, based on their allocation type: (1) Global or static variables defined in the user program and supported directly by the C language. (2) Dynamically allocated variables from the symmetric heap, which is implicitly created during OpenSHMEM library initialization. SDOs allocated from the symmetric heap utilize OpenSHMEM-provided memory management APIs, including *shmem_malloc*, *shmem_calloc*, *shmem_realloc*, and *shmem_free*. Section 2.3 and 2.4 describe the data movement APIs used to access the remotely accessible data objects.

2.3 Remote Memory Access Communication

The OpenSHMEM programming model provides a suite of APIs that enable each PE to remotely access (read, write, and modify) the memory buffers of other PEs. These operations are commonly referred to as *Get*, *Put*, and atomic operations, corresponding to remote read, write, and modification semantics, respectively.

Figure. 2 illustrates the syntax of a typical *Put* operation. This API performs a data transfer from a source buffer (*src*) located on the origin PE to a destination buffer (*dst*) on a target PE, identified by its PE index (*pe*). The amount of data to be transferred is specified by the *size* parameter.

OpenSHMEM Remote Memory Access

Blocking	void shmem_put (TYPE * dst, const TYPE * src, size_t size, int pe) void shmem_ctx_put (shmem_ctx_t ctx, TYPE * dst, const TYPE * src, size_t size, int pe)
Non-Blocking	void shmem_put_nbi (TYPE * dst, const TYPE * src, size_t size, int pe) void shmem_ctx_put_nbi (shmem_ctx_t ctx, TYPE * dst, const TYPE * src, size_t size, int pe)

Figure 2: Illustrating OpenSHMEM RMA Operations.

Multiple variants of the *Put* operation exist, distinguished by their completion semantics. The blocking variant ensures that the source buffer can be safely reused once the function call returns. In contrast, the non-blocking variant does not guarantee immediate reusability of the source buffer, and therefore requires the use of explicit memory ordering operations (as discussed in Section 2.5) to enforce synchronization.

It is important to note that neither variant guarantees remote completion or global visibility of the data upon return. To achieve these guarantees, additional memory ordering operations must be explicitly invoked by the user.

2.4 Atomic Memory Operations

Similar to the Put operations described in Section 2.3, OpenSHMEM specification supports a variety of atomic memory operations (AMOs). These operations enable remote read-modify-write semantics and include primitives such as set, fetch, swap, compare-and-swap, and fetch-add.

Figure. 3 illustrates examples of both fetching and non-fetching AMOs supported by OpenSHMEM. Fetching variants return a value as part of the operation (*shmem_atomic_fetch_add*), while non-fetching variants perform the atomic update without returning a value (*shmem_atomic_add*). Both the blocking and non-blocking variants of the AMOs are supported with similar semantics as described in section 2.3.

OpenSHMEM Atomic Memory Operations

Non-Fetching	void shmem_atomic_add (TYPE * dst, TYPE value, int pe) void shmem_ctx_atomic_add (shmem_ctx_t ctx, TYPE * dst, TYPE value, int pe)
Fetching	TYPE shmem_atomic_fetch_add (TYPE * dst, TYPE value, int pe) TYPE shmem_ctx_atomic_fetch_add (shmem_ctx_t ctx, TYPE * dst, TYPE value, int pe)

Figure 3: Illustrating OpenSHMEM AMO.

2.5 Memory Ordering

The memory ordering operations provided by the OpenSHMEM programming model enable remote completion, ordering, and global visibility of previously initiated data movement operations, as discussed in Sections 2.3 and 2.4. The *shmem_quiet* operation ensures that all outstanding remote memory operations have completed and are globally visible across the system. In contrast, *shmem_fence* enforces the ordering of operations issued from a given origin PE to a specific target PE, ensuring that memory operations are observed in the correct sequence by the target. Figure 4 shows the supported memory ordering operations.

OpenSHMEM Memory Ordering Operations

Completion	void shmem_quiet (void)	void shmem_ctx_quiet (shmem_ctx_t ctx)
Ordering	void shmem_fence (void)	void shmem_ctx_fence (shmem_ctx_t ctx)

Figure 4: Illustrating OpenSHMEM Memory Ordering APIs.

2.6 Collective Communication

Collective routines [50] are defined as coordinated communication or synchronization operations performed by a group of PEs. Examples of such collective operations include *alltoall*, *allreduce*, *broadcast*, and *barrier*. Semantics of these operations are similar to such operations supported in traditional programming models such as MPI [47, 77, 82].

2.7 Point-to-Point Synchronization

The P2P synchronization operations *shmem_wait_until*, *shmem_test*, and its variants are used to portably ensure that memory access operations observe remote updates in the order enforced by the initiator PE.

2.8 Process Subset Management

OpenSHMEM Teams, also known as *process subsets*, enable the creation of a group of PEs that can be isolated and sandboxed within an application. This feature allows communication to be scoped to a specific subset of PEs, rather than involving all PEs in the program, thereby improving flexibility and scalability in collective, RMA, and atomic operations.

2.9 Communication Context

All OpenSHMEM remote memory access (RMA), atomic memory operations (AMO), and memory ordering routines must be issued on a valid communication context [25, 55]. A communication context provides an independent ordering and completion domain, enabling users to overlap communication with computation and to isolate communication streams initiated by different threads within a multithreaded PE. By leveraging multiple contexts, applications can establish independent communication pipelines to improve concurrency and performance. Figure. 2 3 4 illustrates context-based variants of the RMA and AMO operations.

This section provided a high-level overview of the OpenSHMEM programming model and its key features. It is important to note that the current specification does not explicitly define the memory type associated with remotely accessible data objects, nor does it clarify the types of memory buffers that can serve as sources in *shmem_put* operations or as destinations in *shmem_get* operations.

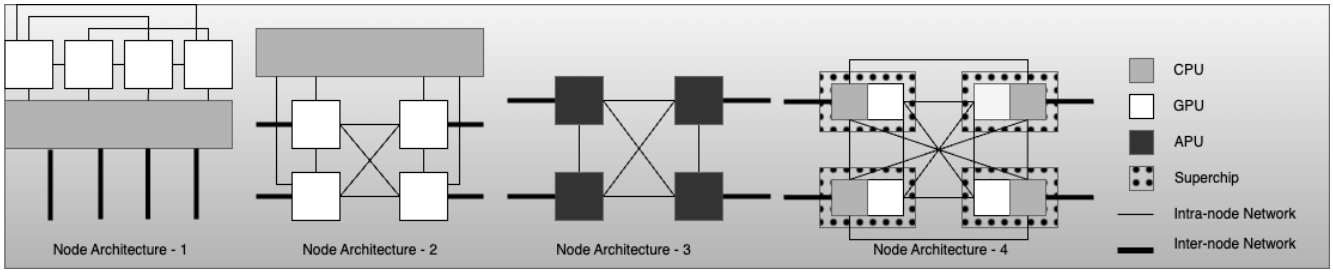


Figure 5: Illustrating Various Heterogeneous Compute Nodes supported by HPE Cray EX and HPE Cray XD System Architectures.

The specification implicitly assumes a homogeneous compute node architecture, where memory buffers are CPU-attached.

This assumption presents a critical limitation when targeting heterogeneous compute node architectures that include CPUs, GPUs, and high-speed network interconnects. To enable OpenSHMEM on such systems, the programming model must be extended to support communication involving GPU-attached memory buffers.

Section 3 motivates the need for this extension, and Section 4 outlines the proposed changes to the OpenSHMEM specification to address this gap.

3 Motivation

This section motivates the need to extend the OpenSHMEM programming model to support GPU-attached memory buffers. Two primary factors drive this requirement: (1) the growing adoption of heterogeneous node architectures, and (2) the performance overhead associated with using the communication library in the absence of GPU-awareness. These factors are discussed in detail in Sections 3.1 and 3.2.

3.1 Heterogeneous Node Architectures

Heterogeneous node architectures typically comprise CPUs, GPUs, and high-speed network interconnects. Figure 5 illustrates several widely adopted heterogeneous configurations based on HPE Cray EX and HPE Cray XD systems, deployed across premier research laboratories and commercial computing environments. Notably, several of these architectures are represented in the TOP 10 of the TOP500 list of the world's fastest supercomputers [26].

These architectures differ significantly across multiple dimensions, including: (1) Type of accelerator (discrete GPUs, APUs, or coherent processing units/superchips), (2) CPU-to-GPU ratio, (3) Inter-node network connectivity (attached to CPUs vs. GPUs), (4) NIC-to-GPU ratio, and (5) CPU-GPU interconnect bandwidth.

Each of these architectural factors can critically influence application performance. For example, the choice between discrete GPUs and APUs affects the number of available memory domains; the CPU-to-GPU ratio impacts the number of processes deployed per node; and the NIC affinity (to CPU or GPU) can affect the latency of fine-grained data transfers, especially when the source or destination buffer resides in GPU memory.

Given the performance sensitivity to architectural characteristics, communication libraries used for inter-process data movement must be capable of exposing and leveraging these hardware features effectively.

However, the current OpenSHMEM programming model lacks the necessary abstractions to fully support such heterogeneous systems. In particular, when multiple memory domains are present, the OpenSHMEM runtime lacks visibility into the memory placement of symmetric shared data objects (as discussed in Section 2.2). As a result, implementations conservatively allocate the symmetric heap in CPU-attached memory, which may lead to both performance degradation and usability limitations. Section 3.2 further explores the performance implications of operating on GPU-attached memory buffers in OpenSHMEM implementations without GPU-awareness.

3.2 Performance Limitations

This section quantifies the performance impact of using GPU-resident memory with a non-GPU-aware OpenSHMEM implementation (HPE Cray OpenSHMEMX). A modified OSU microbenchmark measures the bandwidth of *put* operations across three representative compute node architectures (*Node Architectures 1–3*, see Figure 5), with the CPU managing all control paths.

For systems with discrete GPUs (*Architectures 1 and 2*), the absence of GPU memory recognition in the communication layer necessitates explicit staging: data must be copied from GPU memory to CPU memory before transfer. Similarly, APU-based systems (*Architecture 3*), despite shared or high-bandwidth CPU-GPU memory access, still require staging due to the same GPU-unawareness. Thus, all configurations experience staging overhead.

The benchmark uses GPU-resident source buffers for data movement. Due to the lack of GPU-awareness, each transfer stages the source buffer into host memory before communication. Figure 6 compares this behavior against transfers using CPU-resident buffers. Relative to the CPU-resident baseline, only 5% of the available bandwidth is achieved for small messages, while larger messages reach up to 50% utilization. These results underscore the performance penalties imposed by GPU memory staging in non-GPU-aware OpenSHMEM environments.

The results, presented in Figure 6, demonstrate that staging significantly increases communication latency and reduces effective network utilization. These findings highlight a key limitation of current OpenSHMEM implementations on heterogeneous systems. As heterogeneous compute nodes with CPUs, GPUs, and high-speed interconnects become the norm, the need for GPU-aware communication becomes critical. Section 4 introduces a proposed extension to the OpenSHMEM programming model to address these limitations and enable efficient communication on modern architectures.

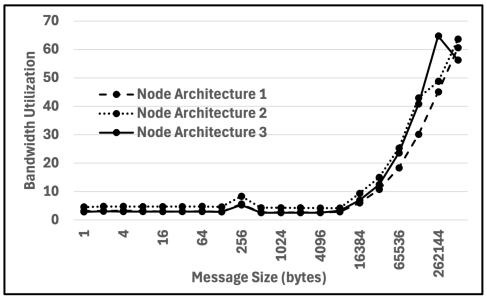


Figure 6: Impact in Staging Communication Buffers

4 Proposal

This section outlines the necessary extensions to the OpenSHMEM memory model to enable support for heterogeneous compute node architectures. It introduces new APIs and semantic modifications required to support GPU-attached memory buffers and ensure correct and efficient behavior across diverse hardware configurations. The proposed extensions are designed to be device-agnostic and portable across a range of architectures.

4.1 Implicit Memory Spaces

With a single symmetric heap created per job to support remotely accessible symmetric data objects, there has traditionally been no need to explicitly reference these heaps. As discussed in Section 2.2, these symmetric heaps are implicitly allocated by the OpenSHMEM runtime, and shared data objects are created from them using standard memory management routines such as *shmem_malloc*. In the current *OpenSHMEM 1.6* specification, user applications do not reference these heaps directly.

However, supporting OpenSHMEM on heterogeneous platforms requires each process to maintain multiple symmetric heaps—typically one CPU-attached and one GPU-attached. Therefore, it becomes necessary to reference these symmetric heaps explicitly through OpenSHMEM API extensions.

The core proposal introduces the concept of *implicit memory spaces*, which serve as labels to identify and access specific symmetric heaps. This includes *SHMEM_SPACE_DEFAULT*, which refers to the current default symmetric heap, and *SHMEM_SPACE_DEVICE*, which corresponds to a new symmetric heap allocated on device memory.

SHMEM_SYMMETRIC_SIZE is an existing environment variable used to set the size of the default heap(*SHMEM_SPACE_DEFAULT*). In parallel, a new environment variable is introduced to control the size of the device space. *SHMEM_SYMMETRIC_SIZE_DEVICE* is the new variable to control the size of *SHMEM_SPACE_DEVICE* space. Figure 7 provides a visual representation of the proposed changes for implicit memory space support.

Importantly, this proposal is both backward compatible and forward extensible. Existing APIs remain unchanged—for instance, *shmem_malloc* will continue to allocate from the default space. Additionally, this framework can be extended in the future to support explicit, user-managed memory spaces associated with OpenSHMEM Teams (see Section 2.8 for details on Teams).

4.2 Space-based Memory Management

This section defines the new memory management routines introduced to support space-based symmetric heaps. Memory operations on these spaces are supported through new space-aware memory management routines: *shmem_space_malloc*, *shmem_space_calloc*, *shmem_space_realloc*, *shmem_space_align*, and *shmem_space_free*, whose semantics are consistent with their existing counterparts in the specification.

4.3 Space-based Communication

Existing OpenSHMEM communication operations (as discussed in Section 2.3 and 2.4) can differentiate between symmetric heap allocations and static/global allocations, enabling the runtime to optimize data movement accordingly. With the proposed extension introducing multiple symmetric heaps via implicit memory spaces, the OpenSHMEM runtime can determine the memory type based on the allocation and execute data movement operations regardless of whether the memory is CPU-attached or GPU-attached.

OpenSHMEM contexts (described in Section 2.9) used to issue these operations remain agnostic to memory spaces. Any context may be used to access symmetric data objects (SDOs) allocated from either *SHMEM_SPACE_DEFAULT* or *SHMEM_SPACE_DEVICE* without requiring additional handling or context specialization.

Regarding the atomicity semantics of AMOs, the OpenSHMEM 1.6 specification defines two atomicity domains: one associated with the *SHMEM_TEAM_WORLD* process group and another with the *SHMEM_TEAM_SHARED* group. These semantics remain unchanged under the proposed extension.

In summary, the proposed changes have no impact on the semantics of existing OpenSHMEM RMA and atomic operations. Rather, they extend the programming model to support symmetric, remotely accessible heaps in GPU-attached memory, enabling their use in inter-process communication operations.

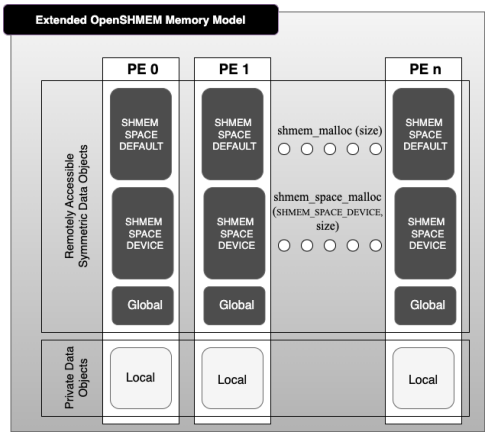


Figure 7: OpenSHMEM Memory Model Extensions.

4.4 GPU-aware Communication

While Sections 4.1 through 4.3 describe the necessary extensions to support remotely accessible GPU-attached memory for OpenSHMEM communication, this section focuses on the use of GPU-attached local memory in such operations. For instance, in the *shmem_put* operation illustrated in Figure 2, it is required that the *dst* buffer reside in symmetric memory (*i.e.*, one of the implicit spaces). However, the *src* buffer may originate from either local memory or a symmetric data object.

When the *src* buffer resides in local memory, the memory may be CPU-attached or GPU-attached. To enable support for GPU-attached local buffers, this proposal introduces a new environment variable: *SHMEM_ENABLE_DEVICE_AWARENESS*. When enabled, this variable allows OpenSHMEM communication operations to use local memory buffers allocated in any addressable memory space, including device memory.

To complement this functionality, a new query interface is introduced. *shmem_query_device_awareness* routine allows applications to determine at runtime whether GPU-aware communication is enabled via the environment variable.

A simple example program demonstrating GPU-aware OpenSHMEM communication is provided in Section 4.5.

4.5 Example

This section discusses the example program illustrated in Figure 8, which demonstrates a simple GPU-aware OpenSHMEM application using the extensions proposed in Sections4.1 through 4.4. The example shows how to allocate remotely accessible symmetric data objects (SDOs) on both the default memory space and the device-based memory space. Additionally, the program demonstrates the use of both CPU-attached and GPU-attached local memory buffers as sources in data transfer operations.

Together with the extensions described in Sections 4.1 through 4.4, these changes introduce GPU-awareness into the OpenSHMEM specification with minimal modifications. Example program shown in Figure 8 shows the usability of these proposed changes, and the design ensures both backward compatibility and forward extensibility. Section 5 describes the implementation of the proposed interface in HPE Cray OpenSHMEMX.

OpenSHMEM Example Program with GPU-awareness

```
#define SIZE 1024
#define ES sizeof(int)

shmem_init ();
int me = shmem_my_pe ();
int np  = shmem_n_pes ();

int * ddst = shmem_space_malloc (SHMEM_SPACE_DEVICE, SIZE * ES);
int * hdst = shmem_space_malloc (SHMEM_SPACE_DEFAULT, SIZE * ES);

int * hsrc = malloc (SIZE * ES);
int * dsrc = NULL;   hipMalloc ((void **)&dsrc, SIZE * ES);

for (int i = 0; i < SIZE; i++) {
  hdst[i] = np;
  hsrc[i] = me;
}

hipMemcpy(ddst, hdst, SIZE * ES, hipMemcpyHostToDevice);
hipMemcpy(dsrc, hsrc, SIZE * ES, hipMemcpyHostToDevice);
hipDeviceSynchronize();

if (me == 0) {
   shmem_int_put(ddst, ddst, SIZE, 1); /* device-to-device transfer */
   shmem_int_put(ddst, dsrc, SIZE, 1); /* local device to remote device transfer*/
   shmem_int_put(ddst, hdst, SIZE, 1); /* host-to-device transfer */
   shmem_int_put(ddst, hsrc, SIZE, 1); /* local host to remote device transfer */
}
shmem_barrier_all();

shmem_free(ddst);
shmem_free(hdst);
free(hsrc);
hipFree(dsrc);

shmem_finalize();
```

Figure 8: OpenSHMEM GPU-aware Example Program.

5 Implementation

This section provides a high-level implementation details of the proposed extensions to the OpenSHMEM programming model described in section 4.

5.1 Cray OpenSHMEMX Overview

HPE Cray OpenSHMEMX is a premier implementation of the OpenSHMEM programming model designed specifically for HPE Slingshot-based network interconnects. It effectively exposes the underlying network capabilities to applications and serves as the primary OpenSHMEM implementation for systems based on the HPE Cray EX and HPE Cray XD architectures.

Figure 9 illustrates the HPE Cray OpenSHMEMX software stack infrastructure for the HPE Slingshot NIC, highlighting the layered architecture and supported features. All features are accessible through standard OpenSHMEM C interfaces and are implemented over various transport mechanisms, including Shared Symmetric Heap (SSHEAP), XPMEM, Cross Memory Attach (CMA), Libfabric, and DMAPP. SSHEAP, XPMEM, and CMA are utilized for intra-node communication between processing elements within the same compute node, whereas Libfabric and DMAPP are employed for inter-node communication. The SMP transport layer efficiently manages a combination of inter-node and intra-node transports to optimize data movement.

The Libfabric-based transport layer is specifically optimized to deliver high-performance OpenSHMEM communication over the HPE Slingshot network, leveraging the capabilities of the HPE Cassini NIC.

5.2 GPU Transport Layer

To support GPU-specific capabilities—such as identifying memory types, allocating and managing GPU-attached memory buffers, determining GPU-to-NIC affinity, performing GPU-to-GPU interprocess communication [30, 60] using optimized GPUDirect peer-to-peer data movement [61] features offered by GPU vendors, and implementing efficient memory ordering operations—a new transport layer, referred to as the *GPU Transport Layer (GTL)*, has been introduced.

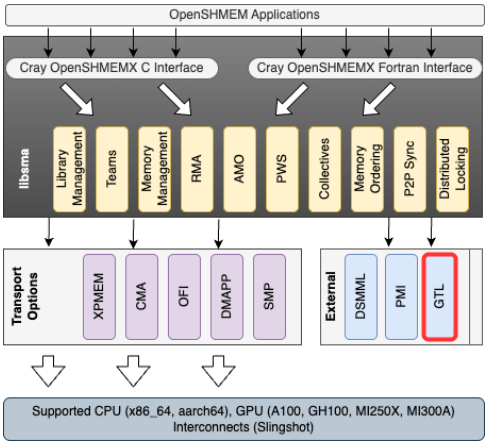


Figure 9: Different Modules in HPE Cray OpenSHMEMX

One of the key features supported by GTL is the ability to cache memory type information for buffers encountered during OpenSHMEM communication operations. As discussed in Section 4.4, OpenSHMEM allows the use of both CPU- and GPU-attached local memory in communication operations. GTL is responsible for querying the memory type and caching the result to avoid repeated, costly queries, thereby improving the performance of data transfers. GTL provides a device-agnostic interface that encapsulates the core functionality required to enable GPU-awareness within the OpenSHMEM runtime.

GTL capabilities are integrated across existing transport layers such as XPMEM, SSHEAP, and Libfabric. Based on the memory type involved in a communication operation, the OpenSHMEM runtime selects the appropriate transport and leverages GTL features to perform the data transfer efficiently. Figure 9 illustrates how GTL integrates with various transport layers in the HPE Cray OpenSHMEMX software stack.

5.3 Libfabric HPE Slingshot Transport

While Section 5.2 introduced the GPU Transport Layer (GTL) to enable GPU-aware communication in HPE Cray OpenSHMEMX and provided an overview of its role in facilitating intra-node data transfers using GPU-attached memory buffers, this section focuses on enabling efficient inter-node communication with GPU-attached memory. Specifically, it highlights the capabilities of the HPE Slingshot NIC that support high-performance GPU-aware one-sided RDMA operations. HPE Cray OpenSHMEMX leverages the Libfabric network interfaces to implement these capabilities on the HPE Slingshot NIC.

The following key features of the HPE Slingshot interconnect are leveraged to achieve GPU-awareness in the HPE Cray OpenSHMEMX implementation:

5.3.1 GPU PeerDirect. GPU PeerDirect is a technology that enables direct communication between GPUs and peripheral devices such as network interface cards (NICs), bypassing the CPU. This mechanism significantly improves data transfer efficiency by reducing latency and eliminating the need for intermediate staging through CPU-attached memory. The HPE Slingshot NIC supports this capability, allowing for high-performance GPU-to-GPU data transfers without requiring CPU involvement, thereby enhancing the effectiveness of GPU-aware communication in distributed applications.

5.3.2 NIC On Demand Paging. On-demand paging (ODP) in the context of a Network Interface Card (NIC) refers to a memory management technique where the NIC requests memory pages from the operating system only when they are needed for data transfer, rather than pre-registering the entire memory region upfront. This approach improves memory efficiency, particularly when working with large memory regions or when the NIC activity is sparse.

ODP is essential for enabling GPU-awareness in communication libraries, particularly in managing dynamic page migrations during application execution. Similar to memory type caching in the GPU Transport Layer (GTL), the underlying Libfabric network interfaces—used with the HPE Slingshot NIC—cache the encountered memory buffers and register them with the NIC for efficient reuse. In heterogeneous compute node architectures, where page migration is common, it becomes critical to detect such migrations and verify the validity of cached buffer registrations. ODP provides the mechanisms needed to manage these transitions effectively, making it a key capability for supporting GPU-aware communication in HPE Cray OpenSHMEMX.

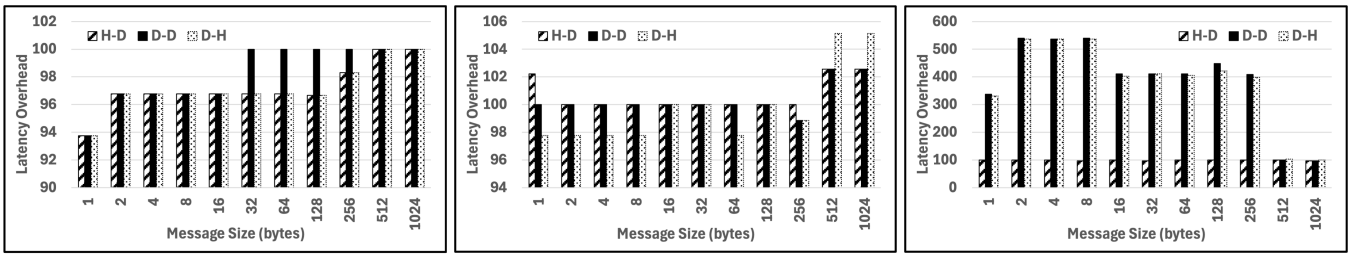
Overall, this section provided a brief overview of the HPE Cray OpenSHMEMX library design and introduced the key architectural changes implemented to enable GPU-awareness, as proposed in Section 4. It further discussed the essential network capabilities—GPUDirect Peer-to-Peer, GPU PeerDirect, and On-Demand Paging (ODP)—that are leveraged to realize the proposed extensions for GPU-aware OpenSHMEM communication. Section 6 will present the performance characteristics of this implementation across different heterogeneous system architectures.

6 Performance Analysis

This section presents a performance overview of the GPU-aware HPE Cray OpenSHMEMX implementation across three distinct node architectures. The objective is to evaluate the portability and effectiveness of a standard GPU-aware OpenSHMEM implementation in heterogeneous environments. For conciseness, the analysis focuses exclusively on *put* operations; other communication primitives are outside the scope of this study.

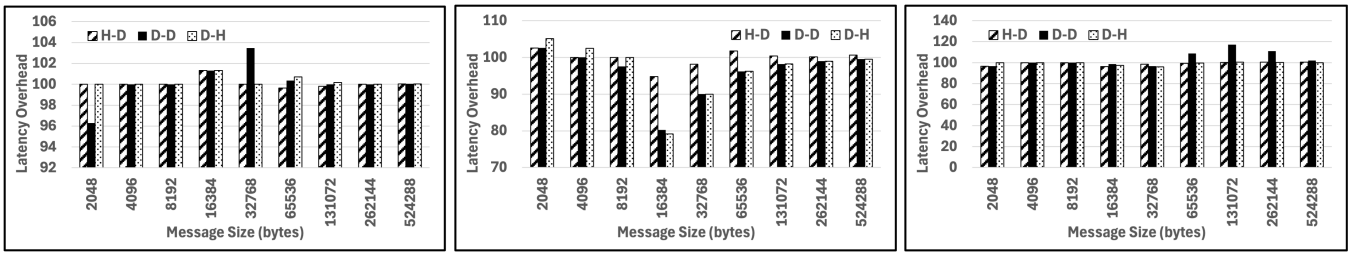
6.1 Test Bed Overview

The analysis is conducted across three compute node architectures (*Node Architectures 1–3*, see Figure 5), representative of systems such as Perlmutter (Architecture 1)[57], Frontier (Architecture 2)[8], and El Capitan (Architecture 3)[48]. Evaluations primarily involve inter-node data transfers, with one OpenSHMEM processing element (PE) mapped to a single GPU. Given the presence of multiple HPE Slingshot NICs per node, GPU–CPU–NIC affinity is optimized per architecture. The affinity is achieved manually during the job execution. In addition to the GPU-aware HPE Cray OpenSHMEMX implementation, architecture-specific vendor libraries—AMD ROCm[7] and NVIDIA CUDA [20]—serve as key dependencies.



(a) Node Architecture 3 (represents El Capitan) (b) Node Architecture 2 (represents Frontier) (c) Node Architecture 1 (represents Perlmutter)

Figure 10: Illustrating Latency Overhead based on the Source and Destination Buffer Types for Small (1B - 1KiB) Payloads



(a) Node Architecture 3 (represents El Capitan) (b) Node Architecture 2 (represents Frontier) (c) Node Architecture 1 (represents Perlmutter)

Figure 11: Illustrating Latency Overhead based on the Source and Destination Buffer Types for Large (2KiB - 512KiB) Payloads

6.2 Test Case Overview

Modified OSU *put* latency and bandwidth tests are employed for this analysis. The latency test measures round-trip latency for remote write operations across varying message sizes, capturing the cost of performing blocking *shmem_put* operations and ensuring delivery and global visibility at the target. The bandwidth test evaluates the sustained throughput of multiple non-blocking *shmem_put_nbi* operations.

The primary modification to the benchmarks involves the control over buffer allocation. While the original tests use CPU-attached symmetric memory for both source and destination buffers, the modified version enables selection between CPU- and GPU-attached memory. This yields four configurations: (1) **H-H** — both source and destination in CPU-attached memory; (2) **D-D** — both in GPU-attached memory; (3) **H-D** — CPU source to GPU destination; and (4) **D-H** — GPU source to CPU destination.

Performance results for these configurations are presented in Sections 6.3 and 6.4, with **H-H** serving as the baseline for comparison.

6.3 Latency Analysis

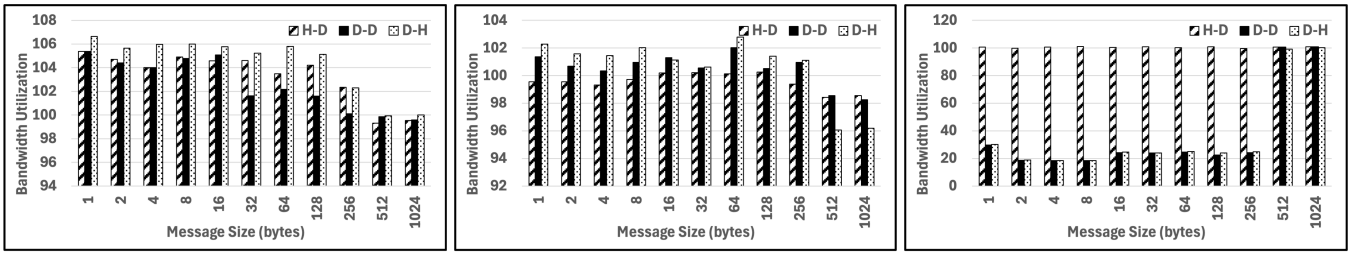
Figures 10 and 11 present the results of the latency analysis across different compute node configurations. The small-message RMA operation, specifically *shmem_put*, utilizes a network transfer protocol that enables packing the payload within the network packet. This protocol, also referred to as *INJECT* transfers, requires copying the data into the NIC command queue as part of the message header, to be consumed by the NIC. The INJECT protocol is generally considered effective for small message transfers, offering better performance compared to registering the source buffer with the NIC for RDMA-based data transfers.

However, with GPU-awareness, the cost of preparing the network packet using the INJECT protocol increases, as it involves copying data from the GPU-attached source buffer into the NIC command queue. The impact of this design is shown in Figure 10. For node architectures 1 and 2, which support better CPU-GPU connectivity (AMD MI250X connected to an AMD Trento processor via the AMD high-speed Infinity Fabric network, and AMD300A with a single memory domain as part of the APU architecture), the cost of the INJECT protocol is not significant. This results in the performance of D-D, D-H, and H-D being on par with the baseline H-H configuration, as depicted in Figures 10a and 10b. In contrast, for node architecture 3, where the connection between the AMD Milan CPU and the Nvidia A100 GPU is through PCIe links, the cost of updating the NIC command queue is significantly higher, resulting in an average latency overhead of 3X-5X.

Figure 11 demonstrates the impact of memory type on large message *put* latency. For these message sizes, there is minimal impact from the memory type of the source or destination buffer used for communication. Regardless of the memory type combinations, performance remains consistent with the H-H baseline. This analysis indicates that while small message sizes are influenced by the memory buffer type due to the communication protocol used, there are no significant issues with large message sizes. Tuning protocol selection based on the underlying compute node architecture can facilitate a portable implementation of the GPU-aware communication library, and the proposed OpenSHMEM API solutions are sufficient to address all necessary use cases.

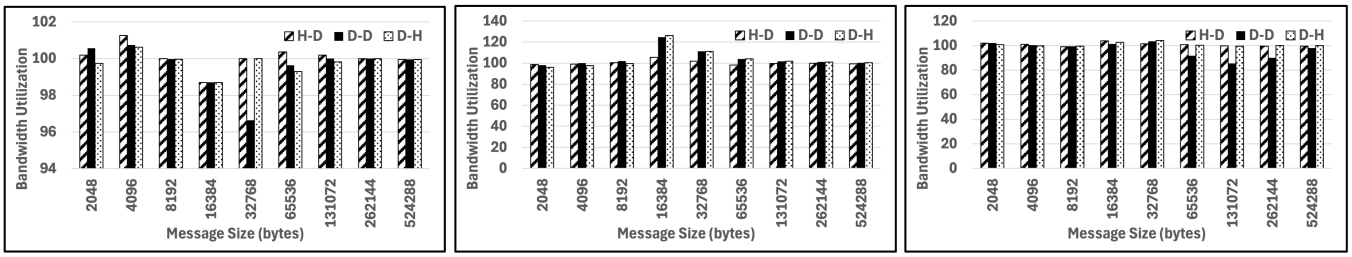
6.4 Bandwidth Analysis

This section discusses the impact of memory buffer type on effectively utilizing network bandwidth. Figure 13 illustrates the



(a) Node Architecture 3 (represents El Capitan) (b) Node Architecture 2 (represents Frontier) (c) Node Architecture 1 (represents Perlmutter)

Figure 12: Illustrating Bandwidth Utilization based on the Source and Destination Buffer Types for Small (1B - 1KiB) Payloads



(a) Node Architecture 3 (represents El Capitan) (b) Node Architecture 2 (represents Frontier) (c) Node Architecture 1 (represents Perlmutter)

Figure 13: Illustrating Bandwidth Utilization based on the Source and Destination Types for Small (2KiB - 512KiB) Payloads

influence of memory type across different compute node configurations in utilizing network bandwidth for data transfers. As shown, irrespective of the memory type used, it is possible to generate and utilize high-speed network bandwidth that approaches the theoretical maximum with large payloads. Figures 11a through 11c demonstrate this behavior across all three compute nodes used for the analysis.

For small message transfers, the impact of CPU-GPU network connectivity becomes apparent. As discussed in Section 6.3, the cost of staging the memory buffer as part of the INJECT data transfers can negatively impact performance, depending on the source buffer memory type. When using a GPU-attached source buffer, the overhead of the INJECT protocol may outweigh the primary benefits of the protocol itself.

Overall, both the latency and bandwidth analyses suggest that a standard one-sided communication semantics, based on the existing OpenSHMEM programming model, can support GPU-aware communication across different vendors. While minor performance issues are observed due to the underlying network architecture, it is possible to tune the implementation for a specific architecture. This validates the need for an effective and portable GPU-aware communication model in OpenSHMEM. Similar results were observed for other OpenSHMEM communication operations, such as *shmem_get* and atomic memory operations; however, for brevity, the results of these analyses are not included in this work.

7 Application Analysis

While Section 6 employed modified OSU microbenchmarks to analyze the performance behavior of the HPE Cray OpenSHMEMX implementation supporting the proposed OpenSHMEM GPU-aware communication features, this section examines the applicability of these features in supporting a proxy benchmark representing a critical communication pattern. Distributed bucket sorting [38] serves as a critical proxy kernel, representing a significant use case for the PGAS-based communication model. A version of the proxy benchmark designed for AMD MI300A-based compute node architecture (refer to *Node Architecture 3* in Figure 5) is used in this analysis. The benchmark is executed across 128 AMD MI300A processes, with 4 AMD MI300A processes per node. The details and results of the benchmark are discussed in Section 7.1.

7.1 Distributed Bucket Sorting

Recognized as a key data analytics kernel in data-driven workloads, bucket sorting plays a critical role in large-scale data processing. This section provides a brief overview of a traditional distributed bucket sorting algorithm and highlights the communication pattern it employs.

The distributed bucket sorting kernel uses a local bucket and a remotely shared table. Each process participating in the sort partitions its unsorted input table into chunks and performs a local sort. The locally sorted data is then loaded into the process's local bucket. At regular intervals, the sorted values in the local bucket are transferred to the appropriate remote tables based on their values. This data transfer mechanism is referred to as an *Append* communication scheme. In the append pattern, each process concurrently determines the offset in the remote table where it can perform updates. This involves executing a remote atomic fetch-and-add operation to compute and reserve the offset, followed by a remote write operation. All processes involved in the sort can perform append operations independently and concurrently across different targets, without requiring inter-process synchronization.

In this evaluation, multiple variants of the bucket sorting kernel are analyzed. A common aspect across all variants is that the local

sorting is performed on a GPU-attached memory buffer. The results of this analysis are shown in Figure 14.

HA_HI refers to the implementation where both the local bucket and the remote table reside in CPU-attached symmetric memory regions. In this model, the sorted data must be staged from the GPU-attached memory into the CPU-attached local bucket before initiating the data transfer. **HA_HI_OMP** extends this design by using multiple OpenMP [22] threads to perform data transfers after staging the data into the local bucket. Both **HA_HI** and **HA_HI_OMP** are designed to work with OpenSHMEM implementations that do not support GPU-awareness, relying on traditional staging methods.

GA_HI is the GPU-aware counterpart to **HA_HI**, where the local bucket is allocated in GPU-attached memory and used directly for communication. **GA_HI_Stream** is a variant of **GA_HI** that employs GPU streams [21] to improve utilization of GPU compute resources. **GA_HI_OMP** is a GPU-aware variant of **HA_HI_OMP**.

The observed performance results are shown in Figure 14, which presents both the lower-bound (LB) and upper-bound (UB) bandwidth observed per process. This test was conducted on 128 AMD MI300 GPUs, each running 4 PEs per node, with each PE utilizing an independent AMD MI300A APU for local sorting. As shown, **HA_HI** exhibits the lowest performance due to the overhead of staging data before communication. **HA_HI_OMP** demonstrates a significant improvement over **HA_HI**, benefiting from communication-compute overlap achieved via multithreading. However, it still suffers from limited LB performance due to uneven thread contributions. **GA_HI** shows improvements in both UB and LB metrics. Finally, combining the advantages of both **HA_HI_OMP** and **GA_HI**, the **GA_HI_OMP** and **GA_HI_Stream** variants yield the most effective bandwidth utilization across all participating processes, achieving strong performance in both LB and UB measures.

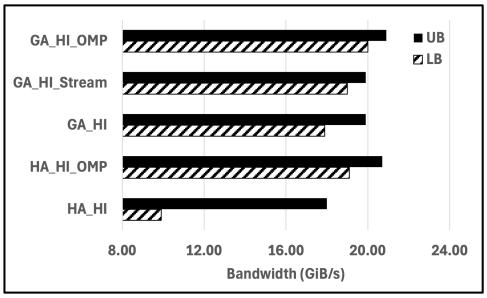


Figure 14: Variants of Distributed Bucket Sorting

Overall, analyzing the distributed bucket sorting kernel using the GPU-aware OpenSHMEM implementation demonstrates the potential of the proposed solution for effectively supporting key data analytics workloads. While the proposed extensions enable application developers to explore the use of OpenSHMEM for heterogeneous systems, several key limitations have been identified. These limitations, along with the motivation for developing advanced GPU-centric communication solutions, are discussed in Section 8.

8 Advanced GPU-Centric Communication

This section describes the limitation of the proposed GPU-aware communication operations, and discuss on the need for extending the proposed solution to support advanced GPU-based communication solutions as future work.

8.1 Improve GPU Autonomy

The proposed solution in Section 4 introduces GPU-aware communication within the OpenSHMEM programming model. This design relies on the CPU to manage the control path of communication operations, requiring CPU threads to orchestrate the data movement and inter-process synchronization operations. As demonstrated in Section 6, the approach achieves performance comparable to existing state-of-the-art solutions involving CPU-attached memory buffers for communication. However, its applicability is limited in scenarios where CPU involvement introduces inefficiencies.

For instance, in our evaluation using GPU-aware distributed bucket sorting (refer Section 7), communication must occur at kernel boundaries to allow the CPU to manage control path of the communication operations. This constraint necessasitates unnecessary CPU-GPU synchronization. Enhancing GPU autonomy by extending the proposed communication scheme—specifically by introducing features to allow the GPU to manage the control path—would enable more flexible and efficient communication schemes and broaden the applicability of the proposed solution.

8.2 Communication and Computation Overlap

Several potential design approaches can improve the overlap between computation and communication using the proposed GPU-aware solution in OpenSHMEM. For example, as demonstrated in Section 7, at least two such strategies—**GA_HI_Stream** and **GA_HI_OMP**—leverage multiple CPU threads in conjunction with GPU streams to achieve this overlap. In these approaches, the GPU executes compute operations, while CPU threads manage the communication operations concurrently. These optimizations rely on effective interaction with GPU streams, which is currently limited in the proposed solution. Enhancing this interaction—specifically between the GPU streams used for compute and the associated communication operations—could lead to meaningful extensions to the solution introduced in Section 4.

8.3 Support Fine-Grained Communication

Another major limitation of the existing proposal, as observed in the application study from Section 7, is the coarse granularity of communication operations. In this example, communication is performed at kernel boundaries, resulting in large message sizes. Consequently, data movement operations across all target processes must wait for the complete availability of the data to be communicated. Enabling communication operations from within a GPU kernel would allow OpenSHMEM users to explore the potential performance benefits of a fine-grained communication model. Supporting such a use case requires extending the proposed solution from Section 4.

To enable and satisfy the above requirements, new advanced GPU-centric communication strategies are required to be introduced into the OpenSHMEM programming model—effectively creating a need to extend the proposed solution. There are multiple such strategies already available in different libraries like NVSHMEM [41], ROC_SHMEM [36], and MPI [53, 54, 68, 84]. In brief, these strategies include: (1) *GPU Stream-Triggered Communication*, where GPU streams manage the control path of the communication operation; (2) *GPU Kernel-Triggered Communication*, where GPU threads manage the control path; and (3) *GPU Kernel-Initiated Communication*, where GPU-aware communication operations are executed directly from within a GPU compute kernel.

For brevity, detailed descriptions of these GPU-centric communication strategies are not discussed in this work. However, this section acknowledges their existence and emphasizes the need to evaluate and incorporate them into the OpenSHMEM specification by extending the proposed solution from Section 4.

9 Related Work

Several efforts have introduced GPU-awareness into standard HPC programming models. The work in [80] extends the MVAPICH MPI implementation to support efficient GPU-aware MPI communication on InfiniBand clusters. As one of the earliest contributions in this space, it identifies key features required for enabling GPU-aware inter-process communication. While [80] focuses on supporting GPU-aware point-to-point communication, subsequent studies such as [75, 83] and [28] address the requirements for enabling GPU-aware non-contiguous and collective communication operations in the MPI [29] programming model. Although these works primarily target systems with NVIDIA GPUs, [45] investigates GPU-aware communication on AMD-based architectures, highlighting the portability and adaptability of these concepts.

In addition to programming models, GPU-awareness has also been explored at the communication middleware level. For instance, [18] identifies the necessary modifications to the UCX [71] framework to enable GPU-aware communication. Other studies such as [42] and [70] evaluate the usability and performance of GPU-aware operations in real-world applications.

Specifically for OpenSHMEM, [64] proposes an extension to support GPU-aware communication. Although this work represents an early step toward enabling GPU-awareness in OpenSHMEM, it does not address standardization or portability across heterogeneous platforms. Furthermore, it does not explore implementation strategies to avoid staging through CPU-attached buffers. Later efforts, such as [65] and [66], introduce optimizations for zero-copy data transfers, enabling direct GPU memory access and improving the efficiency of GPU-aware OpenSHMEM implementations. This work builds on those efforts by evaluating both the portability and implementation efficiency of GPU-aware extensions to the OpenSHMEM programming model on modern state-of-the-art exascale-based compute node architectures.

10 Conclusion

Overall, this work introduced device-agnostic support for basic GPU awareness within the OpenSHMEM programming model and evaluates the performance of the proposed features across diverse heterogeneous system architectures. It details the design and implementation of this capability to advocate for standardizing GPU-awareness in the OpenSHMEM specification.

In addition to enabling fundamental GPU-awareness, this work investigates the necessity of supporting advanced GPU-centric communication mechanisms in OpenSHMEM. This is achieved by analyzing the communication patterns and requirements of representative data-driven workloads, thereby identifying opportunities for enhancing GPU autonomy through features such as stream-triggered and kernel-initiated communication.

By extending the OpenSHMEM programming model with support for GPU-awareness, the proposed work empowers existing OpenSHMEM users to efficiently leverage emerging heterogeneous platforms for data-driven applications. Furthermore, the performance insights and implementation strategies presented can inform and benefit other programming models—such as MPI-RMA—by providing a foundation for adopting optimized communication techniques tailored for GPU-enabled systems.

References

[1] 2002. GASNet: A Portable High-Performance Communication Layer for Global Address-Space Languages. http://gasnet.lbl.gov/pubs/258-paper.pdf.

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

[3] Thomas Alrutz, Jan Backhaus, Thomas Brandes, Vanessa End, Thomas Gerhold, Alfred Geiger, Daniel Grünewald, Vincent Heuveline, Jens Jägersküpper, Andreas Knüpfer, Olaf Krzikalla, Edmund Kügeler, Carsten Lojewski, Guy Lonsdale, Ralph Müller-Pfefferkorn, and Wolfgang Nagel et al. 2013. GASPI – A Partitioned Global Address Space Programming Interface. In *Facing the Multicore-Challenge III (Lecture Notes in Computer Science)*.

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

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

[6] AMD. 2021. AMD Instinct® MI250X Accelerator. https://www.amd.com/en/products/accelerators/instinct/mi200/mi250x Accessed: 2025-05-01.

[7] AMD. 2025. ROCm: Radeon Open Compute. https://www.amd.com/en/products/software/rocm Accessed: 2025-05-01.

[8] 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. 2023. Frontier: Exploring Exascale. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*.

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

[10] David H. Bailey, Eric Barszcz, John T. Barton, David S. Browning, Richard L. Carter, Lawrence Dagum, R. A. Fatoohi, P. O. Frederickson, Thomas A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. 1991. The NAS Parallel Benchmarks. *International Journal of Supercomputer Applications*. Accessed: 2025-05-01.

[11] Maciej Besta, Florian Scheidl, Lukas Gianinazzi, Shachar Klaiman, Jürgen Müller, and Torsten Hoefler. 2024. Demystifying Higher-Order Graph Neural Networks. *arXiv preprint arXiv:2406.12841* (2024). https://arxiv.org/abs/2406.12841 Accessed: 2025-05-01.

[12] Alex Brooks, Philip Marshall, David Ozog, Md. Wasi ur Rahman, Lawrence Stewart, and Rithwik Tom. 2024. Intel® SHMEM: GPU-initiated OpenSHMEM using SYCL. *arXiv* 2409.20476 (2024). https://arxiv.org/abs/2409.20476

[13] D. Callahan, B.L. Chamberlain, and H.P. Zima. 2004. The cascade high productivity language. In *Ninth International Workshop on High-Level Parallel Programming Models and Supportive Environments, 2004. Proceedings.*

[14] William W. Carlson, Jesse M. Draper, and David E. Culler. 1996. S-246, 187 Introduction to UPC and Language Specification.

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

[16] Brad Chamberlain, David Callahan, and Michael Ferguson. 2015. Performance of Chapel on the ISx Benchmark. https://chapel-lang.org/perf-isx.html Accessed: 2025-05-01.

[17] Barbara Chapman, Tony Curtis, Swaroop Pophale, Stephen Poole, Jeff Kuehn, Chuck Koelbel, and Lauren Smith. 2010. Introducing OpenSHMEM: SHMEM for the PGAS Community. In *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model (PGAS '10)*.

[18] Jaemin Choi, Zane Fink, Sam White, Nitin Bhat, David F. Richards, and Laxmikant V. Kale. 2021. GPU-aware Communication with UCX in Parallel Programming Models: Charm++, MPI, and Python. In *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*.

[19] M. A. Clark, Bálint Joó, Alexei Strelchenko, Michael Cheng, Arjun Gambhir, and Richard Brower. 2016. Accelerating Lattice QCD Multigrid on GPUs Using Fine-Grained Parallelization. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*.

[20] NVIDIA Corporation. 2025. CUDA C Programming Guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html Accessed: 2025-05-01.

[21] NVIDIA Corporation. 2025. CUDA Streams. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#streams Accessed: 2025-05-01.

[22] Leonardo Dagum and Ramesh Menon. 1998. OpenMP: an industry standard API for shared-memory programming. *Computational Science & Engineering, IEEE*.

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

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

[25] James Dinan and Mario Flajslik. 2014. Contexts: A Mechanism for High Throughput Communication in OpenSHMEM. In *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models (PGAS 2014)*. Accessed: 2025-05-01.

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

[27] G. Faanes, A. Bataineh, B. Bergen, S. Blond, G. Bosilca, D. Chen, and et al. 2012. Cray Aries Interconnect. In *Proceedings of the Cray User Group (CUG) Conference*. Cray Inc. https://cug.org/proceedings/cug2012_proceedings/includes/pages/papers/pap117.pdf Accessed: 2025-05-01.

[28] Iman Faraji and Ahmad Afsahi. 2018. Design considerations for GPU-aware collective communications in MPI. *Concurrency and Computation: Practice and Experience*.

[29] Message Passing Forum. 1994. *MPI: A Message-Passing Interface Standard*. Technical Report.

[30] Heterogeneous System Architecture (HSA) Foundation. 2025. HSA Platform System Architecture Specification 1.2. https://hsafoundation.com/standards/ Accessed: 2025-05-01.

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

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

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

[34] Evangelos Georganas, Marquita Ellis, Rob Egan, Steven Hofmeyr, Aydin Buluç, Brandon Cook, Leonid Oliker, and Katherine Yelick. 2017. MerBench: PGAS Benchmarks for High Performance Genome Assembly. In *Proceedings of the Second Annual PGAS Applications Workshop (PAW17)*.

[35] Steven Gottlieb and Sonali Tamhankar. 2001. Benchmarking MILC code with OpenMP and MPI. *Nuclear Physics B - Proceedings Supplements*.

[36] Khaled Hamidouche and Michael LeBeane. 2020. *GPU INitiated OPenSHMEM: Correct and Efficient Intra-Kernel Networking for DGPUs*. Association for Computing Machinery.

[37] Khaled Hamidouche and Michael LeBeane. 2020. 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)*.

[38] Jacob Hemstad, Ulf R. Hanebutte, Ben Harshbarger, and Bradford L. Chamberlain. 2016. A Study of the Bucket-Exchange Pattern in the PGAS Model Using the ISx Integer Sort Mini-Application. In *Proceedings of the PGAS Applications Workshop (PAW) at SC16*. Accessed: 2025-05-01.

[39] Torsten Hoefler, James Dinan, Rajeev Thakur, Brian Barrett, Pavan Balaji, William Gropp, and Keith Underwood. 2015. Remote Memory Access Programming in MPI-3. *ACM Trans. Parallel Comput.* (2015).

[40] 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. 2020. Terabase-scale metagenome coassembly with MetaHipMer. *Scientific Reports*.

[41] Chung-Hsing Hsu, Neena Imam, Akhil Langer, Sreeram Potluri, and Chris J. Newburn. 2020. An Initial Assessment of NVSHMEM for High Performance Computing. In *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. https://doi.org/10.1109/IPDPSW50202.2020.00104

[42] James Buford White III. 2023. Performance Portability of Programming Strategies for Nearest-Neighbor Communication with GPU-Aware MPI. In *Proceedings of the SC '23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*.

[43] InfiniBand Trade Association. 2001. InfiniBand Architecture Specifications Release 1.0.a. http://www.infinibandta.org. Accessed: 2025-05-01.

[44] Mathias Jacquelin, Yili Zheng, Esmond Ng, and Katherine Yelick. 2016. An Asynchronous Task-based Fan-Both Sparse Cholesky Solver. https://doi.org/10.48550/arXiv.1608.00044

[45] Kawthar Shafie Khorassani, Jahanzeb Hashmi, Ching-Hsiang Chu, Chen-Chun Chen, Hari Subramoni, and Dhabaleswar K. Panda. 2021. Designing a ROCm-Aware MPI Library for AMD GPUs: Early Experiences. In *ISC High Performance 2021*.

[46] Jeongnim Kim, Andrew D Baczewski, Todd D Beaudet, Anouar Benali, M Chandler Bennett, Mark A Berrill, Nick S Blunt, Edgar Josué Landinez Borda, Michele Casula, David M Ceperley, Simone Chiesa, Bryan K Clark, Raymond C Clay, Kris T Delaney, Mark Dewing, Kenneth P Esler, Hongxia Hao, Olle Heinonen, Paul R C Kent, Jaron T 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, Eric Neuscamman, William D Parker, Sergio D Pineda Flores, Nichols A Romero, Brenda M Rubenstein, Jacqueline A R Shea, Hyeondeok Shin, Luke Shulenburger, Andreas F Tillack, Joshua P Townsend, Norm M Tubman, Brett Van Der Goetz, Jordan E Vincent, D ChangMo Yang, Yubo Yang, Shuai Zhang, and Luning Zhao. 2018. QMCPACK: an open source ab initio quantum Monte Carlo package for the electronic structure of atoms, molecules and solids. *Journal of Physics: Condensed Matter*.

[47] Shigang Li, Torsten Hoefler, Chungjin Hu, and Marc Snir. 2014. Improved MPI collectives for MPI processes in shared address spaces. *Cluster Computing*.

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

[49] 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. 2019. CosmoFlow: using deep learning to learn the universe at scale. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*.

[50] S. Milaković, Z. Budimlić, H. Pritchard, A. Curtis, B. Chapman, and V. Sarkar. 2019. SHCOLL – A Standalone Implementation of OpenSHMEM-Style Collectives API. In *OpenSHMEM and Related Technologies. OpenSHMEM in the Era of Extreme Heterogeneity*.

[51] Naveen Namashivayam, Bob Cernohous, Krishna Kandalla, Dan Pou, Joseph Robichaux, James Dinan, and Mark Pagel. 2018. Symmetric Memory Partitions in OpenSHMEM: A Case Study with Intel KNL. In *OpenSHMEM and Related Technologies. Big Compute and Big Data Convergence*. Accessed: 2025-05-01.

[52] Naveen Namashivayam, Bob Cernohous, Dan Pou, and Mark Pagel. 2018. Introducing Cray OpenSHMEMX - A Modular Multi-Communication Layer OpenSHMEM Implementation. In *OpenSHMEM 2018: Fifth Workshop on OpenSHMEM and Related Technologies*.

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

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

[55] Naveen Namashivayam, David Knaak, Bob Cernohous, Nick Radcliffe, and Mark Pagel. 2016. An Evaluation of Thread-Safe and Contexts-Domains Features in Cray SHMEM. In *OpenSHMEM and Related Technologies. Enhancing OpenSHMEM for Hybrid Environments*. Accessed: 2025-05-01.

[56] Naveen Namashivayam, Bill Long, Deepak Eachempati, Bob Cernohous, and Mark Pagel. 2020. A Modern Fortran Interface in OpenSHMEM: Need for Interoperability with Parallel Fortran Using Coarrays. *ACM Transactions on Parallel Computing* (2020).

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

[58] Robert W. Numrich and John Reid. 1998. Co-array Fortran for Parallel Programming. *SIGPLAN Fortran Forum* 17, 2 (Aug. 1998).

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

[60] NVIDIA. 2024. CUDA Inter-Process Communication (IPC). https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html Accessed: 2025-05-01.

[61] NVIDIA. 2025. GPUDirect. https://developer.nvidia.com/gpudirect Accessed: 2025-05-01.

[62] Bruce Palmer, Manojkumar Krishan, and Abhinav Vishnu. 2011. Parallel programming using the global arrays toolkit: now and into the future. In *2011 IEEE International Parallel & Distributed Processing Symposium*.

[63] Swaroop Pophale, Ramachandra Nanjegowda, Anthony R. Curtis, Barbara Chapman, Haoqiang Jin, Stephen W. Poole, and Jeffery A. Kuehn. 2012. OpenSHMEM Performance and Potential: A NPB Experimental Study. In *Proceedings of the 2012 International Conference on Partitioned Global Address Space Programming Models (PGAS 2012)*.

[64] Sreeram Potluri, Devendar Bureddy, Huamin Wang, Hari Subramoni, and Dhabaleswar K. Panda. 2013. Extending OpenSHMEM for GPU Computing. In *Proceedings of the 27th IEEE International Symposium on Parallel & Distributed Processing (IPDPS 2013)*.

[65] Sreeram Potluri, Devendar Bureddy, Huamin Wang, Hari Subramoni, and Dhabaleswar K. Panda. 2016. CUDA-Aware OpenSHMEM: Extensions and Designs for High Performance OpenSHMEM on GPU Clusters. *Parallel Comput.*

[66] Mohammad Aditya Sasongko, Ilyas Turimbetov, and Didem Unat. 2016. Exploiting GPUDirect RDMA in Designing High Performance OpenSHMEM for NVIDIA GPU Clusters. In *Proceedings of the 2016 International Conference on Parallel Processing (ICPP 2016)*.

[67] David Schneider. 2022. 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).

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

[69] 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. 2024. Exploring GPU-to-GPU Communication: Insights into Supercomputer Interconnects. arXiv:2408.14090

[70] Aamir Shafi, Jahanzeb Maqbool Hashmi, Hari Subramoni, and Dhabaleswar K. Panda. 2021. Efficient MPI-based Communication for GPU-Accelerated Dask Applications. *arXiv preprint arXiv:2101.08878*. https://arxiv.org/abs/2101.08878

[71] Pavel Shamis, Manjunath Gorentla Venkata, M. Graham Lopez, Matthew B. Baker, Oscar Hernandez, Yossi Itigin, Mike Dubman, Gilad Shainer, Richard L. Graham, Liran Liss, Sameer Kumar, Craig Stunkel, George Bosilca, and Aurelien Bouteiller. 2015. UCX: An Open Source Framework for HPC Network APIs and Beyond. In *Proceedings of the 2015 IEEE 23rd Annual Symposium on High-Performance Interconnects (HOTI)*.

[72] Hongzhang Shan, Samuel Williams, Yili Zheng, Amir Kamil, and Katherine Yelick. 2015. 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)*.

[73] Sharad Singhal, Clarete R. Crasta, Mashood Abdulla, Faizan Barmawer, Dave Emberson, Ramya Ahobala, Gautham Bhat, Rishi kesh K. Rajak, and P. N. Soumya. 2021. OpenFAM: A Library for Programming Disaggregated Memory. In *OpenSHMEM and Related Technologies. OpenSHMEM in the Era of Exascale and Smart Networks: 8th Workshop on OpenSHMEM and Related Technologies*.

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

[75] Kaushik Kandadi Suresh, Kawthar Shafie Khorassani, Chen Chun Chen, Bharath Ramesh, Mustafa Abduljabbar, Aamir Shafi, Hari Subramoni, and Dhabaleswar K. Panda. 2023. Network-Assisted Noncontiguous Transfers for GPU-Aware MPI Libraries. *IEEE Micro*.

[76] Thomas Talpey and Cain Recio. 2003. RDMA: Remote Direct Memory Access. In *Proceedings of the IEEE Cluster Computing Conference*. IEEE. Accessed: 2025-05-01.

[77] Rajeev Thakur and William D. Gropp. 2003. Improving the Performance of Collective Operations in MPICH. In *Proceedings of the 2003 ACM/IEEE Conference on Supercomputing (SC '03)*.

[78] 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. 2022. LAMMPS - a flexible simulation tool for particle-based materials modeling at the atomic, meso, and continuum scales. *Comp. Phys. Comm.*

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

[80] Hao Wang, Sreeram Potluri, Miao Luo, Ashish Kumar Singh, Sayantan Sur, and Dhabaleswar K. Panda. 2011. MVAPICH2-GPU: Optimized GPU to GPU Communication for InfiniBand Clusters. *Computer Science - Research and Development*.

[81] Aaron Welch, Swaroop Pophale, Pavel Shamis, Oscar Hernandez, Stephen Poole, and Barbara Chapman. 2014. Extending the OpenSHMEM Memory Model to Support User-Defined Spaces. In *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models (PGAS 2014)*. Accessed: 2025-05-01.

[82] Thomas Worsch, Ralf Reussner, and Werner Augustin. 2002. On Benchmarking Collective MPI Operations. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*.

[83] Wei Wu, George Bosilca, Rolf vandeVaart, Sylvain Jeaugey, and Jack Dongarra. 2016. GPU-Aware Non-contiguous Data Movement in Open MPI. In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing (HPDC '16)*.

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