2024 IEEE 31st International Conference on High Performance Computing, Data, and Analytics (HiPC) | 979-8-3315-0909-5/24/$31.00 ©2024 IEEE | DOI: 10.1109/HIPC62374.2024.00022

Design and Implementation of Kernel-based MPI Reduction Operations for Intel GPUs

Chen-Chun Chen
The Ohio State University
Columbus, Ohio, USA
chen.10252@osu.edu

Goutham Kalikrishna Reddy Kuncham
The Ohio State University
Columbus, Ohio, USA
kuncham.2@osu.edu

Hari Subramoni
The Ohio State University
Columbus, Ohio, USA
subramoni.1@osu.edu

Dhabaleswar K. Panda
The Ohio State University
Columbus, Ohio, USA
panda@cse.ohio-state.edu

***Abstract*—The demand for computing power in high-performance computing and deep learning applications is steadily increasing, leading to a noticeable inclination toward equipping modern exascale clusters with accelerators. In particular, distributed Deep Learning training necessitates high-performance GPU-aware MPI operations, with reduction operations being widely employed. Unlike data movement-based MPI runtimes, reduction operations encompass both communication and computation, making them inherently more intricate to design and optimize for data transmission between GPU buffers. Acknowledging the success of NVIDIA and AMD GPUs in HPC, Intel has actively participated in the development of GPU products, while also fostering their associated ecosystems in recent years. However, existing MPI libraries supporting Intel GPUs rely on naive staging approaches, resulting in elevated latencies and subpar performance. In this paper, we propose a kernel-based reduction collective MPI library designed specifically for Intel GPUs. Our approach leverages IPC techniques to minimize data movement overhead during communication while harnessing highly efficient GPU kernels for the computational aspects of reduction operations. We assess the advantages of our designs through benchmark and application-level evaluations, conducted on ACES and Stampede3 systems. In benchmark-level evaluations, our Allreduce implementations demonstrate an 13.3x performance enhancement compared to Intel MPI at 1GB with 8 GPUs. Moreover, with 32 GPUs, we achieve a 42% performance enhancement. In application-level evaluations, our proposed designs exhibit up to a 22% enhancement for the Deep Learning application TensorFlow with Horovod and a 28% improvement for PyTorch with Horovod on 32 GPUs compared to Intel MPI.**

***Index Terms*—Intel GPUs, IPC, Kernel, Reduction, Allreduce**

I. Introduction

The high demand for computing power and communication throughput from scientific, big data, and deep learning (DL) applications drives the development of a high-performance computing (HPC) ecosystem. As accelerators, such as Graphics Processing Units (GPUs), boast remarkable computational capabilities, vendors like NVIDIA, AMD, and Intel showcase their GPU products, influencing the TOP500 [1] ranking. For instance, in 2023, the top-ranked Frontier system utilizes AMD Instinct 250X GPUs, while the second-ranked Aurora system is powered by Intel Data Center GPU Max Series. Therefore, these advanced dense GPU systems depend on communication runtimes for effective workload scalability. Message Passing Interface (MPI) serves as the standard communication paradigm in HPC systems, facilitating communication across processes, while GPU-aware MPI libraries [2, 3] facilitate data transmission between GPU buffers managed by different processes.

With most MPI runtimes prioritizing data movement between processes, the primary overhead is attributed to communication. However, certain MPI operations entail additional computation, such as MPI_Scan and reduction operations like MPI_Allreduce. For example, in reduction operations utilizing the MAX operation, calculating the maximum value in the buffers among all processes necessitates additional CPU computing power and cycles. As more HPC applications offload their computation to accelerators [4] and distributed DL applications proliferate, the development of efficient and productive reduction runtimes for data on GPU buffers becomes indispensable for GPU-aware MPI libraries. In modern dense GPU systems, which feature multiple accelerators within a node, high-performance interconnectivity such as NVLink and NVSwitch for NVIDIA GPUs is crucial for achieving high throughput communication. Moreover, with GPUs becoming increasingly powerful in computation, developers are inspired to leverage these cutting-edge capabilities when designing efficient MPI reduction operations for GPUs.

A. Motivation

Following in the footsteps of NVIDIA and AMD, Intel is actively designing and developing a range of GPU products and their accompanying ecosystems. In 2020, Intel introduced the Iris X^e Max Graphics, followed by the release of the Intel Data Center GPU Max Series tailored for HPC and AI workloads. Additionally, it also provides Intel X^e links, enabling high-performance interconnectivity among its GPUs within a node. On the software front, application developers port their implementations to support GPU kernels in leading programming models like SYCL [5], OpenMP, and Kokkos to harness the computing power of Intel GPUs. To enhance scalability, applications necessitate communication runtimes with low latency and high throughput for GPUs. For instance, the HPC application heFFTe [6] utilizes GPU-aware MPI point-to-point operations or collective Alltoall(v) communication

This research is supported in part by NSF grants #1818253, #1854828, #2018627, #2311830, #2312927, and XRAC grant #NCR-130002. We would also like to thank the Texas Advanced Computing Center and Texas A&M University for providing access to the HPC systems used in this research.

patterns to optimize data movement among device buffers. In DL applications, plugins [7, 8] for TensorFlow and PyTorch enable efficient AI workloads for intel GPUs, while distributed frameworks [9] streamline multi-GPU training processes, emphasizing the communication of reduction operations such as Allreduce and Reduce_scatter.

Despite the significant demand, particularly for the Allreduce operation in DL applications, there is currently no optimized MPI library providing efficient and high-performance design and implementation for Intel GPUs. As the vendor, Intel has recently developed their GPU-aware Intel MPI [10]. However, due to its closed-source nature, the optimizations utilized are not transparent, and the performance for reduction operations remains low. Furthermore, MPICH [11] employed CPU staging approaches for all reduction operations, resulting in even higher latency, particularly with large messages. Therefore, there is significant room for enhancing the performance of MPI reduction operations on the latest Intel GPUs to improve throughput for both HPC and DL applications.

B. Proposed Solution

We optimize the reduction operations from two primary perspectives: computation and communication. We leverage GPU inter-process communication (IPC) techniques to optimize the communication of data movement between GPUs within a node, and we design and implement GPU kernels for reduction computation using the Intel oneAPI Base Toolkits [12] with a SYCL backend. We also employ a hybrid design to mitigate overhead from IPC utilization and kernel launching, ensuring good performance across all message sizes. In this paper, our primary focus lies in optimizing the Allreduce operation, as it introduces significant communication traffic compared to other reduction operations and is widely utilized. We devise a basic unified reduction kernel to support all reduction operations. However, for the intra-node Allreduce operation, we implement an optimized kernel. This optimization is necessary as it involves a broadcast communication step right after the regular kernel reduction. Moreover, we adopt a two-level algorithm to deal with the inter-node Allreduce communication. **To the best of our knowledge, our proposed design stands as the pioneer in leveraging GPU kernels for MPI reduction operations on Intel GPUs, surpassing the performance of both Intel MPI and MPICH libraries.**

C. Challenges

We address the following challenges to design and implement a hybrid kernel-based MPI library for the reduction based collective communication:

- How can we best strategize and which techniques should we adopt to develop a high-performance MPI library specifically tailored for reduction collective operations on Intel GPUs?
- How can we design a library to facilitate computation and communication among GPUs within a node, capitalizing on the high bandwidth of Intel X^e links?
- How can we effectively design and implement a comprehensive GPU kernel for Intel GPUs using the oneAPI and SYCL library?
- How can we further optimize the ubiquitous Allreduce operation in intra-node and inter-node environments?

D. Contributions

This paper makes the following contributions:

- We proposed, designed, and implemented kernel-based MPI reduction operations for Intel GPUs using SYCL and oneAPI Toolkits (Section III)
- Our designs support common MPI operations/datatypes on GPUs (Section III-C and IV-B, Figure 3 and 4)
- We designed a fundamental kernel to encompass all reduction-based operations, including Allreduce, Reduce, Reduce_scatter, and Reduce_scatter_block (Section III-C and IV-E, Figure 8)
- We optimized the MPI_Allreduce operation for inter-node and intra-node scenarios (Section III-C and III-D)
- We conducted profiling of our Allreduce design and provided analysis for both inter-node and intra-node execution. (Section IV-C and Figure 5)
- Our designs offer up to 13.3 times improvement compared to Intel MPI in osu_allreduce benchmark on 8 GPUs, and offer up to 42% improvement on 32 GPUs (Section IV-D and Figure 7)
- In real application-level evaluations, our designs demonstrate up to a 22% improvement compared to Intel MPI in Horovod with TensorFlow and a 28% improvement in Horovod with PyTorch (Section IV-F, Figure 9 and 10).

II. Background

A. Intel GPUs

Intel has recently entered the discrete graphics card market and has introduced the Max Series product family, specifically designed for the demanding worlds of HPC and AI. This product lineup features two powerhouses: the Intel Xeon CPU Max Series (codenamed Sapphire Rapids HBM) and the Intel Data Center GPU Max Series (codenamed Ponte Vecchio). The Max Series GPU is Intel's highest-density processor, packing over 100 billion transistors into a 47-tile package with up to 128 X^e HPC cores and up to 128 GB of high bandwidth memory. To enable scale-up and scale-out capabilities, the X^e-HPC micro-architecture incorporates the advanced X^e Link technology. X^e Link is a high-speed interconnect solution designed specifically to connect multiple X^e-HPC GPUs in various configurations, such as 2-way, 4-way, 6-way, and even 8-way. Each X^e Link is capable of up to 26.5 GB/s of bandwidth in each direction [13].

B. Reduction Operations

Reduction procedures are critical in parallel computing because they allow the aggregation of data across numerous processes to produce a consolidated result. Among these operations, MPI_Allreduce stands out as a basic collective communication primitive, in which each process contributes its local data and the operation results in all processes receiving the combined result [14]. MPI_Reduce is another collective that offers similar functionality, but with a singular focus: aggregating data solely at a designated ”root” process. This makes it ideal when the final result is only required by one process. Expanding on the concept, MPI_Reduce_scatter is a variant of reduction operations that involve distributing resultant data among processes after performing a reduction operation. Unlike other collective communications, reduction operations involve computations, adding a layer of complexity and resource consumption. As reduction operations involve computational processing, there exists a demand for continuous research efforts focused on enhancing their efficiency.

C. Inter-process Communication (IPC)

Parallel computing applications rely heavily on efficient communication mechanisms to facilitate data exchange and synchronization between processes. These processes operate within distinct address spaces, preventing direct access to device memory or events created by threads in other processes. Similar communication issues exist within a single compute node when dealing with multiple GPUs. To address this, GPU Inter-Process Communication (GPU IPC) emerges as a key technique for optimizing data movement between GPU processes. GPU IPC enables direct data transfer between the GPU memory spaces of distinct processes, bypassing the host CPU entirely. This optimization relies on a process exposing a designated portion of its GPU memory to remote processes. This exposure is achieved by creating a memory handle, which acts as a unique reference point for the shared memory region. The memory handle is then transferred to the remote process, granting it access to the shared memory. Upon receiving the handle, the remote process can read and potentially modify the shared data residing in the remote GPU’s memory space.

D. Kernel-based Computation

Kernel-based computing has emerged as a key paradigm in modern computational research, providing good parallelism and processing capacity. Traditional CPU kernels, or small programs within the operating system that manage system resources, are limited by the CPU’s architecture and security features. In contrast, GPU kernels take advantage of modern GPUs’ massive parallel processing capabilities to perform computationally complex jobs with amazing efficiency. However, optimizing GPU kernels necessitates a distinct methodology that takes into account characteristics such as memory access patterns and workload allocation across several cores. Despite these obstacles, GPU kernels have spread throughout the scientific, engineering, and machine learning fields, propelling advances in computational research and allowing for the study of complicated issues on bigger scales.

III. Design

A. Overview of the Designs

GPU and kernel-based computation offer high performance for parallel computing and vectorized data processing. However, there is a notable overhead during initialization, including IPC handle exchange and kernel launching, making it inefficient for small message runtimes. Therefore, we adopt a hybrid approach to complement our strategy, aiming to achieve low latency across all message ranges. For small messages, we utilize CPU staging approaches to prevent the initialization overheads associated with kernel-based designs. CPU staging approaches use CPU-based MPI runtimes for communication and reduction. This process requires moving data from GPU memory to a temporary CPU buffer before the operation, and after the CPU-based communication is completed, the data is stored back in the device. The memory copying between the device and host buffer is efficient for small messages, but as the message size increases, latency grows significantly. Hence, this is where we switch to kernel-based designs on top of IPC techniques for large message communication.

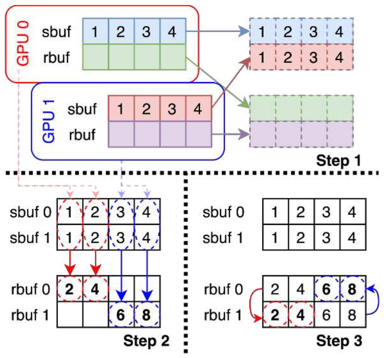
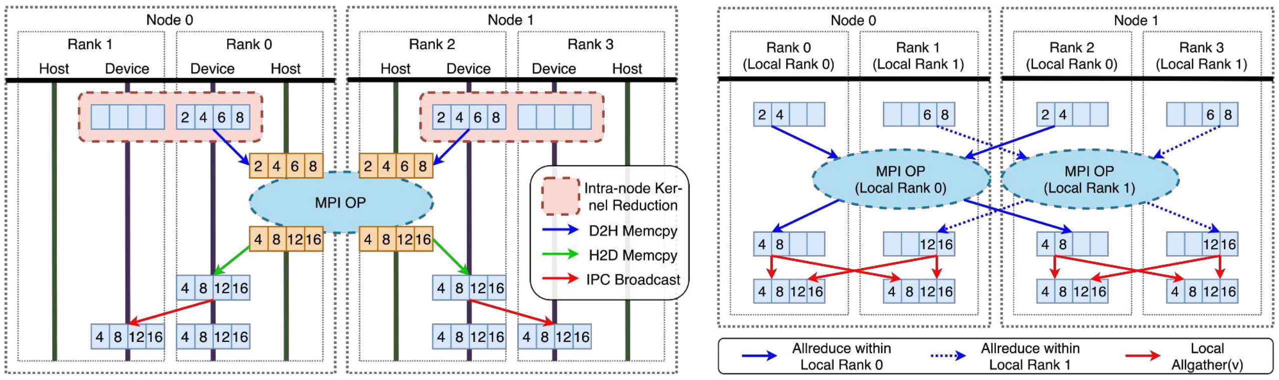


Fig. 1. Overview of the proposed intra-node designs for MPI_Allreduce.

While most GPU MPI operations primarily entail data movement between GPUs, reduction operations encompass both communication and computation. Hence, the overheads from both parts escalate significantly as the message size increases. To mitigate the communication overhead between device buffers, we employ IPC techniques, which enable direct data access between the GPU address spaces of different processes via Intel X^e Links. On the other hand, to enhance computation efficiency, we implement GPU kernels to handle the heavy reduction operations. We utilize Intel oneAPI to develop SYCL reduction kernels optimized for Intel GPUs.

B. IPC Buffers Preparation

Since each process maintains its buffer addresses, and these addresses are inaccessible to other peer processes, it is necessary to exchange IPC handles with peers and open these handles to obtain the real memory locations from others. In most point-to-point communication implementations, either the sender or the receiver process opens the IPC handle and executes the device-to-device memcpy from one GPU to another. However, in collective operations, it is unbalanced to assign all the memcpy tasks to only one process; instead, it requires all processes to share the communication workload. Therefore, in our implementation, we exchange the IPC handles mapped to sendbuf and recvbuf of all peer processes. This ensures that each process can access all buffers from the peer



(a) Single-leader 2-level Inter-node Designs

(b) Multi-leader 2-level Inter-node Designs

Fig. 2. Overview of the proposed inter-node designs for MPI_Allreduce.

processes within the same node. This all-to-all exchanging pattern provides the necessary data access possibility and offers higher benefits for kernel reduction in the next step. To optimize the productivity of passing the remote buffer pointers to the kernel function, we refactor the pointers into a 2D array. The Step 1 section of Figure 1 shows the structure of the 2D array. The first dimension indicates the number of peer processes, while the second dimension indicates the number of data counts for reduction, and each process has the same data structure to access the remote buffers.

C. Reduction Kernel Designs

We delegate the computation to the reduction kernel, with a focus on enhancing capacity and throughput by addressing load balance issues and optimizing memory access patterns. Assuming there are n GPUs per node and N data counts, each GPU is responsible for roughly N/n calculating result elements. For instance, in Figure 1, step 2, GPU 0 computes the first 2 elements in the sendbufs, while GPU 1 handles the remaining 2 elements (marked in red and blue dotted circles). From an implementation perspective, the kernel can access the array header by setting the offset, allowing each work item (similar to a thread on NVIDIA GPUs) to access and compute the vectorized elements in the sendbufs concurrently. This process iterates n times, traversing the data from each peer GPU and storing the reduced data in the assigned recvbuf afterwards. In step 2, GPU 0 calculates the first 2 elements in sendbuf 0 and sendbuf 1, storing the results in recvbuf 0. Meanwhile, GPU 1 calculates the remaining elements in sendbufs and stores the results in recvbuf 1.

Since computations on GPUs may occur simultaneously, multiple kernels may access the same data (indicates on the same GPU) concurrently, leading to potential memory congestion. To address this issue, for each GPU x, our design only accesses the data stored on GPU x in the first iteration. Then, in each subsequent iteration, it accesses the sendbuf located on GPU $x + 1$, and continues this progression. For instance, in Figure 1, during step 2, GPU 1 first computes the data in sendbuf 1 and subsequently processes the data in sendbuf 0 to avoid congestion with GPU 0. After computation, the kernel stores the results in the appropriate buffer, marking the completion of the basic reduction kernel process. For Allreduce, there is an additional step to broadcast the results to all GPUs. Therefore, we first store the results in the local GPU, such as GPU 1 storing the data in recvbuf 1 instead. For the same memory congestion reason, we also employ a strategy of initially copying data to the $x + 1$ GPU, followed by iterative transfers to the $x + 2$ GPU, and so forth, until completing the broadcast process, as shown in Step 3 of Figure 1. Since all GPUs must broadcast their data to others, this behaviour can be considered an Allgather communication pattern.

D. Inter-node Allreduce Implementations

We implemented two inter-node Allreduce algorithms: the single-leader 2-level Allreduce approach and the multi-leader 2-level Allreduce approach.

1) Single-leader 2-level Allreduce: To maximize efficiency, the GPU kernel operates within the node's scope, producing partial reduction values. Hence, for inter-node Allreduce implementation, we adopt a two-level approach for the remaining reduction operations. The first level entails intra-node reduction, performed by the kernel. As the partial-reduced values are not the final results, the last step of broadcasting the results to all GPUs is unnecessary. To efficiently access this intermediate data for the second level of Allreduce, we store the data on local rank 0, the local leader rank, rather than on each GPU. The second level executes inter-node reduction. With the data residing in the leader ranks of each node, only one process per node participates in this operation. Ideally, any Allreduce algorithm or GPU-optimized techniques can be applied here. In our implementation, we demonstrate the naive approach using a CPU-based leader Allreduce algorithm with CPU staging techniques. We begin by copying the data from the GPU buffers of local rank 0 to the CPU staging buffers, then execute the inter-node leadership Allreduce among the involved processes. After this step, we obtain the final results, but the data is only accessible in CPU buffers on the leader processes. Therefore, we must copy the data back to the GPU buffers on the leader ranks and broadcast the results to all other GPUs. Another approach involves broadcasting the results among the CPU staging buffers and then copying the data back to each GPU individually. However, this method introduces additional communication traffic and overhead. Figure 2(a) illustrates an example of our designs running on 2 nodes, each containing 2 GPUs. Note that rank 0 and rank 2 are the local leader processes.

2) Multi-leader 2-level Allreduce: In the single-leader 2-level approach, only one process per node handles the secondary Allreduce phases with all data, while the other processes idle and wait, causing workload imbalance. Therefore, in the multi-leader 2-level approach, each local rank group handles an equal portion of the data to distribute the workload evenly. To support this feature, the reduction kernel should store the corresponding temporary results on each GPU instead of consolidating them to the root (local rank 0). This allows the secondary Allreduce functions to directly access data from their respective GPU buffers without additional data movement. Following this, all the local rank n processes (on each node) form groups, and each group performs a similar inter-node leader Allreduce among the involved processes, but handling smaller portions of the data involved. After the secondary multi-leader Allreduce, the reduced data is distributed across each local rank rather than the local root rank, so we need to distribute the final results to all processes. This Allreduce algorithm is equivalent to the Reduce_Scatter communication pattern followed by the Allgatherv communication pattern (where "v" accounts for uneven data distribution among processes). Therefore, we can simply perform a local Allgather(v) communication to dispatch the final reduction data. Figure 2(b) illustrates an example of our multi-leader 2-level design. In this design, there are 2 inter-node Allreduce operations, with each operation handling only half of the data compared to the single-leader design.

IV. Evaluation

A. Experimental Setup

Our experiments were conducted on the ACES system, developed by Texas A&M University (TAMU), and the Stampede3 system, developed by Texas Advanced Computing Center (TACC). ACES is a composable hardware platform that offers a mixed accelerator testbed, incorporating Intel PVC GPUs, Intel FPGAs, NVIDIA H100 GPUs, and more. We utilized the PVC partition, featuring a dual-socket Intel Xeon Platinum 8468 Sapphire Rapids processor with 48 cores per socket, totalling 96 cores per node. Additionally, the node is equipped with 512 GB of memory and features 4 Intel Data Center GPU Max 1100 GPUs, referred to as Intel PVC 1100. Each PVC 1100 GPU consists of 56 X^e Cores and 48GB of HBM2e memory. The compute nodes are interconnected via Mellanox NDR 400Gbps. The Stampede3 PVC partition features a dual-socket Intel Xeon Platinum 8480 Sapphire Rapids processor, with 48 cores per socket, totalling 96 cores per node. Each node is equipped with 1 TB of memory and 4 Intel Data Center GPU Max 1550 GPUs, referred to as Intel PVC 1550. Each PVC 1550 GPU consists of 128 X^e Cores and 128 GB of HBM2e memory, spread across 2 stacks (tiles). Therefore, at the user level, there are 8 Intel PVC 1550 GPUs visible per node. The interconnect of Stampede3 is a 100 Gb/sec Omni-Path (OPA) network with a fat tree topology.

Evaluation at the Benchmark Level: The OSU Micro-Benchmarks (OMB) suite [15] provides evaluations at the MPI level both point-to-point and collective MPI runtimes, and it supports CUDA and ROCm device buffers on NVIDIA and AMD GPUs, but it currently lacks support for Intel GPUs. To address this limitation, we extended OMB version 7.3 to facilitate the allocation of oneAPI/SYCL device buffers customized for Intel GPUs. If no specific specification is provided, we default to using the osu_allreduce benchmark with the MPI_SUM operation and MPI_FLOAT datatype.

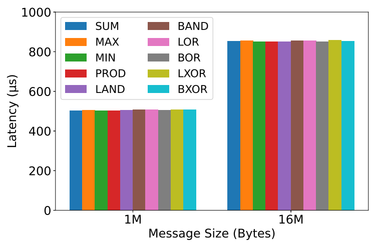
Evaluation at the Application Level: TensorFlow and PyTorch are widely recognized DL frameworks, while Horovod offers a streamlined interface for distributed learning. To harness the capabilities of Intel GPUs, we utilized the Intel Extension for TensorFlow [7] and PyTorch [8] packages for XPU support. Additionally, we utilized the Intel Optimization for Horovod [9] branch and extended it to support MPI standard operations.

We compiled our implementation with Intel oneAPI Base Toolkit 2024.1. To benchmark our designs, we utilized Intel MPI 2021.12 as the baseline.

B. Operations and Datatypes

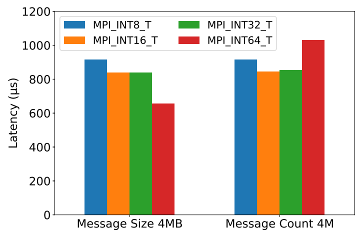
Although summation and floating-point numbers are the most frequently utilized in Allreduce operations, our designs offer comprehensive support for various other common use cases as well. Figure 3 illustrates the performance of various reduction operations on a single node, evaluated at both 1MB and 16MB. The legend enumerates all the supported operations we implemented. The latencies for all 10 different operations consistently range from 503 µs to 508 µs at 1MB and approximately 850 µs to 858 µs at 16 MB. This evaluation proves that our modularized design eases the implementation overhead for GPU kernels while maintaining consistent performance.

Figure 4 illustrates the performance of various data types in Allreduce on a single node. Figure 4(a) displays the results of different types of integers. We present two sets of numbers with the message size fixed at 4MB and the message count fixed at 4M. We offer 4 types of integers: 1, 2, 4, and 8 bytes per integer. Hence, with the fixed message size, the message counts are 4M, 2M, 1M, and 512K, respectively. In this scenario, as the message counts decrease, the running times also decrease due to reduced kernel computation requirements, while the communication volume remains the same. On the other hand, if we maintain the same message counts, the number of computational operations should remain constant. Therefore, the independent variable becomes the message size for communication. However, it's noteworthy that MPI_INT8_T takes more time

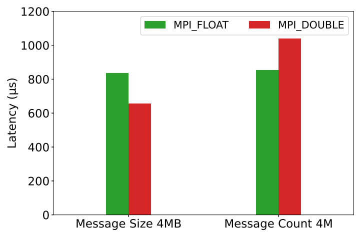


(a) Reduction Operations

Fig. 3. Comparison of Allreduce performance for different reduction operations on 1 node, 4 GPUs.

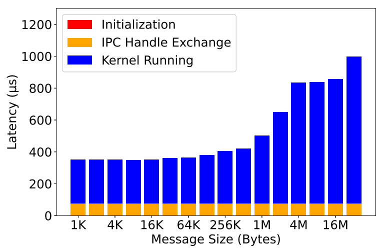


(a) Integers

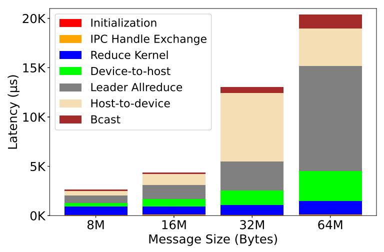


(b) Floating-points

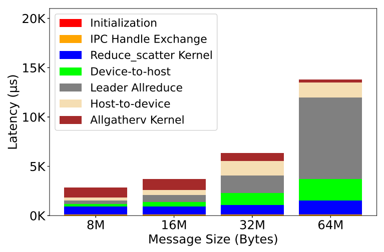
Fig. 4. Comparison of Allreduce performance for different datatypes on 1 node, 4 GPUs. MPI_INT8_T, MPI_INT16_T, MPI_INT32_T, and MPI_INT64_T correspond to 8-bit, 16-bit, 32-bit, and 64-bit integers, while MPI_FLOAT and MPI_DOUBLE represent single-precision and double-precision floating-point numbers.



(a) Intra-node Designs



(b) Inter-node Single-leader 2-level Designs



(c) Inter-node Multi-leader 2-level Designs

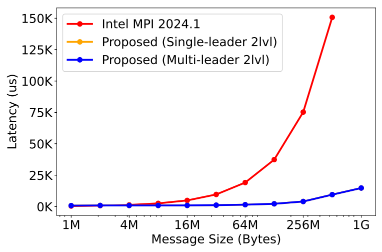
Fig. 5. Time profiling results of the intra-node and inter-node (single-leader and multi-leader 2-level) designs on ACES with 4 GPUs per node.

compared to MPI_INT16_T and MPI_INT32_T. In fact, the performance for MPI_INT16_T and MPI_INT32_T is nearly identical. It is because GPU cores are optimized for 16-bit and 32-bit integers, resulting in more efficient computation for common datatypes. Figure 4(b) displays the results of different types of floating-points. We observe that the performance of MPI_FLOAT is identical to MPI_INT32_T, and MPI_DOUBLE performs similarly to MPI_INT64_T as well.

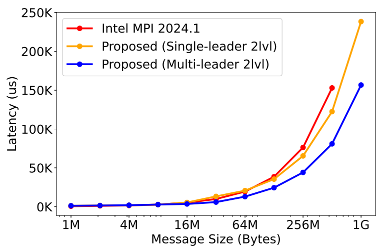
C. Profiling of the Proposed Designs

Due to the additional overhead introduced by kernel-based designs, which becomes noticeable for small messages, we conducted further profiling of MPI operations to analyze our designs. Figure 5 illustrates the time profiling results of the intra-node and inter-node designs for Allreduce. We conducted profiling on ACES using 1 node for intra-node results and 2 nodes for inter-node results, respectively. Figure 5(a) illustrates the three primary stages of the intra-node design: initialization (marked in red), IPC handle creation, exchange, and opening (marked in orange), and kernel launching and running (marked in blue). The initialization phase consistently takes around 2 µs for all message sizes, remaining constant and relatively small, making it negligible. However, a larger overhead is observed in the IPC handle exchange phase, which consistently takes 74 µs regardless of the message size. The constancy of this phase arises from its preparation and exchange of IPC handles rather than actual kernel computation or data communication. In the Kernel Running phase, we also notice a consistent overhead of around 273 µs up to 128KB. Since we cannot delve into the specifics of each kernel, we attribute this to the time taken for kernel launching. This underscores the necessity for alternative approaches to address reduction operations involving small messages. For large messages exceeding 256 KB, the kernel time increases linearly with the message size, indicating the involvement of actual computation. In summary, the Kernel Running phase constitutes the majority of the Allreduce operations. However, compared to other non-kernel-based approaches, its runtime is notably shorter, emphasizing the superiority of the kernel-based methods.

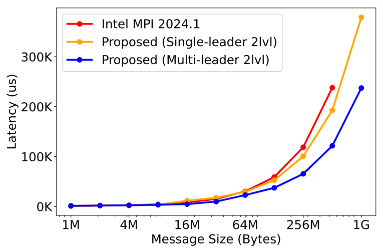
Figure 5(b) and 5(c) illustrate the primary stages of our 2 inter-node designs. Figure 5(b) depicts the profiling results of the single-leader 2-level designs while Figure 5(c) illustrates the results for the multi-leader 2-level designs. In addition to the same phases as intra-node metrics, it includes device-to-host (D2H) memory copying (marked in lime), the inter-node level Allreduce for leader processes (marked in grey), host-to-device (H2D) memory copying (marked in wheat), and IPC broadcast or allgather (marked in brown). We observe a similar trend as intra-node profiling results for the Initialization, IPC Handle Exchange, and Kernel Running phases, with these phases constituting a small portion of the inter-node design. In the single-leader 2-level designs, most of the time is spent in D2H and H2D memory copying and the leader-Allreduce phase, with both phases occupying around 33% and 52% of the total time for a 64MB message. In contrast, in the multi-leader 2-level designs, the time consumed by both the memory copying and the leader-Allreduce phase drops by 45% and 22% compared to single-leader designs, respectively. The other



(a) 4 GPUs

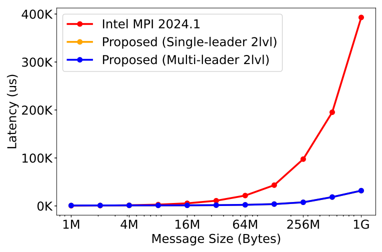


(b) 8 GPUs

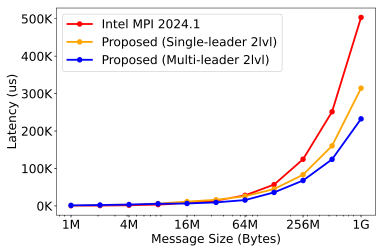


(c) 16 GPUs

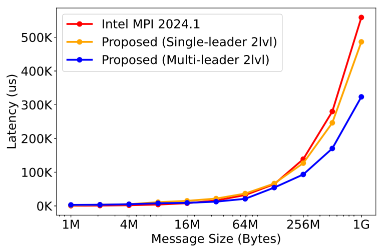
Fig. 6. Comparison of MPI_Allreduce between Intel MPI and proposed designs with single-leader and multi-leader 2-level Allreduce algorithms on ACES.



(a) 8 GPUs



(b) 16 GPUs



(c) 32 GPUs

Fig. 7. Comparison of MPI_Allreduce between Intel MPI and proposed designs with single-leader and multi-leader 2-level Allreduce algorithms on Stampede3.

phases remain consistent, resulting in a total runtime reduction of 32%. This is because the data copying and Allreduce communication workloads are equally distributed across all processes rather than being concentrated on the root rank. The profiling results show that, in the inter-node Allreduce design, most of the time is dedicated to inter-node communication or phases directly related to it. However, despite these challenges, our designs continue to outperform alternative approaches, as demonstrated in the next section.

D. Micro-Benchmark Evaluation

We performed a performance assessment of our kernel-based MPI reduction operations, concentrating on the Allreduce functionality using OMB. The evaluation spanned across configurations involving 4, 8, and 16 GPUs on ACES, and 8, 16, and 32 GPUs on Stampede3. Figures 6 demonstrate that our proposed designs exhibit superior or comparable performance to Intel MPI for message sizes ranging from 1MB to 1GB on ACES. Figure 6(a) illustrates that for larger messages on a single node, the latency of Intel MPI increases to 150.7K µs at 512MB. In contrast, both of our approaches maintain the lowest latency at 9500 µs, which is 15.8 times faster compared to Intel MPI. Notably, Intel MPI hangs at certain message sizes of 1GB, while our approaches continue to deliver good performance. Additionally, the orange and blue lines overlap in some regions. Figures 6(b) and 6(c) further demonstrate that our proposed designs maintain their advantage in multiple-node scenarios. In the 8-GPU scenario, our designs achieve a latency of 122.4K µs with the single-leader designs and 80.9K µs with the multi-leader designs at 512MB, marking a 20% and a 47% improvement over Intel MPI's 152.9K µs, respectively. Similarly, in the 16-GPU environment, our designs achieve a latency of 192.6K and 121.5K µs, presenting a similar 19% and a 49% enhancement over Intel MPI's 237.9K µs, respectively.

Figures 7 demonstrate the performance of our proposed designs on Stampede3. Figure 7(a) illustrates the latencies on a single node. Both of our proposed designs exhibit lower latencies around 31.8K µs, compared to 392.9K µs using Intel MPI at 1GB, which is 13.3 times faster. Figure 7(b) shows that our single-leader designs achieve 314.4K µs and the multi-leader designs achieve an even lower 232.5K µs at 1GB with 16 GPUs, which are 1.6 times and 2.2 times faster compared to Intel MPI's 503.3K µs. Similarly, Figure 7(c) demonstrates on 32 GPUs, our designs achieve latencies of 486.6K and 323.5K µs, presenting 13% and 42% enhancements over Intel MPI's 558.8K µs, respectively. Especially, in the case of medium-sized messages around 64MB, Intel MPI achieves a low latency of 32.2K µs, whereas our single-leader designs achieve 36.6K µs. However, our multi-leader designs significantly improve latency to just 21.0K µs, marking a 34% enhancement over Intel MPI.

E. Extension to Other Reduction Operations

In addition to MPI_Allreduce, our designs have been extended to encompass other reduction-based MPI operations, namely MPI_Reduce, MPI_Reduce_scatter, and MPI_Reduce_scatter_block. Figure 8(a), 8(b), and 8(c) present the performance results on 1 node with 4 GPUs. In summary, our designs demonstrate similar performance to the Allreduce implementations, with latencies ranging from approximately 985 µs to 995 µs for the three operations. Considering that the

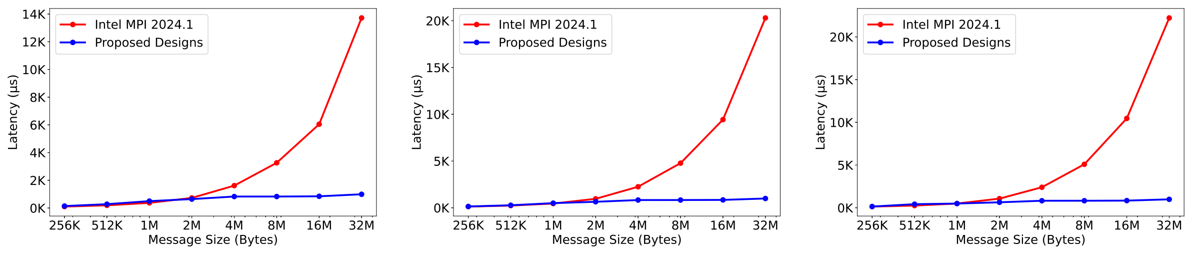

(a) MPI_Reduce (b) MPI_Reduce_scatter (c) MPI_Reduce_scatter_block

Fig. 8. Extension to other reduction operations. Performance comparison with 1 node, 4 GPUs on ACES.

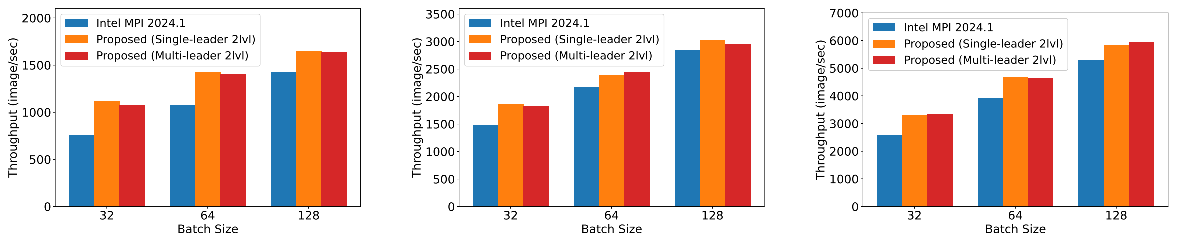

(a) 4 GPUs (b) 8 GPUs (c) 16 GPUs

Fig. 9. Comparison of application-level (TensorFlow with Horovod) performance on ACES. (Higher is better)

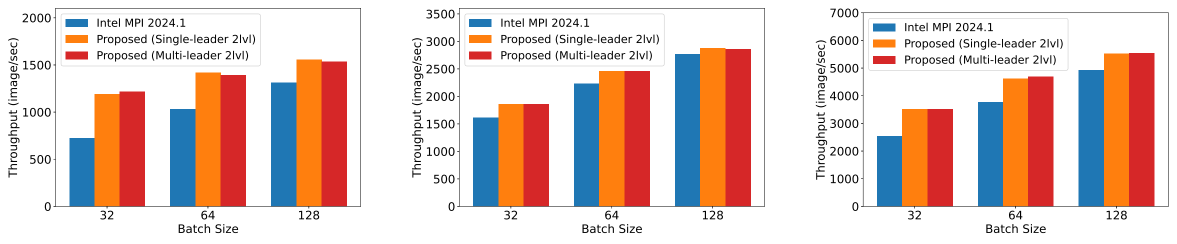

(a) 4 GPUs (b) 8 GPUs (c) 16 GPUs

Fig. 10. Comparison of application-level (PyTorch with Horovod) performance on ACES. (Higher is better)

three operations are subsets of Allreduce, their latencies should ideally be less than or similar to the Allreduce performance. It appears that even Intel MPI lacks optimization for other reduction-based operations, as evidenced by its latencies of 13719 µs, 20300 µs, and 22222 µs for the three operations at 32 MB. This translates to being 13.8x, 20.4x, and 22.5x slower compared to our designs, respectively.

F. Application-Level Evaluation

To assess the advantages of our designs in real-world scenarios, we conducted application-level experiments using DL applications, specifically TensorFlow with Horovod and PyTorch with Horovod.

Figure 9 presents the performance comparison of TensorFlow with Horovod for batch sizes 32, 64, and 128, using the ResNet50 model with both Intel MPI and our proposed designs across 4, 8, and 16 GPUs. On 4 GPUs, as shown in Figure 9(a), our designs achieve throughputs of 1078, 1409, and 1642 img/sec for batch sizes 32, 64, and 128, respectively. In contrast, Intel MPI only provides throughputs of 754, 1074, and 1428 img/sec for the same batch sizes, indicating a 30%, 24%, and 13% decrease in performance compared to our designs. Similar trends are evident in Figure 9(b) and 9(c). For instance, in the 16-GPU scenario, our designs achieve a throughput of 3336 img/sec for batch size 32, outperforming Intel MPI's throughput of 2597 img/sec by 22%.

Figure 10 presents the performance comparison of PyTorch with Horovod for batch sizes 32, 64, and 128, using the ResNet50 model with both Intel MPI and our proposed designs across 4, 8, and 16 GPUs. Similar to the TensorFlow evaluations, our designs consistently outperform the baseline across various batch sizes and scales. Figure 10(a) demonstrates the superior performance of our designs, achieving throughputs of 1216, 1390, and 1537 img/sec for batch sizes 32, 64, and 128, respectively. In contrast, Intel MPI yields lower throughputs of 723, 1033, and 1314 img/sec for the same batch sizes, marking a 41%, 26%, and 15% reduction in performance compared to our designs. In the 16-GPU case depicted in Figure 10(c), our designs achieve 3519 img/sec for batch size 32, surpassing Intel MPI's throughput of 2531 img/sec by 28%.

Reduction operations play a crucial role in DL training, and PyTorch specifically serves as a vital backend for various DL frameworks. The application-level evaluations demonstrate that our design offers significant benefits for common DL tasks, delivering high performance and low latency outcomes.

V. Related Work

The rise of GPU's popularity in modern clusters for compute-heavy workloads necessitates the need for efficient communication between GPUs. Several researchers are actively developing solutions to address this challenge.

Wang et al. [16] explored InfiniBand clusters and developed an optimal NVIDIA CUDA-based design specifically suited for this architecture, facilitating efficient GPU-to-GPU communication. Additionally, in their other work, Wang et al. [17] leveraged Remote Direct Memory Access (RDMA) technology in RDMA-enabled clusters, enabling data transfer between GPUs to bypass the CPU entirely and reduced communication overhead. Jacobsen et. al [18] explored overlapping GPU data movement and MPI communication with computation for simulating computational fluid dynamics using MPI and CUDA. Potluri et al. [19] introduced a hybrid design that combines host-based pipelining techniques with GPUDirect RDMA functionalities. This approach optimizes communication between GPUs located on different nodes within the cluster, leveraging the strengths of both techniques. Subramoni et. al [20] addressed communication inefficiencies by proposing designs that dynamically adapt to the communication patterns of processes at runtime. Their solution allows for seamless transitions between eager thresholds without compromising throughput. To evaluate existing solutions, Kawthar et al. [21] conducted a comparative study on the point-to-point communication performance of popular GPU-aware MPI libraries like MVAPICH2-GDR, Spectrum MPI, and Open MPI, offering valuable insights into their relative strengths and weaknesses. Chen et al. [22] focused on optimizing Alltoall communication, a common data exchange pattern in scientific computing, for dense GPU systems using IPC. Several researchers have investigated methods for optimizing MPI reduction-based collectives on modern GPU architectures. Faraji et al. [23] [24] explored CUDA IPC designs to enhance MPI_Allreduce performance. They expanded on their research by examining various GPU-aware collective algorithms and proposed hybrid designs to optimize medium and large message sizes, utilizing a combination of host-staged and CUDA IPC copies. Chu et al. [25] proposed novel designs for MPI reduction-based collectives, utilizing CUDA kernels for reduction and GPUDirect RDMA features for communication. Furthermore, Chu et al. [26] employed a technique that combines host-staged copies with GPU global memory to accelerate MPI_Allreduce for deep learning workloads.

While research on communication strategies for CUDA-based NVIDIA GPUs has been extensive, there is a growing interest in exploring strategies for alternative architectures too. Kuznetsov et al. [27] ported classical Molecular Dynamics (MD) applications from CUDA to ROCm using HIP and analyzed MD application performance on both NVIDIA and AMD GPUs [28]. Kawthar et al. [29] further extended their work by proposing novel communication designs specifically tailored for AMD GPU clusters using the ROCm-aware MVAPICH2-GDR library, addressing communication challenges for both inter-node and intra-node communication. Similarly, several studies have explored the potential of SYCL, a programming standard for heterogeneous computing, for accelerating various workloads on Intel GPUs. Chen et al. [30] implemented a GPU-aware MPI library for Intel GPUs using oneAPI and SYCL backend and they provided detailed experiments and challenges encountered in integrating Intel GPU-aware support at the MPI layer. Further, thorough benchmarking and evaluations from the paper demonstrate significant speedups in point-to-point and collective MPI operations showcasing the adaptability and performance benefits of the proposed implementations compared to Intel MPI. Zhai et. al [31] designed and implemented SYCL-based GPU backend for the Microsoft SEAL homomorphic encryption library, demonstrating SYCL's effectiveness for cryptographic applications. Deakin et. al [32] evaluated the performance of HPC applications written in SYCL, comparing them to OpenCL and other models. While ardoso da Silva et. al [33] found SYCL's performance not yet on par with OpenCL and OpenMP in their specific study. Ongoing research continues to explore SYCL's potential for optimizing code portability across diverse architectures. Kuncham et al. [34] compared the performance of SYCL code to its CUDA equivalent, demonstrating the viability of SYCL for porting existing CUDA applications. Reguly et al. [35] evaluated the performance portability of an application across various platforms including SYCL, highlighting its potential for achieving code portability.

VI. Conclusion

GPU-aware MPI libraries have been developed over the past decade to meet the rigorous demands of HPC applications. In recent years, the surge in communication requirements driven by emerging DL training has heightened the importance of reduction operations even further. With NVIDIA and AMD GPUs already enjoying a competitive edge, the advancements in GPU-aware MPI libraries have raised expectations for comparable support and optimizations on the upcoming Intel GPUs. The state-of-the-art MPI libraries currently depend solely on naive CPU staging strategies for reduction operations, tasked with managing all messages, resulting in suboptimal performance and significantly high latencies. To tackle this challenge, we have developed kernel-based designs tailored for optimizing MPI reduction operations on Intel GPUs. Our designs prioritize handling large messages, particularly for the prevalent MPI_Allreduce operations. Within intra-node environments, we implemented a pure kernel-based IPC solution to manage the substantial communication and computational demands. In inter-node scenarios, we employed a two-level algorithm to fully exploit the benefits of our kernel designs. In benchmark tests, our Allreduce implementations deliver a 13.3x performance boost over Intel MPI at 1GB with 8 GPUs. Additionally, with 32 GPUs, we realize a 42% performance gain. In application assessments, our designs show up to a 22% performance improvement for TensorFlow with Horovod and a 28% enhancement for PyTorch with Horovod. In the future, we plan to explore more optimization approaches for the second inter-node level Allreduce, aiming to enhance the efficiency of inter-node performance for MPI reduction operations.

References

[1] E. Strohmaier, J. Dongarra, H. Simon, and M. Meuer, "TOP 500 Supercomputer Sites," http://www.top500.org, 1993.

[2] The Open MPI Development Team, "Open MPI : Open Source High Performance Computing," http://www.open-mpi.org, 2004, [Online; accessed October 31, 2024].

[3] MVAPICH2: MPI over InfiniBand, 10GigE/iWARP and RoCE, https://mvapich.cse.ohio-state.edu/, 2001, [Online; accessed October 31, 2024].

[4] S. Khuvis, K. Tomko, S. R. Brozell, C.-C. Chen, H. Subramoni, and D. K. Panda, "Optimizing amber for device-to-device gpu communication," in *Practice and Experience in Advanced Research Computing*, ser. PEARC '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 200–205. [Online]. Available: https://doi.org/10.1145/3569951.3597553

[5] Khronos, "SYCL 2020 Specification Revision 5," https://registry.khronos.org/SYCL/specs/sycl-2020/html/sycl-2020.html, 2022.

[6] A. Ayala, S. Tomov, A. Haidar, and J. Dongarra, "heffte: Highly efficient fft for exascale," in *Computational Science – ICCS 2020*, V. V. Krzhizhanovskaya, G. Závodszky, M. H. Lees, J. J. Dongarra, P. M. A. Sloot, S. Brissos, and J. Teixeira, Eds. Cham: Springer International Publishing, 2020, pp. 262–275.

[7] Intel, "Intel extension for tensorflow," https://github.com/intel/intel-extension-for-tensorflow, 2024.

[8] Intel, "Intel Extension for PyTorch," https://intel.github.io/intel-extension-for-pytorch, 2024.

[9] Intel, "Intel optimization for horovod," https://github.com/intel/intel-optimization-for-horovod, 2024.

[10] Intel Coporation, "Intel MPI Library," http://software.intel.com/en-us/intel-mpi-library/.

[11] MPICH2: High Performance portable MPI implementation, http://www.mcs.anl.gov/research/projects/mpich2.

[12] Intel, "Intel oneAPI," https://www.oneapi.io/, 2022.

[13] Intel, "Intel data center gpu max series overview," https://www.intel.com/content/www/us/en/developer/articles/technical/intel-data-center-gpu-max-series-overview.html.

[14] Message Passing Interface Forum, *MPI: A Message-Passing Interface Standard Version 4.1*, Nov. 2023. [Online]. Available: https://www.mpi-forum.org/docs/mpi-4.1/mpi41-report.pdf

[15] D. Bureddy, H. Wang, A. Venkatesh, S. Potluri, and D. K. Panda, "OMB-GPU: A Micro-benchmark Suite for Evaluating MPI Libraries on GPU Clusters," in *Proceedings of the 19th European Conference on Recent Advances in the Message Passing Interface (EuroMPI)*, 2012, pp. 110–120.

[16] H. Wang, S. Potluri, M. Luo, A. K. Singh, S. Sur, and D. K. Panda, "Mvapich2-gpu: Optimized gpu to gpu communication for infiniband clusters," *Comput. Sci.*, p. 257–266, 2011.

[17] H. Wang, S. Potluri, D. Bureddy, C. Rosales, and D. K. Panda, "GPU-Aware MPI on RDMA-Enabled Clusters: Design, Implementation and Evaluation," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 10, pp. 2595–2605, Oct 2014.

[18] D. Jacobsen, J. Thibault, and I. Senocak, "An mpi-cuda implementation for massively parallel incompressible flow computations on multi-gpu clusters," *Inanc Senocak*, vol. 16, 2010.

[19] S. Potluri, K. Hamidouche, A. Venkatesh, D. Bureddy, and D. K. Panda, "Efficient Inter-node MPI Communication Using GPUDirect RDMA for InfiniBand Clusters With NVIDIA GPUs," in *Parallel Processing (ICPP), 2013 42nd International Conference on*. IEEE, 2013, pp. 80–89.

[20] H. Subramoni, S. Chakraborty, and D. K. Panda, "Designing Dynamic and Adaptive MPI Point-to-Point Communication Protocols for Efficient Overlap of Computation and Communication," in *High Performance Computing*, J. M. Kunkel, R. Yokota, P. Balaji, and D. Keyes, Eds. Cham: Springer International Publishing, 2017, pp. 334–354.

[21] K. S. Khorassani, C.-H. Chu, H. Subramoni, and D. K. Panda, "Performance Evaluation of MPI Libraries on GPU-enabled OpenPOWER Architectures: Early Experiences," in *International Workshop on OpenPOWER for HPC (IWOPH 19) at the 2019 ISC High Performance Conference*, 2018.

[22] C.-C. Chen, K. S. Khorassani, Q. G. Anthony, A. Shafi, H. Subramoni, and D. K. Panda, "Highly efficient alltoall and alltoallv communication algorithms for gpu systems," in *2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2022, pp. 24–33.

[23] I. Faraji and A. Afsahi, "Design considerations for gpu-aware collective communications in mpi," *Concurrency and Computation: Practice and Experience*, vol. 30, no. 17, p. e4667, 2018, e4667 cpe.4667. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.4667

[24] Faraji, Iman and Afsahi, Ahmad, "Gpu-aware intranode mpi_allreduce," in *Proceedings of the 21st European MPI Users' Group Meeting*, ser. EuroMPI/ASIA '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 45–50. [Online]. Available: https://doi.org/10.1145/2642769.2642773

[25] C.-H. Chu, K. Hamidouche, A. Venkatesh, A. A. Awan, and D. K. D. Panda, "Cuda kernel based collective reduction operations on large-scale gpu clusters," in *Proceedings of the 16th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*, ser. CCGRID '16. IEEE Press, 2016, p. 726–735. [Online]. Available: https://doi.org/10.1109/CCGrid.2016.111

[26] C.-H. Chu, P. Kousha, A. A. Awan, K. S. Khorassani, H. Subramoni, and D. K. D. K. Panda, "Nv-group: link-efficient reduction for distributed deep learning on modern dense gpu systems," in *Proceedings of the 34th ACM International Conference on Supercomputing*, ser. ICS '20. New York, NY, USA: Association for Computing Machinery, 2020. [Online]. Available: https://doi.org/10.1145/3392717.3392771

[27] E. Kuznetsov and V. Stegailov, "Porting cuda-based molecular dynamics algorithms to amd rocm platform using hip framework: Performance analysis," in *Supercomputing*, V. Voevodin and S. Sobolev, Eds. Cham: Springer International Publishing, 2019, pp. 121–130.

[28] N. Kondratyuk, V. Nikolskiy, D. Pavlov, and V. Stegailov, "Gpu-accelerated molecular dynamics: State-of-art software performance and porting from nvidia cuda to amd hip," *The International Journal of High Performance Computing Applications*, vol. 35, no. 4, pp. 312–324, 2021.

[29] K. Shafie Khorassani, J. Hashmi, C.-H. Chu, C.-C. Chen, H. Subramoni, and D. K. Panda, "Designing a ROCm-Aware MPI Library for AMD GPUs: Early Experiences," in *High Performance Computing: 36th International Conference, ISC High Performance 2021, Virtual Event, June 24 – July 2, 2021, Proceedings*. Springer-Verlag, 2021, p. 118–136.

[30] C.-C. Chen, K. S. Khorassani, G. K. R. Kuncham, R. Vaidya, M. Abduljabbar, A. Shafi, H. Subramoni, and D. K. Panda, "Implementing and optimizing a gpu-aware mpi library for intel gpus: Early experiences," in *2023 IEEE/ACM 23rd International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, 2023, pp. 131–140.

[31] Y. Zhai, M. Ibrahim, Y. Qiu, F. Boemer, Z. Chen, A. Titov, and A. Lyashevsky, "Accelerating encrypted computing on intel gpus," in *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2022, pp. 705–716.

[32] T. Deakin and S. McIntosh-Smith, "Evaluating the performance of hpc-style sycl applications," in *Proceedings of the International Workshop on OpenCL*, 2020, pp. 1–11.

[33] H. C. Da Silva, F. Pisani, and E. Borin, "A comparative study of sycl, opencl, and openmp," in *2016 International Symposium on Computer Architecture and High Performance Computing Workshops (SBAC-PADW)*. IEEE, 2016, pp. 61–66.

[34] G. K. R. Kuncham, R. Vaidya, and M. Barve, "Performance study of gpu applications using sycl and cuda on tesla v100 gpu," in *2021 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2021, pp. 1–7.

[35] I. Z. Reguly, "Performance portability of multi-material kernels," in *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, 2019, pp. 26–35.