2023 IEEE/ACM 23rd International Symposium on Cluster, Cloud and Internet Computing (CCGrid) | 979-8-3503-0119-9/23/$31.00 ©2023 IEEE | DOI: 10.1109/CCGRID57682.2023.00022

# Implementing and Optimizing a GPU-aware MPI Library for Intel GPUs: Early Experiences

Chen-Chun Chen*, Kawthar Shafie Khorassani*, Goutham Kalikrishna Reddy Kuncham*, Rahul Vaidya*,
Mustafa Abduljabbar*, Aamir Shafi*, Hari Subramoni† and Dhabaleswar K. Panda†
Department of Computer Science and Engineering
The Ohio State University, Columbus, Ohio
*Email: {chen.10252, shafiekhorassani.1, kuncham.2, vaidya.84, abduljabbar.1, shafi.16}@osu.edu
†Email: {subramon, panda}@cse.ohio-state.edu

***Abstract*—As the demand for computing power from High-Performance Computing (HPC) and Deep Learning (DL) applications increase, there is a growing trend of equipping modern exascale clusters with accelerators, such as NVIDIA and AMD GPUs. GPU-aware MPI libraries allow the applications to communicate between GPUs in a parallel environment with high productivity and performance. Although NVIDIA and AMD GPUs have dominated the accelerator market for top supercomputers over the past several years, Intel has recently developed and released its GPUs and associated software stack, and provided a unified programming model to program their GPUs, referred to as oneAPI. The emergence of Intel GPUs drives the need for initial MPI-level GPU-aware support that utilizes the underlying software stack specific to these GPUs and a thorough evaluation of communication. In this paper, we propose a GPU-aware MPI library for Intel GPUs using oneAPI and an SYCL backend. We delve into our experiments using Intel GPUs and the challenges to consider at the MPI layer when adding GPU-aware support using the software stack provided by Intel for their GPUs. We explore different memory allocation approaches and benchmark the memory copy performance with Intel GPUs. We propose implementations based on our experiments on Intel GPUs to support point-to-point GPU-aware MPI operations and show the high adaptability of our approach by extending the implementations to MPI collective operations, such as MPI_Bcast and MPI_Reduce. We evaluate the benefits of our implementations at the benchmark level by extending support for Intel GPU buffers over OSU Micro-Benchmarks. Our implementations provide up to 1.8x and 2.2x speedups on point-to-point latency using device buffers at small messages compared to Intel MPI and a naive benchmark, respectively; and have up to 1.3x and 1.5x speedups at large message sizes. At collective MPI operations, our implementations show 8x and 5x speedups for MPI_Allreduce and MPI_Allgather at large messages. At the application-level evaluation, our implementations provide up to 40% improvement for 3DStencil compared to Intel MPI.**

***Index Terms*—oneAPI, Intel GPUs, GPU-aware MPI**

## I. Introduction

Graphics Processing Units (GPUs) from various industry vendors have increasingly become an integral component of modern High-Performance Computing (HPC) systems [1]. Popular Deep Learning (DL) frameworks like PyTorch and TensorFlow—as well as traditional scientific applications—are now capable of exploiting the raw computer power available on these devices. Historically, NVIDIA and AMD GPUs have dominated the list of accelerators used in HPC systems. The TOP500 project [1] tracks the 500 fastest supercomputers in the world twice every year. According to the latest TOP500 list published in June 2022, NVIDIA Ampere and Volta GPUs hold 26.9% of the overall performance share. The AMD MI250X occupies 30.2% of the overall performance share. However, this share is mainly influenced by the #1-ranked TOP500 system called Frontier at the ORNL.

Despite starting late, Intel is currently actively designing and developing a range of GPU products and the associated ecosystem. In 2020, Intel launched the new discrete GPU—named Intel Iris $X^e$ Max [2]—and provided a new platform, called Intel DevCloud, for developers to access Iris $X^e$ and consumer laptop grade GPUs for development and testing. Intel's next-generation GPUs—called Ponte Vecchio—aim to deliver 2× higher performance than NVIDIA's A100 GPUs [3]. The data-center follow-up GPU named Rialto Bridge is also underway. The Ponte Vecchio GPUs are planned to power the compute nodes of the upcoming exascale system "Aurora" at the Argonne National Laboratory. Figure 1 shows the architectural overview of the Intel Iris $X^e$ Max, also known as DG1. This GPU has 1 Slice, and the Slice consists of 6 Subslices. Each Subslice contains 16 Execution Units (EUs), so there are 96 EUs in total. Just like NVIDIA and AMD GPUs, the programmers are responsible for porting their CPU codes to Intel GPUs by using the associated software ecosystem. An upfront porting effort is also needed for CUDA and HIP-based applications.

As the pioneer of the high-performance GPUs market, NVIDIA came up with its in-house programming toolkit, Compute Unified Device Architecture (CUDA), to simplify the development of GPU applications. Inspired by NVIDIA, AMD also brought Radeon Open Compute (ROCm) software stacks, and the HIP features allow programmers to develop portable applications for AMD GPUs. There are almost one-to-one mappings of CUDA and HIP APIs. Table I summarizes some common APIs of CUDA and HIP. AMD also provides a tool, called hipification, for porting CUDA codes to ROCm applications. Currently, these compute devices—offered by several vendors—are being programmed using a plethora of complex programming languages and environments.

*This research is supported in part by NSF grants #1818253, #1854828, #1931537, #2007991, #2018627, #2112606, and XRAC grant #NCR-130002.

979-8-3503-0119-9/23/$31.00 ©2023 IEEE
DOI 10.1109/CCGrid57682.2023.00022

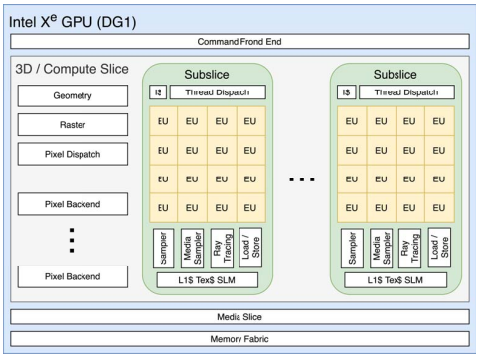



Fig. 1. The architecture overview of Intel Iris Xe Max, a.k.a. DG1. DG1 has 1 Slice and 6 Subslices, and each Subslice contains 16 EUs, so there are 96 EUs in total. Each EU consists of 7 Threads. The global memory size is 7.53 GB, and the local memory size is 64 KB.

TABLE I
THE DIFFERENCE BETWEEN CUDA, ROCM(HIP), AND ONEAPI FOR DIFFERENT API CALLS

| CUDA | ROCm(HIP) | oneAPI |
|---|---|---|
| cudaMemcpy | hipMemcpy | sycl::queue.memcpy |
| cudaMalloc | hipMalloc | sycl::malloc_device |
| cudaFree | hipFree | sycl::free |
| cudaStreamCreate | hipStreamCreate | sycl::queue |
| cudaDeviceSynchronize | hipDeviceSynchronize | sycl::queue::wait |
| cudaMallocHost | hipHostMalloc | sycl::malloc_host |
| cudaMallocManaged | hipMallocManaged | sycl::malloc_shared |
| __syncthreads | __syncthreads | sycl::nd_item::barrier |

Intel simplifies the programmer's task by supporting DPC++ (short for Data Parallel C++), which is an SYCL [4] standard-compliant programming interface. This support comes along with a single unified API and associated packages—called oneAPI [5]—to program the processing elements from multiple vendors to innovate the next generation of DL and HPC applications. oneAPI supports a variety of hardware architectures, including CPUs, GPUs, and even FPGAs. oneAPI [5] includes DPC++ with SYCL and OpenMP for C, C++, and Fortran. Programmers can utilize oneAPI to develop their codes with either kernel (DPC++) or directive-based (OpenMP) style. DPC++ supports the SYCL standard, which is a cross-platform abstraction layer that builds on OpenCL for heterogeneous systems. Intel also provides the translation tool, DPCT, for programmers to easily port their CUDA code to SYCL/DPC++ code-based and support different hardware, including Intel GPUs.

### A. Motivation

The Message Passing Interface (MPI) [6] is a standard for developing distributed parallel programs. With the high demand for GPUs on HPC systems, GPU-aware MPI libraries play a pivotal role by offering efficient and productive communication between GPU-based processes. Programmers can pass the device buffer pointers to MPI primitives as they used to deal with host buffer pointers. The state-of-the-art MPI libraries, such as MVAPICH2-GDR [7] and Open MPI [8], have supported NVIDIA GPUs for a long time now. Support for AMD GPUs has recently been added because of the recent adoption of these GPUs. With the emergence and the potential of Intel GPUs on next-generation clusters, communication libraries need to provide support for these GPUs, too. **The primary motivation of this work is to add support for Intel GPUs in an MPI library.** Intel MPI [9] currently has support for Intel GPUs using the so-called *offloading* approach. We show in this paper that the offloading approach does not provide the best performance, and our proposed solution provides an efficient alternative. Also, Intel MPI is closed-source software, so we, and the community, do not have any insight into how the MPI library is implemented and optimized. *In order to fulfill the high demand from HPC and DL applications on the upcoming supercomputers equipped with Intel accelerators, it is critical to have another high-performance and efficient GPU-aware support on Intel GPUs.*

### B. Challenges

We address the following challenges to develop and optimize an efficient GPU-aware MPI library:

- How can we implement and optimize a high-performance GPU-aware MPI library that supports Intel GPUs and the other GPU systems in the market?
- How to decide the switching point between using versus not using pipeline parallelism for the GPU-aware implementation? And, how to tune the block size for pipeline stages to reduce latency?
- What are the features and limitations of Intel GPUs for GPU-to-GPU communication that can benefit or impact the MPI operations?
- How can we seamlessly design a benchmark to evaluate the performance of the MPI library on Intel GPUs without breaking the generality of the existing OSU Micro-Benchmarks (OMB) interface and main code-base?

### C. Contributions

This paper makes the following contributions:

1) Implement and optimize a GPU-aware MPI runtime by adopting the CPU staging approach and extending the host-based MPI with oneAPI library to support Intel GPUs. (Section III)
2) Explore the features of Intel GPUs and identify the challenges of supporting GPU-aware MPI operations. (Figure 1, Table I)
3) Analyze the different host buffer types to achieve enhanced memory copy performance for communication on Intel GPUs; optimize the implementations with pipeline techniques, identify the bottleneck, and propose advanced implementations. (Section IV)
4) Implement benchmarks to evaluate communication performance on Intel GPU buffers with GPU-aware MPI runtime; implement benchmarks using a naive approach as the baseline. (Section V-A)
5) Evaluate MPI point-to-point and collective operations for GPU resident data and compare our proposed implementations with Intel MPI, Intel oneCCL, and naive approach. The latency of our proposed implementation is better than Intel MPI's by 18%, and by 30% for

the naive point-to-point operations. It reaches up to 8x improvement for collective operations. (Figures 7 and 8)
6) Demonstrate that our proposed implementations offer up to 40% improvement for 3DStencil in the application-level evaluation. (Figure 9)

**To the best of our knowledge, our proposed implementation is the first that offers better performance for both point-to-point and collective operations against the Intel MPI library on Intel GPUs.**

The rest of the paper is structured as follows. The background of our work is given in Section II. Section III and Section IV detail the implementation and optimization of our designs. The experimental results are presented in Section V. The related work is discussed in Section VI, and the paper is concluded in Section VII.

## II. Background

### A. oneAPI

oneAPI [5] provides a "unified" programming model that can be used to target multiple hardware architectures, such as CPU, GPU, FPGA, and other accelerators. It comprises various toolkits and libraries designed for diverse applications and use cases. oneAPI simplifies programming on diverse architectures using the SYCL programming model, which is built on top of OpenCL [10].

### B. SYCL

SYCL [4] is a cross-platform abstraction layer developed by Khronos Group enables code for heterogeneous systems to be written in a single-source C++ file. With SYCL, it is possible to create the software from a single source, allowing C++ template functions to design complicated OpenCL-accelerated algorithms and reuse those methods throughout their source code for various forms of data.

### C. Intel GPU

Intel launched its new discrete GPU "Intel Iris $X^e$ Max" [2] (also known as "DG1") in 2020. This GPU has 96 Execution Units with a maximum clock frequency of 1650 Mhz. It supports up to three-dimensional work items with a maximum of 512 work items in each dimension. The maximum work-group size is 512, the global memory size is 7.53 GB, the maximum memory allocation is 3.76 GB, and the local memory size is 64 KB.

### D. GPU-aware MPI

MPI [6] is a standardized API for communicating data and messages across distributed processes. In conventional MPI implementations, to communicate between two GPUs on different nodes, developers need to take care of **staging** GPU buffers using memcpy. **The staging technique refers to copying source GPU data to host memory to enable host-to-host transfer using MPI, and finally copy the receive buffer to destination GPU.** However, with GPU-aware MPI, the MPI library is capable of sending and receiving GPU buffers directly, without needing to first stage them in host memory.

**Algorithm 1:** GPU-aware Implementation in MPI

```
1  if is_device(buf) then
2  |   void* h_buf = malloc_host<char>(size, q);
3  |   q.memcpy(h_buf, buf, size).wait();;
4  |   void* ori_buf = buf;
5  |   buf = h_buf;
6  end
7  /* Perform MPI operation for host buffer */
8  if is_device(buf) then
9  |   q.memcpy(ori_buf, buf, size).wait();;
10 |   buf = ori_buf;
11 |   free(h_buf, q);
12 end
```

## III. Implementation of GPU-aware MPI Library using oneAPI

Since Intel is new to developing its accelerators, the existing advanced techniques, such as RDMA and GPUDirect RDMA technology, are under development and do not apply to Intel GPUs currently. Hence. in our early experiment, we utilized a CPU staging approach to implement the proposed GPU-aware MPI library. Algorithm 1 reveals how to apply CPU staging techniques to our implementations. First, the MPI library should identify whether the buffer pointers passed by the users are either host buffer or device buffer. If the pointers are device buffers, create and maintain the host staging buffer and start data copying. Otherwise, perform the regular host-based MPI operations.

Existing GPU-aware MPI libraries, such as MVAPICH2-GDR and Open MPI, have explored NVIDIA and AMD GPUs for the past decade. They utilize CUDA or ROCm toolkits to implement their own algorithm over GPUs. Hence, following a similar idea, we adopt the Intel oneAPI library to implement our proposed implementations for supporting Intel GPUs. However, the implementation is more challenging than people ported design from supporting NVIDIA GPUs to AMD GPUs. It is because CUDA and ROCm libraries are nearly 1-to-1 mapping, but this fact does not hold for oneAPI APIs. Table I summarizes the common GPU library APIs. We can find corresponding API pairs in CUDA and ROCm with the prefix `cuda` or `hip`, but the API primitives are quite different for oneAPI. It is to say we have to re-implementation the code structure and algorithm for oneAPI-based implementation. Furthermore, CUDA and ROCm have their runtime libraries called `cuda_runtime.h` and `hip_runtime.h`, which allows users to call the APIs even in C code. However, oneAPI does not support such runtime libraries. It raises the difficulty for programmers to integrate oneAPI into their previous implementation if it was written in C.

We implement both MPI point-to-point operations, including blocking calls MPI_Send, MPI_Recv, and non-blocking calls MPI_Isend, MPI_Irecv, and collective operations, MPI_Bcast, MPI_Reduce, MPI_Allreduce, and MPI_Allgather. In this paper, we will focus on discussing MPI_Send and MPI_Recv.

### A. Buffer Identification

Modern GPU programming interface usually provides a convenient runtime API for users to identify the

buffer type. For example, programmers using CUDA can call `cuPointerGetAttribute` to get the printer attributes, and `attributes.type` will tell the printer type. ROCm programmer can also utilize the corresponding API `hipPointerGetAttributes`. However, oneAPI does not provide 1-to-1 mapping APIs, but developers can use `get_pointer_type` in SYCL to achieve the same functionality. This function will return the buffer type in terms of `sycl::usm::alloc::host`, `sycl::usm::alloc::device`, etc. In our implementation, we only consider `sycl::usm::alloc::device` as the device buffer, so the rest of them return the type of host buffer.

### B. CPU Staging

CPU staging is a well-known and simple approach to dealing with data in the device memory. It maintains a staging buffer in host memory so that we can trigger the original CPU-based MPI algorithm to the host buffer pointer. Algorithm 1 demonstrates how we apply the CPU staging approach to GPU-aware MPI implementations. To fully provide downward compatibility to the regular host buffer, we maintain the original device buffer, as lines 4, 5, and 10 show, before and after the MPI call. Like CUDA provides `cudaMemcpy` interface for programmers to copy data between host and device, oneAPI offers an easier way. In an SYCL queue, `memcpy` allows programmers to copy data without considering the buffer type. It is to say the programmers do not have to pass the memory copying kind tag, such as `cudaMemcpyHostToDevice`, the oneAPI `memcpy` will handle it underlying. Notice that in lines 3 and 9, a `wait` function is called after `memcpy`. It indicates in this algorithm, we are using synchronous memory copy. It will block the execution on the host until the data copies completely. The programmers can perform asynchronous memory copy by removing the `wait` function. This technique can be utilized in our optimized pipeline designs.

### C. Extension to Non-blocking Point-to-point MPI Operations

Extending the current CPU staging implementations to non-blocking MPI operations takes more effort. In blocking MPI operations, there is only one function to be called and the data transferring is completed before the function returns. However, in non-blocking MPI usage, MPI_Wait or MPI_Waitall is called after the non-blocking operations to make sure the message is transferred, which increases the difficulty of maintaining both the host staging buffer and the original device buffer pointers. In our implementations, we keep this necessary information in MPI_Request so that we can perform the third step of CPU staging in MPI_Wait or MPI_Waitall after the data transfer between processes is done.

### D. Extension to Collective MPI Operations

Extending the current implementations from point-to-point to collective operations is simple. We take the same idea of copying data to the host buffer and applying the above

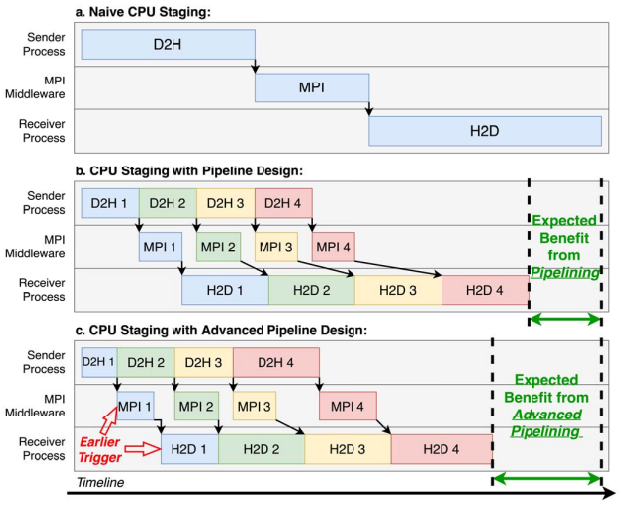



Fig. 2. General timeline view for naive CPU staging and pipeline designs. D2H denotes device-to-host data transferring, MPI denotes CPU-based MPI_Send/MPI_Recv operations between processes, and H2D denotes host-to-device data transferring. The number indicates the index of each message segment. By earlier triggering the H2D 1, the advanced pipeline design (c) provides more expected benefits.

techniques to collective implementations. Make sure to copy only useful data between the host and device buffer, or it may introduce unnecessary communication and drag down the performance. For example, in MPI_Bcast, only copy the data in the root buffer to the staging buffer; or in MPI_Reduce operation, only copy the data in the root buffer back to the device side. Also, make sure to allocate a proper size of the host staging buffer. For example, in MPI_Allgather, the recv buffer size should be the multiplication of `recvcount` and the communication size.

## IV. Optimization of GPU-aware MPI Library

### A. Host Memory Type

The host buffer is allocated and managed by the developers. The very intuitive idea is to allocate CPU memory through `malloc` in the C library. Considering the memory alignment for better performance, people may apply `posix_memalign` to their implementation. Although we can simply call `q.memcpy`, (assuming "q" is a sycl::queue object belonging to GPU device) for copying data between the host and device buffer. The device cannot access any host buffer we pass directly. The data need to be copied to another staging pinned memory allocated on host memory and then be moved from the temporary pinned memory to the final device buffer [11]. Inspired by the fact that using pinned memory in the CUDA program improves the data-transferring speed, we explore similar techniques using oneAPI. It may be more efficient if we can allocate a pre-pinned memory as the host buffer because it can save the time of copying data from the host buffer to the temporary pinned memory.

According to the Intel Developer Guide [12], SYCL USM host allocations may use pre-pinned memory by calling `q.malloc_host`. We also find that we can allocate aligned host pinned memory by calling `q.aligned_alloc_host`

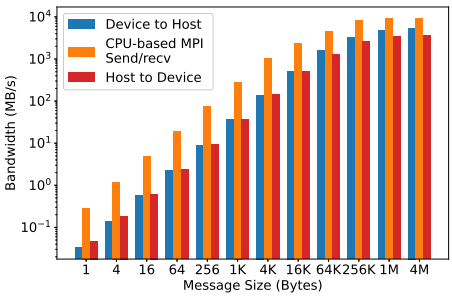



Fig. 3. Individual bandwidth of device-to-host, CPU-based MPI_Send/MPI_Recv, and host-to-device data transferring using Intel GPU on 1 machine. There is overhead for small messages, and the trend of bandwidth becomes smooth after 64KB. Also, host-to-device data transferring has the lowest bandwidth, which indicates it can be the bottleneck of the whole flow.

As a result, we explore 4 different types of memory allocation ({C, oneAPI} x {malloc, aligned malloc}), evaluate the memory transferring time between host and device, choose the best one and use it in our implementations.

### B. Pipeline Design

Algorithm 1 shows the idea of implementing a GPU-aware MPI using CPU staging techniques. This naive approach provides good productivity but poor performance because we do not fully utilize all communication channels. Specifically, the communication between the host and device memory can be overlapped with the communication between processes. Part A of figure 2 depicts this naive approach. It is clear that at any given time, there is no overlapping of any 2 channels. The CPU-based MPI does not start transferring data between processes until the data is entirely copied to the host staging buffer. In the meantime, there is no data copied back to the device buffer until the whole data is entirely transferred through CPU-based MPI communication. However, because the message segments are independent, we can separate the entire message into pieces and send it using pipeline techniques. It provides the potential for overlapping communication among different channels.

Part B of figure 2 demonstrates a timeline profiling result using pipeline designs. The original array of messages is divided into 4 pieces. The CPU-based MPI operation (marked as MPI 1) can start transferring data once the 1$^{st}$ piece of message copied to the staging buffer (marked as D2H 1). Similarly, H2D 1 can be triggered once MPI 1 is done. In the meantime, a similar data-transferring process for the 2$^{nd}$ piece of a message can be triggered once the 1$^{st}$ D2H step is done. Although there are more steps for more message segments, the total elapsed running can be saved because of the overlapping of different communication channels.

Despite the potential for overlapping, the pipeline strategy is unlikely to overlap the communication, especially in the beginning stage fully. Ideally, we can divide the message block very small and try to trigger the following step earlier for more overlapping. However, there are overheads for each step. Figure 3 shows the device-to-host (D2H), CPU-based MPI send-recv (MPI), and host-to-device (H2D) bandwidth

Listing 1. Pseudo code of MPI_Send for the advanced pipeline designs.

```
/* Allocate & maintain the host/device buffer */
for (int i = 0; i < pipeline_length; i++) {
  if (pipeline_length > 1) {
    /* Use the advanced pipeline designs */
    if (i == 0) {
      /* the 1st block */
      offset = i*block_size;
      block_size = block_size - block_size/2;
    } else if (i == pipeline_length - 1) {
      /* the last block */
      offset = i*block_size - block_size/2;
      block_size = block_size*1.5;
    } else {
      /* the other blocks */
      offset = i*block_size - block_size/2;
      block_size = block_size;
    }
  } else {
    /* Without using pipeline approach */
    offset = i*block_size;
    block_size = block_size;
  }
  oneapi_memcpy(buf + offset, block_size, ...);
  MPI_Isend(buf + offset, block_size, ...);
}
for (int i = 0; i < pipeline_length; i++) {
  MPI_Wait(request[i], ...);
}
/* Deallocate & maintain the host/device buffer */
```

of all message sizes. Unfortunately, the bandwidth for small messages is quite low, indicating it requires more time to transfer the same amount of data using a smaller block size. Hence, we are not going to use the pipeline strategy for small message transferring. The bandwidth trend becomes smooth after 64K, which hints that we can set a threshold here for applying the pipeline designs.

The poor performance for small messages hampers us from adopting a more fine-grained pipeline strategy. Figure 2 shows there are serial parts in the timeline, especially at the beginning and before the end of execution. Figure 3 also shows the host-to-device has the worst performance for large messages compared to the previous 2 steps, which indicates it will leave a longer ”tail” that cannot be overlapped at the end. From figure 2 and figure 3, we know that host-to-device data transferring almost dominates the total running time because it is the most time-consuming step, and it can only be triggered after the CPU-based step of the 1$^{st}$ message block is finished. Hence, we propose advanced pipeline designs that aim to trigger the host-to-device data transfer earlier. In the naive pipeline designs, the message block size is even over all blocks. However, the advanced pipeline designs reduce the block size of the first block of the message for sooner device-to-host and CPU-based MPI running time and expect the earlier host-to-device data transferring starting time that benefits the total ending time of the last message block. In the advanced designs, the amount of diminished data in the first message block will be handled in the last message block to ensure all data are transferred. As a result, the proposed designs have lower total running time compared to the naive CPU staging pipeline designs.

We have decided where to start using pipeline designs,

but how many blocks we should take is still challenging. A higher granularity allows more overlapping but introduces more overhead, and vice versa. Evaluation V-D shows the trend that the block size could be half of the message size. Hence, we design a tunable parameter to let users assign the proper number of blocks value based on their machine architecture. In our implementations, we set the default value to 2.

Listing 1 illustrates the pseudo-code of the MPI_Send part for the proposed advanced pipeline designs. It modifies the message sizes of different blocks and intends to trigger the first block of device-to-host data transferring earlier. Line 4 to 6 demonstrates we shrink the block size to half of the normal size, and line 7 to 9 show we add the remaining message size to the last block. Moreover, we need to modify the offset of each buffer pointer by shifting half of the block size. SYCL memory copy and non-blocking send are called after the block size and offset modifications. This mechanism allows the first block to send the message sooner than the regular pipeline designs.

### C. CPU Mapping

Because our algorithm relies on a significant amount of data transferring between the host and device buffer, finding the shortest and most efficient path between CPU and GPU memory is essential. Typically, we prefer the GPU to be bound to the CPU in the same socket and NUMA node. In a multi-node cluster system, we will also consider the HCA (Host Channel Adapter) location and try to bind it with the nearest HCA.

## V. Evaluation

### A. Experimental Setup

We conducted the following evaluations on our in-house systems. The compute node is equipped with 4 Intel® Xeon® Gold 6348H CPUs (96 physical cores in total), 376GB memory, and Intel Iris $X^e$ MAX Graphics card (DG1 as the code name) [2]. The Intel DG1 GPU contains 96 EUs and 7.53 GB of device memory with 68 GB/s memory bandwidth.

To evaluate the basic performance metrics of Intel GPUs, we developed micro-benchmarks using DPC++ to compare the elapsed time for memory copying between buffers. To evaluate the MPI-level performance with GPU-aware support, we implemented and added Intel GPU buffer support to existing benchmarks based on OSU Micro-Benchmarks (OMB) [13] suite version 5.8. It utilizes oneAPI to allocate the device buffer and directly passes device buffer pointers to MPI calls. We only modified and enhanced the backend part of OMB so all the benchmarks, including point-to-point and collective, can benefit without code changes. We also implemented a latency benchmark using a naive approach based on `osu_latency` to simulate the condition that people do not use GPU-aware MPI libraries. We call it the "proposed naive approach" in the following figures.

We use 3DStencil in the application-level evaluation. 3DStencil uses plenty of MPI_Isend and MPI_Irecv pairs to communicate. We ported a previous CUDA version of 3DStencil to SYCL-based code with Intel DPCT tool and manually edited the code fragments which were not ported by Intel DPCT.

We report the mean values of 5 runs for the following benchmark and application-level evaluations. We use Intel MPI Library version 2021.6 and Intel oneCCL Library version 2021.6 as the baseline and Intel oneAPI DPC++/C++ version 2022.1.0 as the oneAPI library.

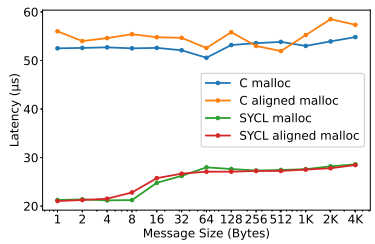
### B. Basic Memory Copy Evaluation for Intel GPUs

To have a comprehensive understanding of the feature of new-coming Intel GPUs, we develop a benchmark to evaluate the basic metrics, including host-to-device (H2D) and device-to-host (D2H) memory copy latency. We allocate the host or device buffer at the beginning and perform memory copy operations through `q.memcpy` using oneAPI. We take two numbers from the mean value of 1000 iterations after 200 warmups.

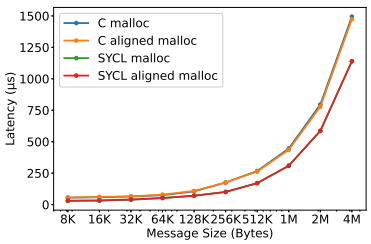
We evaluate the memory copy speed between the device and different types of host buffers. The host buffer types include memory and aligned memory allocated using C and oneAPI library. In particular, we use `malloc` and `posix_memalign` in C and use `malloc_host` and `aligned_alloc_host` in SYCL. We fix the alignment value to 4KB. Figure 4 depicts the results of H2D and D2H memory copy latency from 1 byte to 4MB. We notice that the numbers for *malloc* and *aligned malloc* are almost identical for both C and SYCL type of buffer, so we only consider aligned memory numbers in the following discussion. The H2D and D2H memory copy latencies with SYCL malloc host buffer are as low as 24.8 µs and 28.4 µs for 16 bytes in figure 4(a) and figure 4(c). Compared to the numbers with C malloc host buffer, which are 54.1 µs and 54.8 µs shown in, it is approximately 2x faster. For large message at 4M shown in figure 4(b) and figure 4(d), the numbers with SYCL malloc host buffer are 1140 µs and 773 µs, which is 1.3x and 1.2x faster compared to 1492 µs and 1249 µs using a C malloc host buffer. The results suggest using pinned host buffer has significant performance improvement compared to a regular host buffer, and the improvement is more pronounced for small messages. Moreover, we notice the D2H latency is lower than the H2D latency for large messages. For example, the D2H speed is 1.5x faster than the H2D speed at 4MB. These evaluation results suggest our proposed implementation to use SYCL malloc to allocate the host staging buffer. We need to be aware that host-to-device memory copy can be a bottleneck in the staging flow.

### C. Different Memory allocation Approaches for CPU Staging Technique

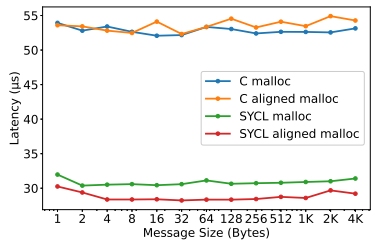
We implement GPU-aware MPI point-to-point operations (MPI_Send and MPI_Recv) with a naive CPU staging approach (without pipeline design) and evaluate the performance with different types of host staging buffer through



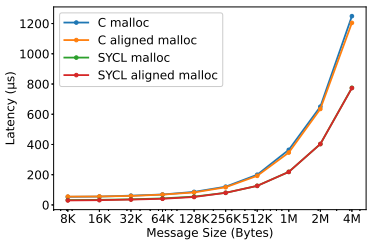

(a) Host to Device - Small Messages




(b) Host to Device - Large Messages

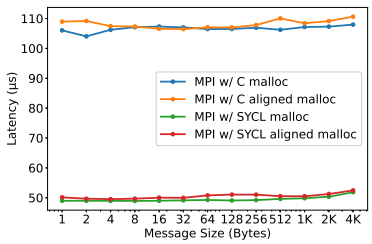



(c) Device to Host - Small Messages

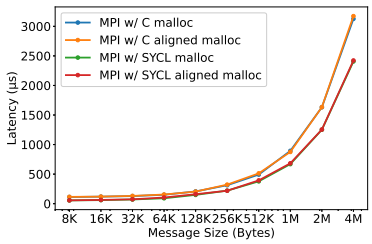



(d) Device to Host - Large Messages

Fig. 4. Comparison of D2H and H2D memcpy latency using different host buffer allocation approaches: malloc and aligned malloc (with alignment set to 4KB) using C and SYCL. The memory copy speed using SYCL malloc is about 2x compared to the memory copy speed using C malloc. It suggests utilizing the SYCL malloc memory space as the host staging buffer.




(a) Small Messages




(b) Large Messages

Fig. 5. Comparison of `osu_latency` using different host buffer allocation approaches: malloc and aligned malloc (with alignment set to 2K) using C and SYCL.

`osu_latency` in modified OMB. We extend the OMB backend to support memory allocation and data validation on the device buffer. This evaluation demonstrates that our proposed implementation can carry out GPU-aware MPI operations and we can utilize different types of host staging buffers in our implementation. Figure 5 shows the latency of our proposed naive implementation with different types of buffers indicated in the previous evaluation. The trend is also very similar to figure 4, which means the latency numbers for using C malloc staging buffer is higher than the ones using SYCL malloc buffer, and there is not much difference between regular memory and aligned memory. The latency of naive GPU-aware MPI implementation using C malloc is 107.3 µs at 16 bytes, which is 2.1x slower compared to 50.0 µs, the one using the SYCL malloc approach. For large messages at 4 MB, the SYCL malloc approach can achieve 1.3x faster latency compared to the C malloc approach from 3126 to 2406.5 µs.

The evaluations show better performance and support the assumption that we should adopt the SYCL malloc approach to allocate host staging buffer. Therefore, we choose to utilize SYCL aligned malloc approach in the following implementations and evaluations.

### *D. Pipeline Threshold selection for Advanced Proposed Designs*

As described in section IV-B, there is an overhead for transferring small messages. Hence, it is unlikely to divide the messages into smaller block sizes and expect highly efficient overlapping and good performance without any restriction. Figure 6 demonstrates the latency for large messages (from 64KB to 4MB) using the advanced pipeline design with different block sizes. We normalize the latency values to the one with block size 1, which indicates no pipeline executing. We see obvious benefits from using our advanced pipeline design. For example, we observe 12.5%, 9.3%, and 5.5% of improvements with block sizes 2, 4, and 8 at 4MB. However, the performance decreases for smaller messages if it uses more pipeline blocks. For instance, we get 8.3% improvement at 512 KB using 2 pipeline blocks, but the latency grows 34% if it adopts block size 8, which is even worse than the baseline. We notice that pipeline design has no benefit if the message size is lower than 64 KB. The overhead becomes extremely large, so we set a threshold at 64 KB and apply the pipeline design techniques after 64 KB.

In the meantime, we also observe the best block size is usually 2 for most message sizes, which suggests that message sizes greater than the 64 KB threshold, we should use 2 blocks to design our advanced pipeline approach.

### *E. Point-to-point Communication Evaluations*

We integrate our findings from the pipeline CPU staging approach and implement a high-performance GPU-aware MPI_Send and MPI_Recv operations, and extend the implementations to MPI_Isend and MPI_Irecv. We also implement a naive approach based on `osu_latency`, `osu_bw` and `osu_bibw` as the baseline to simulate the condition that users do not utilize a GPU-aware library to transfer messages between device buffers. Users have to maintain and operate the host staging buffer by themselves. On the other hand, compile and run the Intel GPU buffer-supported point-to-point benchmarks in OMB using Intel MPI 2021.6 as another baseline. It is the same version that we evaluated for our proposed implementation. Figure 7 shows the latency and bandwidth of the 3 designs. We see the proposed implementation reports as low as

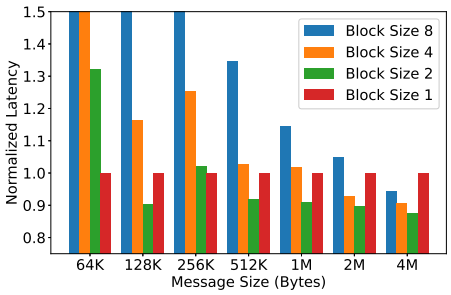



Fig. 6. Normalized latency over different pipeline block sizes. The numbers are normalized to "block size 1" (serial). In most message sizes, block size 2 can deliver the lowest latency. There are no benefits for message sizes less than 64 KB. Note that the figure is clipped at the top at normalized latency of 1.5.

51.0 µs latency at 16 bytes in figure 7(a). Compared to 91.5 µs and 108.0 µs using Intel MPI and naive approach, our proposed implementation achieves 44.2% and 52.8% improvements. For the large message, the proposed implementation continues to stay ahead. Figure 7(d) shows 2196 µs for our proposed implementation compared to 2688 µs and 3168 µs for Intel MPI and naive approach, respectively. In figure 7(b), 7(e), 7(c) and 7(f), we see the bandwidth of proposed implementations is 0.2x to 1.4x higher than the bandwidth of Intel MPI and 1.6x higher for bi-directional bandwidth. In this range, the proposed implementations deliver 1.76 GB/s bandwidth and 1.82 GB/s bi-directional bandwidth at 4 MB. The evaluation results verify that our proposed implementation provides the GPU-aware MPI support for point-to-point operations with the best performance at all message sizes compared against Intel MPI and the naive approach.

### F. Collective Communication Evaluations

We extend our proposed implementation to MPI collectives operations, including MPI_Bcast, MPI_Reduce, MPI_Allreduce, and MPI_Bcast. We evaluate the latency performance of the proposed implementation, Intel oneCCL and Intel MPI on 8 MPI processes over 1 GPU in figure 8. For oneCCL version, we use `ccl::allreduce`, `ccl::reduce`, and `ccl::broadcast` to implement the benchmarks. Note that there is no `ccl::allgather` in Intel oneCCL library, so we just compare the results of 3 collectives. For MPI_Bcast results shown in figure 8(a), our proposed implementation is 2.3x and 4x faster for small messages. The latency is as low as 128.0 µs compared to 299.2 µs and 523.0 µs for Intel MPI and Intel oneCCL at 16 bytes. For larger messages, we see 878.7 µs at 512 KB for our proposed implementation compared to 4981.9 µs and 951.2 µs for Intel MPI and Intel oneCCL. We notice there is a step for Intel MPI at 8 KB, but our proposed implementation depicts a smooth trend there, which is the key to having a much better performance for larger messages. In the meantime, there is no such step or Intel oneCCL, so the latency for large messages is even better than Intel MPI. However, our implementation still performs the best among all sizes and designs. Figure 8(b) shows a similar trend for MPI_Reduce, our proposed implementation achieves 188.9 µs and 1129.5 µs latency at 16 bytes and 512 KB, which is 1.8x and 1.6x faster compared to the 337.8 µs and 1789.4 µs latency for Intel MPI, and 3.6x and 1.5x faster compared to the 671.5 µs and 1638.5 µs latency for Intel oneCCL. We see a similar trend for MPI_Allreduce and MPI_Allgather comparisons in figure 8(c) and 8(d). our proposed implementations have 4x to 8x lower latency for MPI_Allreduce and 5x lower latency for MPI_Allgather.

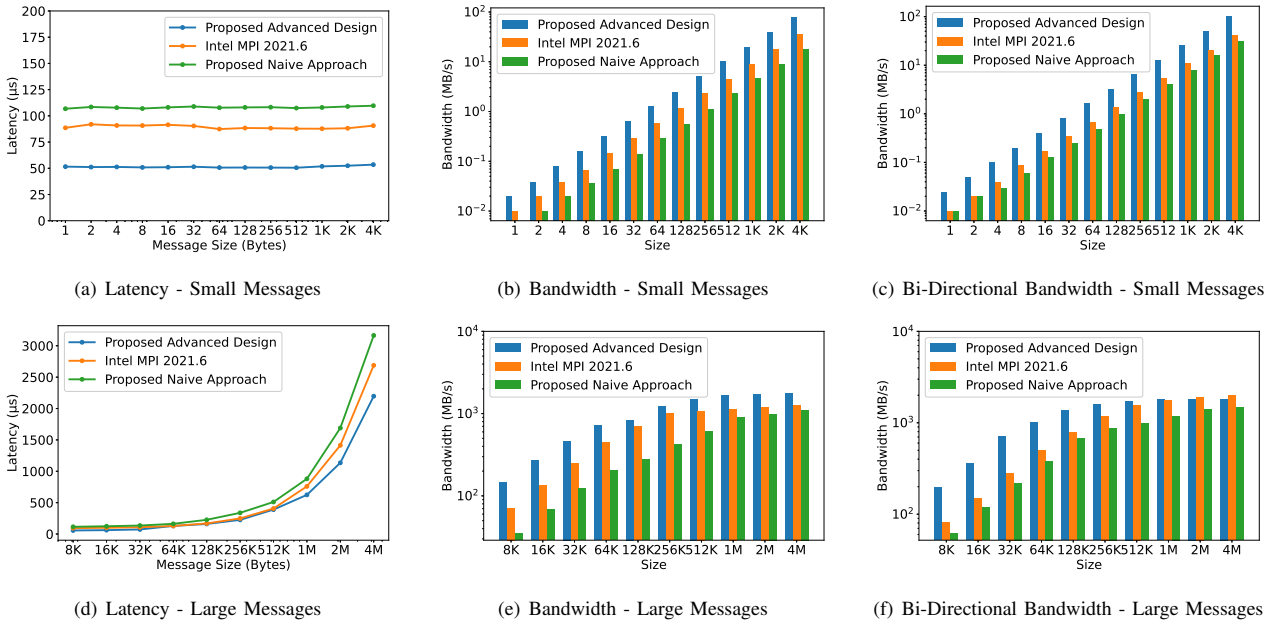
### G. Application-level Evaluations

We evaluate our implementations at the application level with 3DStencil. It uses lots of non-blocking point-to-point MPI_Isend and MPI_Irecv pairs to commute data. Figure 9 demonstrates the performance results using 4 PPN (process per node) and 8 PPN on 1 node. In figure 9(a), the latency of our proposed implementations is 39% better than Intel MPI at 16 bytes, and have 16% improvement at 256 KB. Figure 9(b) shows a trend similar to 8 PPN. The proposed implementations have approximately 40% improvement from 1 byte to 64 KB, and stay ahead up to 256 KB. The trend of this evaluation is quite similar to the bandwidth evaluations in figure 7. That is because both benchmarks consist of many non-blocking point-to-point operations.

## VI. Related Work

Over the past decade, GPUs have gained significant popularity in performing compute-intensive tasks and are widely used in many modern-day clusters. Therefore, it is pertinent to have robust communication between GPUs as well as across GPU and host memory for optimal performance. In the early works, Jacobsen et al. [14] have overlapped GPU data movement and MPI communication with computation to simulate computational fluid dynamics (CFD) using MPI and CUDA. Wang et al. [15] came up with an optimal CUDA-based design to achieve GPU-GPU communication for InfiniBand clusters. Their proposed solution integrated GPU data movement with MPI interfaces and achieved inter-node GPU communication using host-based pipelining techniques. Wang et al. [16] have also proposed novel research to improve data transfers between GPUs in RDMA-enabled clusters without the involvement of the CPU. Potluri et al. [17] utilized the GPUDirect RDMA feature in CUDA to improve MPI-based inter-node GPU-GPU communication. They proposed a hybrid design that incorporated host-based pipelining and GPUDirect RDMA. Subramoni et al. [18] proposed designs capable of dynamically adapting to the communication characteristics of processes at runtime. Their proposed solution can make the transition from one eager threshold to another without any impact on the throughput. Kawthar et al. [19] analyzed the performance of various CUDA-aware MPI libraries such as MVAPICH2-GDR, Spectrum MPI, and Open MPI + UCX by comparing their point-to-point performance.

While most of the research has been focused on NVIDIA GPUs/CUDA, few researchers have made use of AMD GPUs/ROCm. Kuznetsov et al. [20] investigated whether




(a) Latency - Small Messages (b) Bandwidth - Small Messages (c) Bi-Directional Bandwidth - Small Messages

(d) Latency - Large Messages (e) Bandwidth - Large Messages (f) Bi-Directional Bandwidth - Large Messages

Fig. 7. Comparison of MPI point-to-point operations between proposed GPU-aware MPI library, Intel MPI 2021.6, and naive approach using `osu_latency`, `osu_bw`, and `osu_bibw`. The advanced proposed implementations have the lowest latency and highest bandwidth over all ranges of message sizes.

ROCm can be compared to CUDA and described their experiences while porting Classical Molecular Dynamics (MD) applications from CUDA to ROCm using the HIP framework. Kondratyuk et al. [21] analyzed the performance of MD applications on NVIDIA and AMD GPUs. Kawthar et al. [22] designed the state-of-the-art ROCm-aware MPI runtimes for the MVAPICH2-GDR library. Their proposed design demonstrated higher bandwidth when compared to OpenMPI + UCX for inter-node and intra-node communication on AMD GPU clusters.

In the context of SYCL and Intel GPUs, Zhai et al. [23] designed SYCL-based GPU backend for Microsoft SEAL (Homomorphic Encryption (HE) library) APIs. Their proposed solution to optimize the NTT Key algorithm for HE achieved a speedup of 9.93x compared to the naïve GPU implementation. Deakin et al. [24] studied the performance of HPC-based SYCL Applications and compared it with OpenCL and other programming models. Cardoso da Silva et al. [25] compared SYCL with OpenCL and OpenMP and concluded that SYCL's performance was not on par with OpenCL and OpenMP. Kuncham et al. [26] evaluated the performance of SYCL and CUDA on NVIDIA GPUs.

## VII. Conclusion

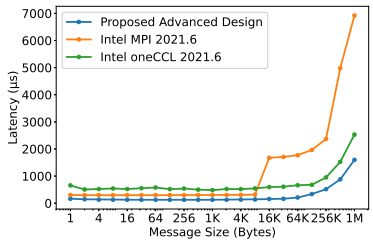
As the upcoming HPC supercomputers are equipped with Intel GPUs and more applications are ported to SYCL-based implementation and run on the next-generation accelerators, it is critical to have an MPI-level middleware to support highly efficient communication for these DL and HPC applications. In the past decade, GPU-aware libraries such as MVAPICH2-GDR and Open MPI have enhanced the support for NVIDIA and AMD GPUs through CUDA and ROCm toolkits. The success of the previous GPU-aware designs triggers the expectation to have support and optimizations for data transfer on Intel GPUs using state-of-the-art MPI libraries. In this paper, we explored the features of Intel GPUs and took up the challenges of implementing and optimizing a GPU-aware MPI library using the oneAPI library. We adopted the CPU staging approach and optimized the memory copy speed and flow by choosing the host staging buffer type and advanced pipeline designs. We evaluated the performance of point-to-point and collective operations with our proposed GPU-aware MPI library at the benchmark level using OMB compared to Intel MPI and the naive approach. We observed about 2x lower latency and 1.4x and 1.6x increased bandwidth and bi-directional bandwidth, respectively. We also demonstrated up to 8x improvement of latency for MPI_Allreduce. In application-level evaluation, our proposed implementations reach up to 40% improvement for 3DStencil. In the future, we intend to continue this work to extend the proposed implementations to more variety of MPI operations and evaluate the performances in multi-GPU and multi-node environment.

## VIII. Acknowledgement

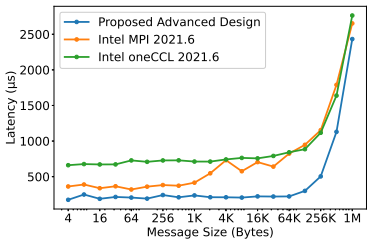
We would like to thank Dr. Sameer Shende (University of Oregon) for providing access to the HPC systems used in the paper.

## References

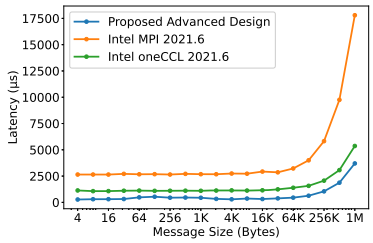
[1] E. Strohmaier, J. Dongarra, H. Simon, and M. Meuer, "TOP 500 Supercomputer Sites," http://www.top500.org, 1993.
[2] Intel, "Intel Iris Xe MAX Graphics," https://www.intel.com/content/www/us/en/products/sku/211013/intel-iris-xe-max-graphics-96-eu/specifications.html.
[3] H. Jiang, "Intel's ponte vecchio gpu : Architecture, systems & software," in *2022 IEEE Hot Chips 34 Symposium (HCS)*, 2022, pp. 1–29.
[4] Khronos, "SYCL 2020 Specification Revision 5," https://registry.khronos.org/SYCL/specs/sycl-2020/html/sycl-2020.html, 2022.



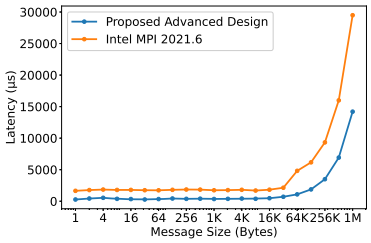

(a) MPI_Bcast




(b) MPI_Reduce

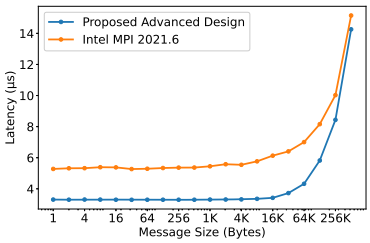



(c) MPI_Allreduce

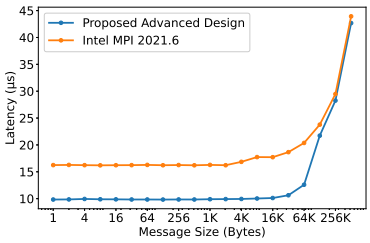



(d) MPI_Allgather

Fig. 8. Comparison of MPI collective operations between proposed GPU-aware MPI library, Intel oneCCL 2021.6 and Intel MPI 2021.6 on 1 node, 8 PPN. The advanced proposed implementations have the lowest latency over all ranges of message sizes in all 4 collective operations.




(a) 3DStencil 4PPN




(b) 3DStencil 8PPN

Fig. 9. Performance comparison of 3DStencil using 4 PPN and 8 PPN on 1 node.


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

[6] Message Passing Interface Forum, "MPI: A message-passing interface standard version 4.0," Jun. 2021. [Online]. Available: https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf

[7] D. K. Panda, H. Subramoni, C.-H. Chu, and M. Bayatpour, "The mvapich project: Transforming research into high-performance mpi library for hpc community," *Journal of Computational Science*, p. 101208, 2020. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1877750320305093

[8] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall, "Open MPI: Goals, concept, and design of a next generation MPI implementation," in *Proceedings, 11th European PVM/MPI Users' Group Meeting*, Budapest, Hungary, September 2004, pp. 97–104.

[9] Intel, "Intel Message Passing Interface (MPI) Library," https://www.intel.com/content/www/us/en/developer/tools/oneapi/mpi-library.html.

[10] Khronos, "OpenCL 3.0 specification," https://registry.khronos.org/OpenCL/specs/3.0-unified/html/OpenCL_API.html, 2022.

[11] NVIDIA, "How to Optimize Data Transfers in CUDA C/C++," https://developer.nvidia.com/blog/how-optimize-data-transfers-cuda-cc/, 2012.

[12] Intel, "FPGA Optimization Guide for Intel® oneAPI Toolkits - Prepinning Memory," https://www.intel.com/content/www/us/en/develop/documentation/oneapi-fpga-optimization-guide/top/optimize-your-design/throughput-1/host/prepinning-memory.html, 2022.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.