Enhancing Intra-Node GPU-to-GPU Performance in MPI+UCX through Multi-Path Communication

Amirhossein Sojoodi
Queen's University
Kingston, Ontario, Canada
amir.sojoodi@queensu.ca

Yıltan Hassan Temuçin
Queen's University
Kingston, Ontario, Canada
yiltan.temucin@queensu.ca

Ahmad Afsahi
Queen's University
Kingston, Ontario, Canada
ahmad.afsahi@queensu.ca

Abstract

Efficient communication among GPUs is crucial for achieving high performance in modern GPU-accelerated applications. This paper introduces a multi-path communication framework within the MPI+UCX library to enhance Point-to-Point (P2P) communication performance between intra-node GPUs, by concurrently leveraging multiple paths, including available NVLinks and PCIe through the host. Through extensive experiments, we demonstrate significant performance gains achieved by our approach, surpassing baseline P2P communication methods. More specifically, in a 4-GPU node, multi-path P2P improves UCX Put bandwidth by up to 2.85x when utilizing the host path and 2 other GPU paths. Furthermore, we demonstrate the effectiveness of our approach in accelerating the Jacobi iterative solver, achieving up to 1.27x runtime speedup.

CCS Concepts: • **Software and its engineering → Message passing**.

Keywords: MPI, UCX, GPU, P2P, Multi-Path Communication, NVLink, PCIe

ACM Reference Format:
Amirhossein Sojoodi, Yıltan Hassan Temuçin, and Ahmad Afsahi. 2024. Enhancing Intra-Node GPU-to-GPU Performance in MPI+UCX through Multi-Path Communication. In *The 3rd International Workshop on Extreme Heterogeneity Solutions (ExHET '24), March 02–06, 2024, Edinburgh, United Kingdom.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3642961.3643800

1 Introduction

Utilizing GPUs in High-Performance Computing (HPC) systems have become increasingly widespread in recent years, delivering substantial speedups to distributed applications in various domains [1, 14]. To fully harness the compute power of these accelerators, distributed applications should efficiently handle the communication among GPUs. In particular, performant data transfers between intra-node GPUs using the communication libraries like Message Passing Interface (MPI), the de facto standard for distributed computing [5], is important for achieving desirable compute power [4].

Traditionally, P2P communication in GPU-accelerated systems is driven via direct data transfers through a single NVLink or Peripheral Component Interconnect Express (PCIe) communication path, often becoming the communication bottleneck and leading to limited performance [12]. Although several studies have been conducted to improve P2P communication performance by splitting it across various paths, like in the Unified Communication X (UCX) library [10], they still lack the ability to provide concurrent data staging through host and device [8, 12].

In our earlier work, we showcased how harnessing host-staging data transfer could enhance P2P communication [12, 13]. In this paper, we build upon our prior research and extend it further. Our contributions are as follows:

- We propose a multi-path communication framework within the UCX library, specifically designed to enhance P2P communication between two intra-node GPUs, by leveraging concurrent path utilization, including available NVLinks and PCIe through the host.
- We design and implement a 2-D pipelining engine to statically scatter the communication along and across the communication channels.
- We provide end users with tuning capabilities for scheduling parameters, ensuring adaptability to diverse communication patterns and hardware configurations.

To evaluate the effectiveness of our approach, we conducted several experiments on GPU-accelerated micro-benchmarks and an application. The results highlight the importance of concurrent path utilization, across multiple NVLinks and PCIe paths, in accelerating GPU-to-GPU communication and optimizing overall application performance.

The remainder of this paper is organized as follows. Section 2 provides the necessary background for this work. Section 3 describes the design and implementation of our multi-path communication framework. Section 4 presents the experimental setup, performance evaluation, and analysis of the results. Section 5 provides a review of the related

ExHET '24, March 02–06, 2024, Edinburgh, United Kingdom

ACM ISBN 979-8-4007-0537-3/24/03
https://doi.org/10.1145/3642961.3643800

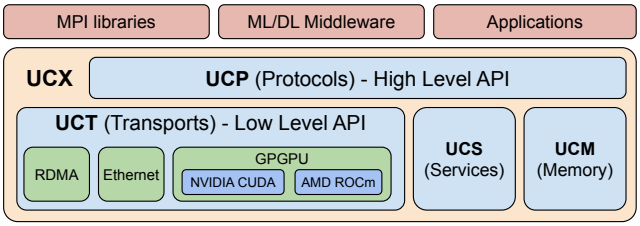

Figure 1. UCX architecture and some of its components

work. Finally, Section 6 concludes the paper, summarizing the contributions and discussing future directions.

2 Background and Motivation

2.1 MPI and UCX

MPI is a portable message-passing standard for parallel programming and Inter-Process Communication (IPC) in HPC which is maintained by MPI Forum [5]. Open MPI is a popular MPI implementation which has been widely used in HPC and AI [9]. It provides a modular architecture that allows for the integration of various communication frameworks, such as UCX. UCX is an open-source library that provides a communication framework for distributed systems [10]. UCX allows applications to utilize various communication protocols and hardware, including shared memory, InfiniBand, RoCE, and more. UCX provides such applicability through various components, some of which are depicted in Fig. 1.

One of the major components of UCX is the Unified Communication Transports (UCT) layer, which consists of various modules that provide communication facilities for different hardware platforms. Each of these modules contains several resources and communication objects [16], including:

- ***MD***: The Memory Domain object facilitates memory registrations and allocations for the underlying transports.
- ***Iface***: This module represents a communication resource on a specific device with a specific transport, associated with a particular worker.
- ***EP***: Endpoint represents a connection to a remote peer.

2.2 CUDA and CUDA module in UCT

Compute Unified Device Architecture (CUDA) provides a programming interface and tools enabling developers to harness the computational power of NVIDIA GPUs [3]. Over the past few years, distributed middleware and libraries have been developed to support GPUs in HPC systems. Combining UCX and Open MPI is one of the various methods to enable GPU support in Open MPI. Some other worth mentioning methods include NVIDIA Collective Communications Library (NCCL) [7], Unified Communication Collectives (UCC) [15], and Open MPI built-in support for GPUs [9]. Although this work targets CUDA and Nvidia GPUs, it is also applicable to AMD, Intel GPUs, etc.

The `uct_cuda` module [16] consists of a `base` and three other submodules: `cuda_copy`, `cuda_ipc`, and `gdr_copy`. While the `gdr_copy` module is performant for small messages, and `cuda_copy` is designed for single-process environments, we integrate our design with `cuda_ipc` module, as this module is frequently used for MPI's large messages.

Typically, data transfers with sizes larger than 64KB are executed in a *rendezvous* fashion. Depending on if sender or receiver reaches to the P2P communication first, as well as the UCX internal configurations, *put* or *get* operation will be executed. To execute either of these operations, the *src* and *dst* buffers' pointers should belong to a common CUDA *context*. In other words, the communication partners should share their CUDA IPC memory handles to transfer data across their address spaces. This challenge is handled by the *CUDA IPC Cache*.

CUDA *contexts* are created by the CUDA *driver* and are associated with a single CUDA *device*. Each *context* belongs to a single process, and each process, as well as each device, can have multiple *contexts*. The CUDA *context* is like an umbrella for the collection of CUDA resources, such as CUDA *streams*, CUDA *events*, and memory pointers. Therefore, at any given time, any CUDA operation, utilizing the device, must be associated with an active CUDA *context*.

Traditionally, in GPU accelerated applications, the CPU needed to coordinate GPU computation and communication tasks explicitly, and messages between third-party devices had to be transferred through the system's main memory. However, GPUDirect technologies have enabled on-node or off-node GPUs to directly exchange data. More relevant to this work, GPUDirect_P2P enables the same feature between the GPUs of a single node.

NVLink is a bidirectional interconnect which consists of sub-links for each direction. Fig. 2(a) demonstrates a typical four-GPU node with NVLink interconnects. In this configuration, each pair of GPUs has two NVLink sub-links, and the GPUs are connected to the CPU through PCIe.

2.3 Motivation

As mentioned earlier, UCX relies on a single communication path for intra-node GPU-to-GPU data transfers. As depicted in Fig. 2(b), if a communication over a single channel (e.g., GPU-0 to GPU-1) has already saturated the available bandwidth, it can be split into smaller chunks and be transferred through other available paths concurrently. This is particularly important for HPC or Deep Learning (DL) applications that utilize MPI's P2P communication for large messages.

While this approach can theoretically improve the communication bandwidth for large messages, the performance gain is dependent on the concurrent communication pattern, and the hardware configuration. Although, the mentioned challenges are not trivial, the performance gain can still be significant, considering NVLink's ability to handle bidirectional communication.

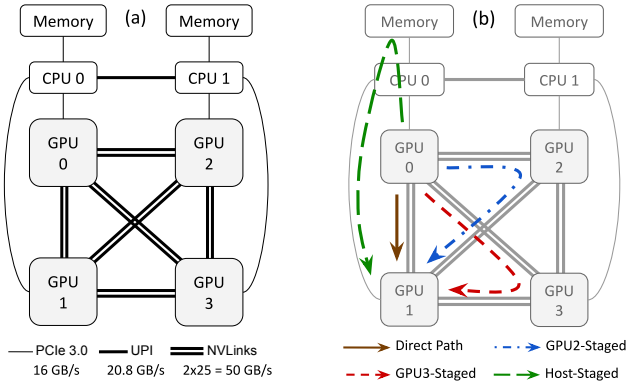

Figure 2. (a) A typical four-GPU node with NVLink (two sub-links) per GPU pair, (b) A communication between GPU-0 and GPU-1 is split and transferred through multiple paths

3 Design

Our design should seamlessly integrate with the existing capabilities of `uct_cuda` module while introducing novel features to exploit concurrent paths for enhanced communication efficiency. Our key objectives include:

1. **Multi-GPU awareness:** Ability to utilize all the available GPUs and their interconnects within UCX instances.
2. **Path selection:** Ability to select the most suitable paths for the current communication.
3. **Communication scheduling:** Ability to schedule the communication along the selected paths.
4. **Path optimization:** Optimize communication bandwidth by increasing the overlap between concurrent data transfer across multiple paths.
5. **Data integrity:** Ensure the data integrity by avoiding link contention and data corruption.
6. **Low overhead:** The overhead of the framework should be negligible compared to the communication time.

3.1 Framework Design and Implementation

Fig. 3 provides a simplified overview of our framework's architecture, illustrating the connections between the key components. The top-level entities of our framework are briefly described below:

- **Base module:** The `base` module is responsible for detecting the available GPUs and their interconnects, and initializing the structures and objects for each GPU, including: *streams*, *events*, *memory handles*, *path configs*, and *path handlers*.
- **CUDA IPC module:** The `cuda_ipc` module is responsible for handling the communication requests, selecting the suitable paths, handling configurations from environment variables, and scheduling the communication along the selected paths with the 2-D pipelining engine. The connection between the `base` and `cuda_ipc` modules is established through proxy entities.

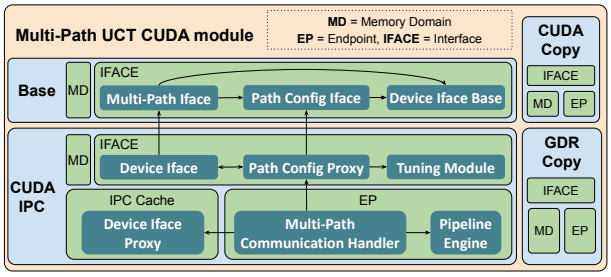

Figure 3. Multi-path communication framework within the `uct_cuda` module, based on UCX v1.14.0

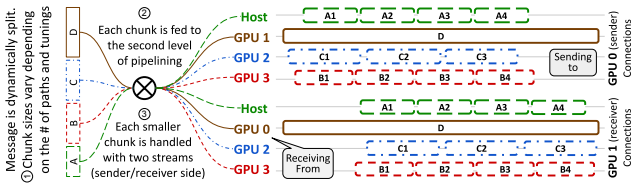

Figure 4. A simplified view of 2-D pipelined communication from GPU-0 to GPU-1 using the available NVLinks and PCIe. Staging GPUs' timelines are not shown for simplicity.

3.2 2-D Pipelining and Configuration Tuning

Fig. 4 demonstrates how our 2-D pipelining engine scatters a message: first, the message is split into multiple chunks according to the number of selected paths; then, each of those is transferred through a different path with another level of pipelining. The number of chunks per path varies from 1 to 16, depending on the size and the path itself. Each of the final chunks is handled by two separate CUDA *streams*, one for transferring from *src* to the staging device, and the other for transferring from the staging device to the destination GPU. The number of CUDA *streams* per path is equal to the number of chunks per path. The CUDA *streams* are synchronized at the end of each path to ensure that the data is received.

Configurations can be tuned through either environment variables or topology-specific tuned parameters. In the former, the user can set the environment variables to tune the framework's behavior, like disabling the host path, setting the number of paths, and setting the number of chunks for each path. In the latter case, we have implemented a static tuning approach that exhaustively searches for the best configuration for the topology. This algorithm works based on the number of GPUs, the number of paths, and the number of chunks for each path. Moreover, this offline tuning does not consider the ongoing communication on the system, and its results are supposed to be used as an initial configuration.

Our experiments reveal that our proposed approach is suitable for large messages (larger than 1MB), as the available bandwidth is not fully saturated for smaller sizes, and splitting the message into multiple chunks and transferring them concurrently will not improve the performance.

3.3 Ensuring Data Integrity

The pipelining engine and the communication handler utilize the synchronization and ordering semantics of CUDA Application Programming Interface (API) to ensure the following properties:

- **Data corruption:** For any direction in the topology (e.g., GPU-2 to GPU-3, or GPU-1 to host) only one communication should occur at any given time. This policy holds true by using the same CUDA *streams* and CUDA *events* for that specific direction.
- **Dependency:** The pipelining engine ensures that each chunk should be received by the staging device before relaying it to the destination GPU.
- **Ordering:** The communication scheduler ensures that the chunks of a single message from each direction are transferred before those of the next message.
- **Synchronization:** All the pipelines are synchronized at the end of the communication, to ensure that the data is transferred completely.

4 Evaluation

To assess the effectiveness and performance of our framework, we conducted the assessment based on three micro-benchmarks and one application. The micro-benchmarks include: UCX Put Bandwidth, OSU Micro-Benchmarks (OMB) MPI Bandwidth (`OMB_BW`), and OMB MPI Bidirectional Bandwidth test (`OMB_BIBW`) [2]. Also, for the application measurements, we tested the Jacobi iterative solver [6].

4.1 Experimental Setup

The experiments were conducted on two different multi-GPU node configuration from the Digital Research Alliance of Canada clusters: Beluga and Narval, equipped with four NVIDIA V100 and NVIDIA A100 GPUs, respectively. In both systems, the GPU topology is full-mesh, and each GPU pair has two NVLinks in Beluga (similar to Fig. 2) and four NVLinks in Narval. The experiments were executed using the UCX library v1.14.0 and Open MPI v4.1.5.

4.2 Micro-benchmark Results

4.2.1 UCX Put Bandwidth. Fig. 5 demonstrates the bandwidth of UCX *Put* operation, using two ranks, on Beluga and Narval clusters. The results show that the bandwidth of *Put* operation is significantly improved by our framework compared to the default UCX. With three GPU paths and host-staging, we observe up to 2.75x and 2.85x speedup, on Narval and Beluga, respectively. As depicted, enabling the host path does not dramatically improve the bandwidth (up to 15%), since the host path bandwidth is much lower than the NVLink bandwidth on these topologies. When utilizing one or two GPU paths, improvements from enabling the host path are similar to using three GPU paths, therefore their respective plots are omitted for the sake of clarity.

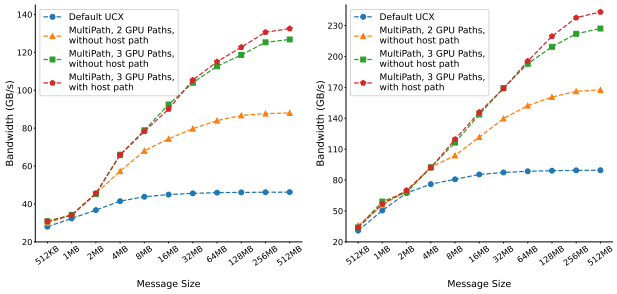

Figure 5. UCX Put Bandwidth comparison against default UCX (UCT::CUDA-IPC), on Beluga (left) and Narval (right)

4.2.2 OMB Micro-benchmarks. Fig. 6 depicts OMB MPI unidirectional and bidirectional bandwidth measurements on Beluga and Narval nodes based on various *window* sizes. The *window* size is the number of communications that can be posted without waiting for the completion of the previous ones. From Fig. 6, we can observe the following:

1. On Beluga, the results are similar to the UCX *Put* bandwidth. However, on Narval, we observe less improvement for a *window* size of one, compared to Beluga: suggesting that four NVLinks on this system do not get saturated similarly to the two NVLinks on Beluga (Figures 6(d) and 6(j), compared to 6(a) and 6(g)). As expected, if we increase the *window* size, the improvement becomes more significant on Narval (Figures 6(e), 6(f), 6(k), and 6(l)).
2. On both clusters, as the *window* size increases, the bandwidth utilization increases for 8MB - 64MB data sizes.
3. As stated earlier, utilizing the host path on Beluga and Narval does not improve the bandwidth significantly, specially for Narval and *window* size one (Fig. 6.d). This might be due to the larger difference between NVLink and the host path bandwidths on Narval compared to Beluga.
4. For bidirectional bandwidth tests, enabling host staging has an adverse effect on the performance, since the host path's PCIe link would become the bottleneck when used concurrently in both directions.

We are currently investigating the reasons behind the drop in performance for 1MB or 2MB messages. We speculate this could be due to the algorithm/protocol change.

4.3 Jacobi Results

We evaluated the MPI version of the Jacobi iterative solver [6], using our proposed multi-path technique. In this measurement, we spawn four MPI ranks on a single node, and each rank is responsible for one GPU. In each iteration of the Jacobi solver, after the ranks have completed their computation tasks, they exchange their data with their adjacent ranks. A similar pattern is depicted in Fig. 7(a), which is basically a *ring*. Considering the unutilized diagonal links, and the NVLink's bidirectional feature, 2/3 of the total available bandwidth is unused in this pattern. Therefore, we can

Figure 6. OMB Unidirectional MPI Bandwidth (BW) and Bidirectional MPI Bandwidth (BIBW) comparison against default UCX (UCT::CUDA-IPC), using two MPI ranks, on Beluga and Narval clusters

enable multi-path communication for all of these data exchanges, and select the staging GPUs in a way that there is no contention on the NVLinks. Fig. 7(b) demonstrates how each communication is split into two paths, and the data is transferred concurrently through the NVLinks.

In our case, Jacobi uses the 2-D Halo Exchange pattern. To evaluate its performance for different data sizes, we set a constant number of elements for one of its dimensions (8) and increase the other dimension (from 2^{23} to 2^{30}). As previously discussed, we have observed poor performance for cases where the host path is enabled, therefore we only report the results for the cases where the host path is disabled. As depicted in Fig. 8, application runtime improvement for cases when we have two paths per P2P communication is considerable, and it increases as the application size increases (up to 1.26x and 1.15x on Beluga and Narval, respectively). Also, the results for the cases with three GPU paths are still better than the baseline, even when there is contention on the NVLinks (up to 1.2x and 1.08x on Beluga and Narval, respectively). It is worth mentioning that Jacobi convergence is not affected by our multi-path communication framework.

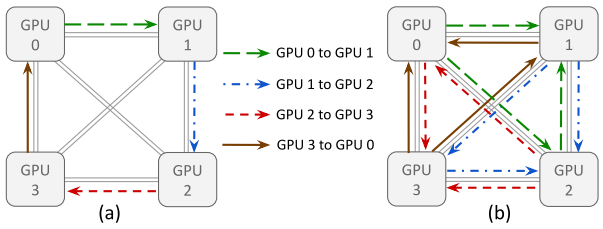

Figure 7. Jacobi communication pattern (a) without multi-path communication, and (b) with multi-path communication (two paths per communication) on a four-GPU node

5 Related Work

Tatsugi and Nukada [11] propose a method to enhance the performance of a data transfer from a GPU to host, by utilizing the idle GPUs. Their framework targets single-GPU applications running on a multi-GPU node, while our approach is designed for multi-GPU applications.

As part of our prior studies [12, 13], we enhance P2P communication within the UCX library by utilizing host-staging

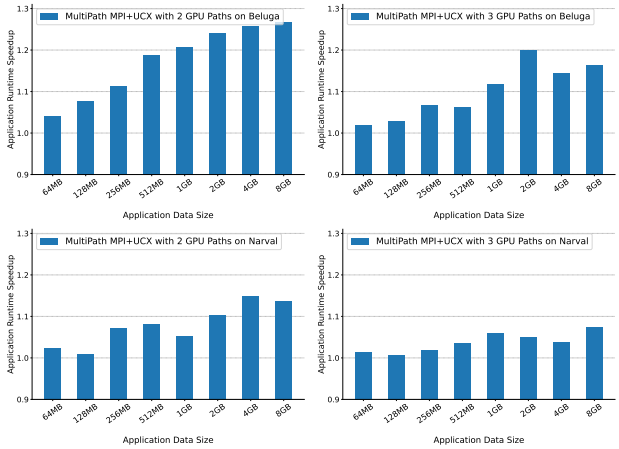

Figure 8. Jacobi speedup against default UCX (UCT::CUDA-IPC), using four MPI ranks, on Beluga and Narval clusters

multi-path communication. Comparing to our prior work, we extend our design to support concurrent path utilization, including both NVLinks and PCIe paths.

In [8], Nukada proposes a method that utilizes PCIe path to accelerate `MPI_Allreduce` on a multi-GPU system. Although their method involves collectives, they follow a similar approach to our previous work to enhance each P2P communication. Again, our approach utilizes both NVLinks and PCIe paths, while their method is limited to PCIe.

6 Conclusion and Future Work

We proposed a multi-path communication framework implemented within the UCX library. By leveraging available communication channels, our 2-D pipelining engine scatters P2P communication across both NVLink and PCIe channels, to maximize communication bandwidth between intra-node GPUs. We also provide end users with tuning capabilities for scheduling parameters, ensuring adaptability to diverse communication patterns and hardware configurations.

In our experiments, we observed up to 2.85x and 2.75x improvement in bandwidth tests for Beluga and Narval clusters, respectively, when using the host path and two GPU paths. We also observed that besides utilizing unused interconnects, harnessing NVLink's bidirectional features will also improve communication performance. However, the lack of the same feature in PCIe communication channels may lead to contention and consequently, performance degradation. Finally, we showed that our multi-path communication framework can improve the performance of the Jacobi iterative solver by up to 1.26x.

While our approach can theoretically improve the communication bandwidth, the performance gain is dependent on the concurrent communication pattern. A possible future direction is to dynamically adapt the communication pattern based on both the application's communication pattern and the hardware configuration.

Acknowledgments

This research was supported in part by the Natural Sciences and Engineering Research Council of Canada and Digital Research Alliance of Canada. Computations were performed on Beluga and Narval with support from Calcul Québec (calculquebec.ca).

References

[1] David E. Bernholdt, Swen Boehm, George Bosilca, Manjunath Gorentla Venkata, Ryan E. Grant, Thomas Naughton, Howard P. Pritchard, Martin Schulz, and Geoffroy R. Vallee. 2020. A survey of MPI usage in the US exascale computing project. *Concurrency and Computation: Practice and Experience (CCPE)* 3 (2020), 1–16. https://doi.org/10.1002/cpe.4851

[2] Devendar Bureddy, H. Wang, A. Venkatesh, S. Potluri, and D. K. Panda. 2012. OMB-GPU: A micro-benchmark suite for evaluating MPI libraries on GPU clusters. In *Proceedings of the European MPI Users' Group Meeting (EuroMPI)*. 110–120. https://doi.org/10.1007/978-3-642-33518-1_16

[3] CUDA 2023. CUDA. https://docs.nvidia.com/cuda/index.html

[4] Ang Li, Shuaiwen Leon Song, Jieyang Chen, Jiajia Li, Xu Liu, Nathan R. Tallent, and Kevin Barker. 2020. Evaluating Modern GPU Interconnect: PCIe, NVLink, NV-SLI, NVSwitch and GPUDirect. *IEEE Transactions on Parallel and Distributed Systems* 31, 1 (2020), 94–110. https://doi.org/10.1109/TPDS.2019.2928289

[5] MPI Forum 2023. MPI Forum. https://www.mpi-forum.org/

[6] Multi-GPU Jacobi Solver 2023. Multi-GPU Jacobi Solver. https://github.com/NVIDIA/multi-gpu-programming-models

[7] NCCL 2023. NVIDIA Collective Communications Library. https://github.com/NVIDIA/nccl

[8] Akira Nukada. 2022. Performance Optimization of Allreduce Operation for Multi-GPU Systems. In *Proceedings of the International Conference on Big Data (Big Data)*. IEEE, 1–6. https://doi.org/10.1109/bigdata52589.2021.9672073

[9] OpenMPI 2023. Open MPI. https://www.open-mpi.org/

[10] Pavel Shamis, Manjunath Gorentla Venkata, M Graham Lopez, Matthew B Baker, Oscar Hernandez, Yossi Itigin, Mike Dubman, Gilad Shainer, Richard L Graham, Liran Liss, et al. 2015. UCX: an open source framework for HPC network APIs and beyond. In *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*. IEEE, 40–43.

[11] Yuya Tatsugi and Akira Nukada. 2022. Accelerating data transfer between host and device using idle GPU. In *Proceedings of the Workshop on General Purpose Processing using GPUs (GPGPU)*. 1–6. https://doi.org/10.1145/3530390.3532732

[12] Yıltan Hassan Temucin, Amirhossein Sojoodi, Pedram Alizadeh, and Ahmad Afsahi. 2021. Efficient Multi-Path NVLink / PCIe-Aware UCX based Collective Communication for Deep Learning. In *Proceedings of the IEEE Symposium on High-Performance Interconnects (HOTI)*. 1–10. https://doi.org/10.1109/HOTI52880.2021.00018

[13] Yıltan Hassan Temucin, Amirhossein Sojoodi, Pedram Alizadeh, Benjamin W Kitor, and Ahmad Afsahi. 2021. Accelerating Deep Learning using Interconnect-Aware UCX Communication for MPI Collectives. *IEEE Micro* (2021), 1–9. https://doi.org/10.1109/MM.2022.3148670

[14] Top500 2023. Top500. https://top500.org/

[15] UCC 2023. Unified Collective Communication (UCC). https://github.com/openucx/ucc

[16] UCX 2023. The Unified Communication X Library. https://openucx.org/