# Enabling Reconfigurable HPC through MPI-based Inter-FPGA Communication

Nicholas Contini, Bharath Ramesh, Kaushik Kandadi Suresh, Tu Tran, Ben Michalowicz, Mustafa Abduljabbar, Hari Subramoni, Dhabaleswar Panda
The Ohio State University
The Department of Computer Science and Engineering
Columbus, OH, USA
{contini.26,ramesh.113,kandadisuresh.1,tran.839,michalowicz.2,abduljabbar.1,subramoni.1}@osu.edu
panda@cse.ohio-state.edu

## ABSTRACT

Modern HPC faces new challenges with the slowing of Moore's Law and the end of Dennard Scaling. Traditional computing architectures can no longer be expected to drive today's HPC loads, as shown by the adoption of heterogeneous system design leveraging accelerators such as GPUs and TPUs. Recently, FPGAs have become viable candidates as HPC accelerators. These devices can accelerate workloads by replicating implemented compute units to enable task parallelism, overlapping computation between and within kernels to enable pipeline parallelism, and increasing data locality by sending data directly between compute units. While many solutions for inter-FPGA communication have been presented, these proposed designs generally rely on inter-FPGA networks, unique system setups, and/or the consumption of soft logic resources on the chip. In this paper, we propose an FPGA-aware MPI runtime that avoids such shortcomings. Our MPI implementation does not use any special system setup other than plugging FPGA accelerators into PCIe slots. All communication is orchestrated by the host, utilizing the PCIe interconnect and inter-host network to implement message passing. We propose advanced designs that address data movement challenges and reduce the need for explicit data movement between the device and host (staging) in FPGA applications. We achieve up to **50%** reduction in latency for point-to-point transfers compared to application-level staging.

## CCS CONCEPTS

• **Software and its engineering** → **Message passing**; **Message oriented middleware**; • **Hardware** → **Hardware accelerators**; • **General and reference** → **Performance**; **Experimentation**; • **Networks** → *Network experimentation*; **Programming interfaces**.

## KEYWORDS

MPI, FPGA, heterogeneous computing, OpenCL, distributed computing, high performance computing


*ICS '23, June 21–23, 2023, Orlando, FL, USA*

ACM ISBN 979-8-4007-0056-9/23/06…$15.00
https://doi.org/10.1145/3577193.3593720

**ACM Reference Format:**
Nicholas Contini, Bharath Ramesh, Kaushik Kandadi Suresh, Tu Tran, Ben Michalowicz, Mustafa Abduljabbar, Hari Subramoni, Dhabaleswar Panda. 2023. Enabling Reconfigurable HPC through MPI-based Inter-FPGA Communication. In *International Conference on Supercomputing (ICS '23), June 21–23, 2023, Orlando, FL, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3577193.3593720

## 1 INTRODUCTION

In recent years, we have seen the diversification of HPC hardware. For example, several of the Top500 [29] systems include GPUs, most notably the number one system, Frontier. They excel at tackling "embarrassingly parallel" compute tasks due to the hardware's focus on simpler design with massive multi-threading capabilities in contrast to the design of a CPU that targets much more complex logic flow. Some accelerators are even designed with specific applications in mind; ASICs such as Tensor Processing Units are available through Google Cloud infrastructure expressly for accelerating Deep Learning workloads [10]. Data-Processing Units (DPU) are one of the latest attempts at specialized hardware, enabling the ability to offload certain communication-based tasks to free the CPU for more compute-intensive tasks.

These advances are spurred by the slowing of Moore's Law and Dennard Scaling. Computational scientists can no longer expect new hardware to double in efficiency and performance every few years. Co-design between hardware designers, system engineers, and application developers will be vital in the coming years. Considering this, FPGAs and other reconfigurable hardware are becoming very attractive options in the HPC space. With the rising demand for specialized hardware, these devices enable rapid development of hardware logic. Furthermore, they excel at tackling computational tasks that exhibit more irregular forms of parallelism that other architectures such as GPUs would be hindered by [24]. For example, implementing Fast Fourier Transform on a CPU or GPU would require many transfers between registers and larger memory systems further away from the computational units. An FPGA can retain high data locality by feeding the output of one block of soft logic into the next. FPGAs also have also made their way into green supercomputing spaces, with an FPGA-based system, ENIAD, taking the number 10 spot on the Graph 500 Green list for small datasets and the number 9 spot for big datasets [6].

One of the biggest hurdles preventing the adoption of these devices from accelerating modern HPC workloads is the difficulty of development; FPGAs were traditionally used for hardware prototyping, many of the programming abstractions differing significantly

from software development abstractions. For example, developers were previously limited to programming FPGAs using hardware description languages such as VHDL and Verilog which even hardware engineers found challenging to use. When using these languages, many concepts used in software do not exist, such as requesting memory from an operating system as one would in C/C++. These languages to this day are still very challenging for software developers to use. Luckily, many advances have been made to decrease development overhead. Vendors have created robust High-Level Synthesis workflows, enabling software developers to create efficient and performant designs using familiar C/C++, OpenCL, and Python interfaces[16, 24, 27]. However, there is still much work to be done in aiding the development of FPGA-based HPC applications. For example, MPI is a vital component of HPC applications, as it is considered the defacto programming model in HPC. It enables scaling computation beyond a single compute node, possibly scaling an application across thousands of nodes within a cluster. As of now, there are no available FPGA-aware MPI implementations. Without FPGA-to-FPGA support within MPI, many traditional HPC applications will have to be rewritten in order to utilize FPGAs effectively.

Previous attempts have been made to utilize MPI with reconfigurable hardware [4, 5, 13, 14, 20]; however, the resulting designs either focused on using FPGAs as independent nodes, in-network devices, or using inter-FPGA networks. While these setups can provide advantages, adding requirements at a cluster level may discourage adoption. Another issue with existing approaches is that they use some of the soft logic resources on the FPGA for implementing custom circuits, taking away resources that can potentially be used for computation. Although these designs may take minimal space, any use of FPGA resources may prevent an optimal placement of logic used for actual computation. **In this paper, we propose an MPI runtime that enables applications to initiate optimal data transfers between FPGA devices without the need to explicitly transfer data between the host and the FPGA using OpenCL APIs and Xilinx OpenCL extensions.**

```
if (rank == 0) {
    clEnqueueReadBuffer(command_queue, fpga_buffer, CL_TRUE, 0,
      BUFFER_SIZE, send_buffer, 0, NULL, NULL);
    MPI_Send(send_buffer, BUFFER_SIZE, MPI_BYTE, 1, tag1,
      MPI_COMM_WORLD);
} else {
    MPI_Recv(recv_buffer, BUFFER_SIZE, MPI_BYTE, 0, tag2,
      MPI_COMM_WORLD, &status);
    clEnqueueWriteBuffer(command_queue, fpga_buffer, CL_TRUE, 0,
      BUFFER_SIZE, recv_buffer, 0, NULL, NULL);
}
```

**Listing 1: A naive implementation of inter-FPGA communication that achieves high productivity but may not be optimal.**

## 1.1 Motivation

Putting the burden of inter-FPGA communication on the application developer can lead to suboptimal performance and/or low productivity. For example, consider the naive approach presented in Listing 1 demonstrating a basic send and receive between two ranks. In this code snippet, the sending rank first transfers data from FPGA memory to host memory by calling `clEnqueueReadBuffer` before calling `MPI_Send`. The receiving rank calls `MPI_Recv` and then uses `clEnqueueWriteBuffer` to transfer the received data to the FPGA. While this can easily be implemented by most software developers, it is suboptimal. Consider Figure 1, for example. This breaks the overall communication into FPGA-to-host, host-to-host, and host-to-FPGA transfers and spreads them on a timeline. Since host-to-FPGA and host-to-host transfers use independent resources on the system, they can be run simultaneously, meaning the portion of the D-H transfer corresponding to timestamp 1 can be overlapped with the H-H transfer corresponding to timestamp 3. The same relationship exists between H-H and H-D transfers; the H-H transfer occurring at timestamp 4 can be overlapped with the H-D transfer at timestamp 6.

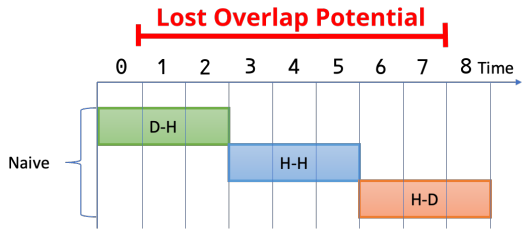



**Figure 1: Diagram showing the lost potential overlap of data transfer.**

Now consider a more optimized approach to inter-FPGA transfers in Listing 2. In this snippet the sending rank initiates multiple non-blocking `clEnqueueReadBuffer` calls. It then waits on an event corresponding to each of these commands. When the event is marked complete, the data is guaranteed to exist within host memory and thus the sender calls `MPI_Isend` before waiting on the next chunk of data to arrive. Once all the `MPI_Isend` commands are issued, `MPI_Waitall` is called to ensure each communication completes. The receiving rank executes similar code, except several calls of `MPI_Irecv` are made first. As each `MPI_Irecv` completes, a `clEnqueueWriteBuffer` call is made to move the received data to FPGA memory. After each device write is enqueued, `clWaitForEvents` is called on the corresponding events to ensure data has reached the FPGA. This takes advantage of the potential overlap presented in Figure 1, but it is significantly more verbose than the naive approach presented above. Code this complex lowers productivity and is prone to errors. Furthermore, even though extra work was put into writing it, this code still might not be the most optimal solution for every given scenario. As technologies mature more advanced forms of data transfer may be available, rendering this code outdated. Constant refactoring of code to keep up with the state of the art further limits productivity.

By creating an FPGA-aware MPI, application developers can easily achieve the best of both worlds. MPI developers can integrate optimizations such as the one presented above into the runtime, removing any need for explicit data transfer between the host and FPGA at the application level. This maximizes productivity while achieving good performance. Furthermore, instead of several HPC applications requiring refactoring to keep up with new optimizations, only MPI implementations need to update their source.

### 1.2 Contributions

Our implementation will allow developers to easily adopt reconfigurable hardware implementations into existing HPC applications. By keeping communication host-driven, our implementation can build upon decades of optimizing point-to-point and collective communication both in the context of CPU-based and accelerator-based computation. Traditional HPC applications will not have to rewrite their MPI code to stage data onto FPGA accelerators, nor will future HPC clusters be forced to be designed any differently from how they are designed today other than installing the FPGA accelerator into a node. Furthermore, no resources on the FPGA itself are used for the implementation, meaning all resources on the FPGA can be used for computation and no FPGA shell or kernel must be integrated with the application kernels. For this study we implement our designs on an InfiniBand cluster. However, our findings can be reapplied to other higher performance networks.

```
cl_event events[BUFFER_SIZE / CHUNK_SIZE];
MPI_Request requests[BUFFER_SIZE / CHUNK_SIZE];
MPI_Status statuses[BUFFER_SIZE / CHUNK_SIZE];
if (rank = 0) {
    for (int i = 0; i < BUFFER_SIZE / CHUNK_SIZE; ++i) {
        size_t size = CHUNK_SIZE;
        if (CHUNK_SIZE * (i + 1) > BUFFER_SIZE) {
            size = BUFFER_SIZE T % CHUNK_SIZE;
        }
        clEnqueueReadBuffer(command_queue,  fpga_buffer, CL_FALSE,
      i * CHUNK_SIZE,  size, ((char *)send_buffer) +  CHUNK_SIZE *
      i, 0, NULL, &events[i]);
    }
    for (int i = 0; i < BUFFER_SIZE / CHUNK_SIZE; ++i) {
        size_t size = CHUNK_SIZE;
        if (CHUNK_SIZE * (i + 1) > BUFFER_SIZE) {
            size = BUFFER_SIZE T % CHUNK_SIZE;
        }
        clWaitForEvents(1, &events[i]);
        MPI_Isend(((char *)send_buffer) + CHUNK_SIZE * i, size,
     MPI_BYTE, 1, tag1, MPI_COMM_WORLD, &requests[i]);
    }
    MPI_Waitall(BUFFER_SIZE / CHUNK_SIZE, requests, statuses);
} else {
    for (int i = 0; i < BUFFER_SIZE / CHUNK_SIZE; ++i) {
        size_t size = CHUNK_SIZE;
        if (CHUNK_SIZE * (i + 1) > BUFFER_SIZE) {
            size = BUFFER_SIZE T % CHUNK_SIZE;
        }
        MPI_Irecv(((char *)recv_buffer) + CHUNK_SIZE * i, size,
     MPI_BYTE, 1, tag1, MPI_COMM_WORLD, &request[i]);
    }
    for (int i = 0; i < BUFFER_SIZE / CHUNK_SIZE; ++i) {
        size_t size = CHUNK_SIZE;
        if (CHUNK_SIZE * (i + 1) > BUFFER_SIZE) {
            size = BUFFER_SIZE T % CHUNK_SIZE;
        }
        MPI_Wait(&requests[i], &statuses[i]);
        clEnqueueWriteBuffer(command_queue, fpga_buffer, CL_FALSE,
     i * CHUNK_SIZE, size, ((char *)recv_buffer) + CHUNK_SIZE * i
     , 0, NULL, &events[i]);
    }
    clWaitForEvents(BUFFER_SIZE / CHUNK_SIZE, events);
}
```

**Listing 2: A pipelined implementation of inter-FPGA communication that potentially is optimal but at the loss of productivity**

In this paper we contribute the following:

(1) Identify a paradigm for inter-FPGA communication that requires minimal coding effort and therefore increases productivity.
(2) An evaluation of the effect of mapping OpenCL buffers on FPGA transfer and MPI latency.
(3) An analysis of P2P (PCIe peer-to-peer) functionality made available through the Xilinx Runtime.
(4) Identify bottlenecks and potential areas of overlap for inter-FPGA communication operations.
(5) Propose an FPGA-aware MPI runtime with optimizations for intranode and internode FPGA-to-FPGA communication.

We compare our designs against paradigms using explicit staging calls to implement inter-FPGA communication. Our proposed designs achieve a **25%** improvement in intranode point-to-point latency for message sizes less than 16KB and a **33%** and **50%** improvement in intranode and internode point-to-point latency for message sizes above 2MB.

## 2 BACKGROUND

### 2.1 The Message Passing Interface

The Message Passing Interface (MPI) is the de-facto standard used for communication, enabling applications to run on the world's fastest HPC clusters [15]. Multiple MPI processes can be distributed amongst multiple cores within a single machine as well as amongst machines connected through a network. By providing a generic interface for communication, application developers can avoid interfacing with low-level communication mediums such as Ethernet, InfiniBand, up-and-coming network interconnects such as Slingshot, shared memory, etc. The MPI standard has been adapted and modified over the years to keep up with ever-evolving hardware, from host processors and software tools to accelerators such as GPUS and FPGAs. MPI implementations such as OpenMPI [19] and MVAPICH2 [11] have enabled direct accelerator-to-accelerator communication by integrating vendor-provided APIs such as CUDA and HIP. For example, the authors of [30] presented GPU-aware designs over MVAPICH2 such that application developers can avoid manually copying data to/from GPU memory. SmartNICs, such as Bluefield DPUs, have also gained attention from MPI implementations [1, 25].

### 2.2 FPGAs

Field Programmable Gate Arrays (FPGA) are traditionally classified as "reconfigurable hardware." Traditionally, these devices are used for hardware prototyping. A circuit can be designed using a hardware design language, flashed onto an FPGA development board, and then tested. When flashed, portions of the circuit are mapped to different components on an FPGA. These are:

1) Logic/soft blocks: Units of reconfigurable hardware built using hardware lookup tables (LUTs), registers, and multiplexers. Recently DSP units have been added to FPGAs to enable better floating-point computation performance.

2) Hard blocks: Static circuits that cannot be reconfigured but can execute logic more efficiently than soft blocks. These blocks are interconnects, controllers, or even embedded processors.

3) Configurable buses: these buses are configurable using several multiplexers, allowing flexible routing of output from one component to many other potential components.

It is vital to be cautious when using these resources on FPGAs. Using too many of one resource can cause issues with placement

of logic. For example, two soft blocks may not be able to be located closely together on the FPGA if too many soft blocks are already consumed. This may lead to lower clock rates as the circuit's critical path may have to be lengthened to accommodate the poor locality of logical components. Once issues with the design are identified, the code can be altered, reflashed, and retested until a design that is ready to be manufactured is realized.

The flexibility of the FPGA makes them very attractive. Instead of being restricted to the fetch, decode, execute, and store paradigm of common processors, FPGAs can be configured to custom circuits. One of the main deterrents for adopting these devices in the HPC community and beyond was the lack of familiarity with the traditional FPGA development languages such as VHDL and Verilog. These hardware designs languages differ greatly from the software programming languages used for HPC applications. However, the research community and vendors have long worked at creating a higher-level approach to program FPGAs called High Level Synthesis (HLS). HLS programming interfaces allow software developers to create hardware designs using more familiar languages such as C, C++, OpenCL, and Python. These advances have greatly lowered barriers preventing software developers from considering FPGAs as viable accelerators. As FPGAs become easier to use, this enables hardware co-design with applications

FPGAs are also known to excel at pipelineable/parallelizable code. Instead of moving data between computational units (CUs) and memory, data can be fed directly from one CU to another, increasing data locality. These different CUs can execute concurrently; once one CU completes a calculation that other CUs depend on, it can continue to process the next portion of input data while other CUs process its outputted data.

## 2.3 OpenCL

The Open Computing Language (OpenCL) standard is a cross-platform parallel programming model for accelerators in various capacities [18]. Like the MPI standard, the OpenCL standard provides a plethora of memory objects in addition to specifications for shared virtual memory, ordering rules, and consistency models for backward compatibility with prior releases. OpenCL is a major force in allowing FPGAs to be integrated into data centers and HPC clusters and is vendor-agnostic; through this, developers are able to write vendor-agnostic or vendor-specific codes as needed. Our designs (described in Section 3), rely on the use of OpenCL for queues to store various send/receive-related objects as well as being able to map/unmap buffers as needed..

## 2.4 The Xilinx Runtime

The Xilinx Runtime (XRT) is a low-level runtime and API for interacting with Xilinx FPGA accelerator cards. XRT provides a low-level native API and an implementation of the OpenCL standard (XCL). Within XCL, extensions are implemented to provide advanced functionality for which the OpenCL standard does not provide interfaces. These functionalities include the ability to allocate device memory in specific global memory banks and translation from OpenCL handles to native XRT handles.

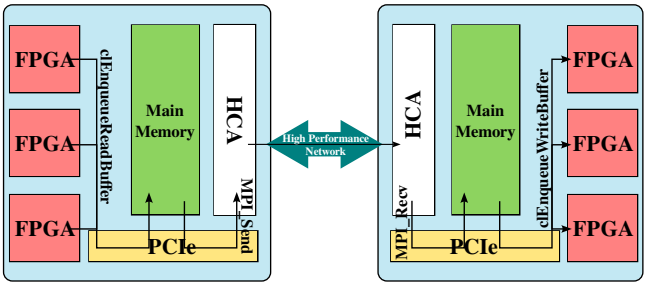
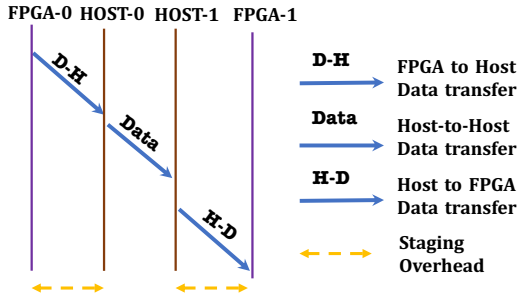
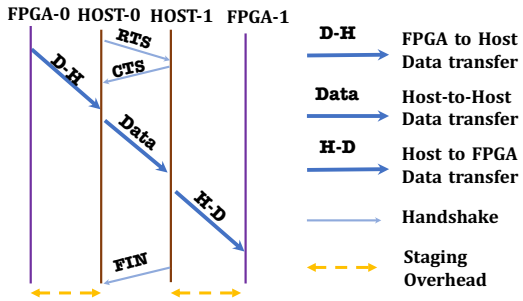



**Figure 2: Diagram showing different stages of FPGA to FPGA MPI transfers.**




**(a) Eager protocol. The lack of a handshake phase minimizes latency as copies incur little overhead at small message sizes.**




**(b) Rendezvous protocol. For larger message sizes, a handshake phase allows the receiver to prepare a receive buffer to avoid unnecessary copies.**

**Figure 3: Demonstration of Eager and Rendezvous protocols. The red arrows demonstrate the extra staging calls needed to make them FPGA-aware.**

# 3 DESIGN

Currently, FPGA application developers are expected to explicitly manage data movement between host and device, which we refer to as staging. OpenCL provides two different interfaces to stage data:

1) **Explicit data transfers**, calls to `clEnqueueWriteBuffer`, `clEnqueueReadBuffer`, or `clEnqueueMigrateMemObject` are made. Any one of these will submit a data transfer request to the OpenCL queue. Afterward, any other entries in the queue depending on this data movement must wait on an event object corresponding to the transfer. These transfers require the CPU to be involved.

2) **Implicit data transfers**, where a device buffer is mapped to host memory using `clEnqueueMapBuffer`. Depending on the OpenCL implementation, this may cause host memory to be pinned, preventing it from being "paged out." Pinning memory lets DMA engines coordinate the data transfer, thus minimizing the CPU's involvement. Once the device buffer is mapped, the host may interact with the corresponding host memory using loads and stores. Then the application can call one of `clEnqueueWriteBuffer`, `clEnqueueReadBuffer`, or `clEnqueueMigrateMemObjects` to migrate data to/from the device explicitly. Alternatively, `clEnqueueUnmapMemObject` can be called to release the mapped host buffer, potentially triggering data movement to the device in the event of writes on the host.

Our design removes the need for any application-level data movement between the host and device and instead opts to handle this within the MPI runtime itself. The first task we tackle is how to identify device buffers. We focus on the second method of staging described above as this best fits within MPI semantics. Our FPGA-aware MPI implementation wraps calls to `clCreateBuffer` and `clEnqueueMapBuffer` in order to store the OpenCL buffer handle and virtual address pair created. Later when the application calls an MPI primitive, the implementation checks to see if the send/receive buffer address corresponds to a device buffer, prompting it to invoke device-specific protocols to transfer the data from one device to another. This transfers the responsibility of optimizing inter-FPGA data movement from the application developers to the MPI developers. Furthermore, as more optimizations are added to our implementation, application developers simply must install a new version of the implementation to reap the benefits. The following subsections will describe our designs for inter-FPGA point-to-point communication.

## 3.1 One-Shot Staged Design

Our base design for intranode transfers utilizes shared memory as a communication medium. We utilize an Eager protocol for small messages which is visualized in Figure 3a. No handshake is executed between the communicating processes in the Eager protocol, and the sending process starts transferring data immediately. Once the receiver has posted a receive request and the corresponding incoming data has been transferred, the data can be directly moved into application buffers. To enable these transfers between FPGAs, data is copied directly from FPGA buffers to shared memory. Once the receiver matches its request to the sent data, a final copy is executed from either shared memory or a temporary host buffer to a device buffer.

A Rendezvous protocol is implemented for larger messages (see Figure 3b). Copying larger messages incurs significant overhead, and the rendezvous protocol mitigates this by instead executing a handshake between processes to bypass copies and send/receive data directly to the application-level buffers. The handshake first starts with a Ready-to-Send (RTS) packet. Once the receiver has posted its receive request, it will respond with a Clear-to-Send (CTS) packet. Data can then freely be transferred to completion, upon which the receiver will send a Finished (FIN) signal, thus semantically allowing the send buffer to be written to and the receive buffer to be read. In order to enable FPGA communication within this framework, we have the sending process transfer data from device memory to host memory after sending the RTS, overlapping the Rendezvous protocol with device-to-host transfers. Upon receiving the CTS, the sender copies the data into shared memory. The receiving process then transfers the data from shared memory to device memory and sends a FIN, completing the point-to-point operation.

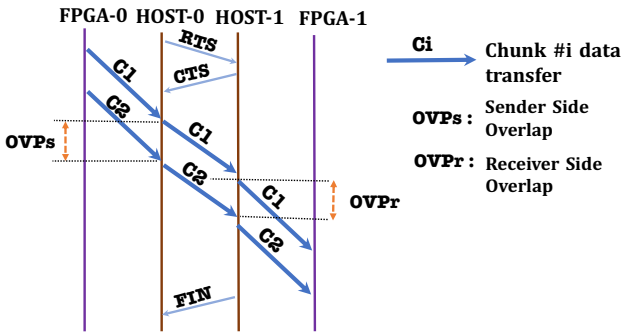
When the two FPGAs are on separate nodes, shared memory no longer can be used as the communication medium. In this case, we take advantage of the InfiniBand network. Copies into shared memory are instead replaced with Remote Direct Memory Access (RDMA) operations. These operations allow one process to send data through the network and directly place data into another process' memory space. This requires that two processes exchange "keys" that grant access to a specified buffer. The process that owns the buffer first generates local and remote keys (lkey and rkey) corresponding to that process. The rkey is then transferred to another process, which then uses this rkey in every read/write operation. In our designs, these rkeys are exchanged during `MPI_Init`. When transferring an Eager message over InfiniBand, the sending process puts the message directly into preallocated memory held by the receiving process. This preallocated memory is created during `MPI_Init`. The receiving process then copies this data into an application buffer. This incurs an extra copy, but since this protocol is used at lower message sizes, the overhead is negligible compared to exchanging an RTS and CTS in the Rendezvous protocol. To add support for FPGA buffers, a device-to-host transfer is executed before the RDMA "put," and a host-to-device buffer transfer is executed from the preallocated memory on the receiver.

At larger message sizes, copies become more expensive, and thus it is beneficial to utilize the Rendezvous protocol to avoid extra copies. This is done by attaching a buffer address to the RTS signal to allow the receiving process to execute an RDMA get directly into an application buffer or attaching a buffer address to the CTS to allow the sending process to execute an RDMA put directly into an application buffer. To support transfers from and to FPGA buffers, device-to-host transfers are made while the sending process waits for the CTS and host-to-device transfers are made once data is received, similar to our scheme in the shared memory design.

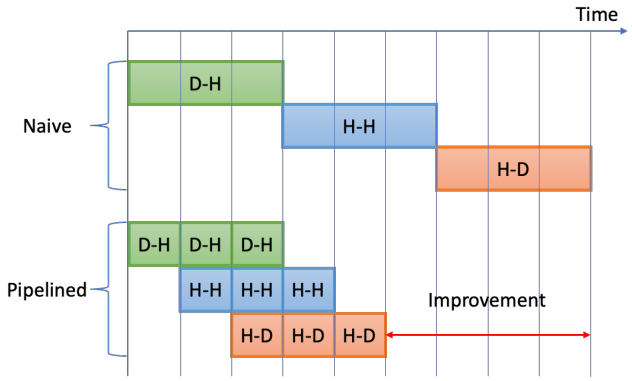
These designs have some shortcomings. For example, whenever staging occurs, the shared memory or InfiniBand are completely unused. This means that bandwidth is not being maximized. More optimized designs will saturate the bandwidth available on the PCIe interface and the transport simultaneously. Furthermore, intranode transfers use a redundant copy. Two devices both connected over PCIe should theoretically be able to transfer data between each other without first sending the data to host memory. This removes some overhead as well as frees up the CPU to work on other tasks. We explore this in the next section.

## 3.2 Internode Pipelined Design

There are multiple stages within point-to-point inter-FPGA communication: the staging calls and the transfer between hosts. These transfers use separate resources, the staging using the PCIe interface and the host-to-host transfer using either shared memory or InfiniBand. These operations can happen simultaneously, so long




(a) Stages of the pipelined design




(b) A display of the achieved improvement.

**Figure 4: Visual representations of the pipelined design. By overlapping different stages of the overall transfer, overall latency is reduced.**

as they operate on different portions of the message. In this section, we discuss the requirements of creating a pipelined design for larger messages.

By breaking down one large message into smaller chunks, we can divide the entire communication process into separate pipeline stages and overlap each of the stages to reduce overall communication latency. In our pipelined design there are three stages: device-to-host transfer, a send via the inter-node network, and a host-to-device transfer. Benefits can only be seen when staging overheads are equal to or less than the host to host transfers. This means that a pipelined design will only work for large message sizes. The benefit of the pipelined design is demonstrated in Figure 4b. We decide to utilize this design for internode transfers as the latency of InfiniBand transfers is higher than a shared memory transfer. This allows staging operations to overlap the host-to-host communication more effectively. Furthermore, we discuss a separate optimization for intranode transfers in the coming section.

## 3.3 Intranode Peer-to-Peer Design

A regular inter-FPGA transfer generally requires the application to first make a call to transfer from the source device's global memory into the host's main memory. A separate call to move the data from the main memory into the destination device's global memory would then be made. Xilinx provides an extension to the OpenCL

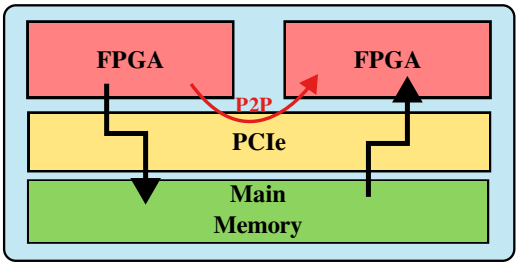
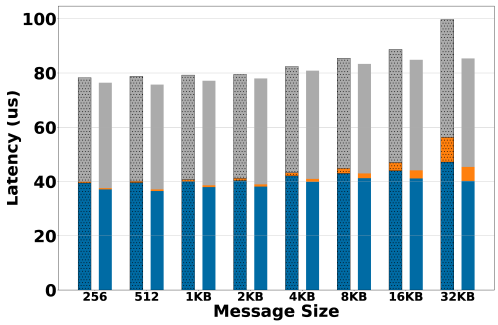



**Figure 5: A diagram demonstrating the path of a standard inter-FPGA transfer within a single node vs using P2P. Using P2P reduces the number of copies by one.**

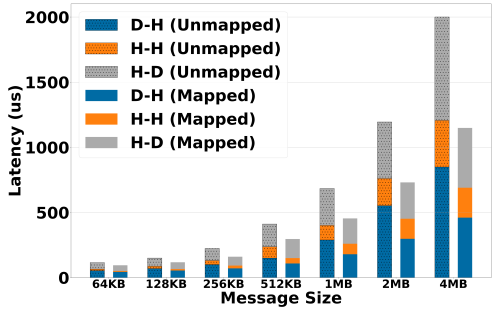
interface that includes a feature to execute Peer-to-Peer (P2P) transfers between two FPGAs connected to the same host. This enables callers to avoid transferring data to host memory by instead writing to PCIe BAR memory. Figure 5 compares the standard data path for an FPGA to FPGA transfer vs. using P2P. Ideally, this feature would be used between two FPGAs located on the same PCIe switch. However, in our testing, we saw the benefits of using these features under the unideal system configurations where the FPGAs are located under separate PCIe roots when transferring large messages.

Executing P2P transfers between two processes requires some initialization. First, one of the two buffers involved in the data transfer must be created using the `XCL_MEM_EXT_P2P_BUFFER` flag in Xilinx's OpenCL memory extension. The buffer then must be exported using a call to `xclGetMemObjectFd`. This function returns a file descriptor corresponding to the input buffer. This file descriptor must then be transferred to the other process. Since the file descriptor is simply an integer that is only valid under the context of the originating process, special steps must be taken to generate a valid file descriptor for the receiving process. We achieve this through the use of Unix Domain Sockets (UDS). After the second process receives the file descriptor, it imports the originating process's buffer's handle by calling `xclGetMemObjectFromFd` on the received file descriptor. After this setup is complete, the second process is free to call `clEnqueueCopyBuffer` using the imported handle and any other buffer handle. The second process's buffer handle does not have to be flagged as a P2P buffer.

We propose a protocol that utilizes P2P data transfers to enhance intranode point-to-point transfers between FPGAs. We use the Rendezvous protocol as a base for our design. To integrate P2P within this framework, we have the sending process attach the exported file descriptor and an offset to the RTS. It is important to note that this descriptor is not valid to the receiving process, and thus only serves as an identifier. We will refer to this descriptor as the identifying file descriptor (IFD). Upon receiving the RTS, if the receiver does not recognize the received IFD, it must go through P2P initialization. The receiver will then read the exported file descriptor through a UDS that is set up within `MPI_Init` between each process local to a node. The receiving process then imports the buffer handle corresponding to the sending process, storing the imported buffer handle in a cache using the IFD as a key. Subsequent transfers utilizing the same buffer can skip the initialization step using this




**(a) For message size up to 8KB, mapped buffers show neither an advantage nor disadvantage.**




**(b) For messages sizes greater than 64KB, mapped buffers exhibit greater advantages and reduce both staging and MPI latency.**

**Figure 6: A comparison of using mapped and unmapped buffers when using MPI for inter-FPGA communication on a single node.**

cached handle. After retrieving the imported buffer handle the receiving process sends back a CTS and executes an asynchronous `clEnqueueCopyBuffer`. Once the copy is complete, the receiver sends its FIN to complete the rendezvous transfer.

# 4 EXPERIMENTS AND EVALUATION

In this section, we discuss the systems/hardware used for our experiments, the experiments run, and our evaluation of them.

## 4.1 Experimental Systems and Software

Our primary system is comprised of dual-socket nodes featuring AMD Milan 7713 processors (64 cores) at 2.00GHz with Mellanox 100/200 HDR. We utilize 16 nodes containing three Xilinx Alveo U280 cards each. In addition to 32 GiB of DDR RAM on each FPGA, the FPGAs are equipped with 8 GiB of HBM2 memory. For intranode tests, including our analysis of P2P buffers, we utilize a system with a single node containing two AMD EPYC 7713 processors with 64 cores each at 2GHz and two Xilinx Alveo U200 cards. These FPGA cards have 32 GiB of DDR RAM each.

We have implemented our proposed designs in an MPI library. We tune our MPI implementation to utilize the optimal design for a given message range. We find that our One-Shot designs perform the best at message sizes less than message sizes less than 2MB and our P2P and pipelined designs are best at larger message sizes for intranode and internode respectively. We refer to this final implementation as "Proposed". Furthermore we use an adapted version of OSU Micro Benchmarks (OMB) [12] to collect our point-to-point latencies. This version is able to allocate FPGA buffers. We use this benchmark to compare our proposed design with implicit staging against "Application Level Staging", which utilizes explicit staging calls in conjunction with a host-based MPI.

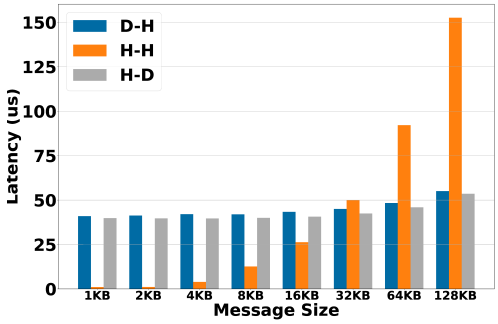
We do not compare against ACCL, an existing collective communication library, due to the fact that it utilizes a TCP/IP transport over an inter-FPGA network. The emphasis of these projects is to provide a communication runtime that works in the more common use case where traditional high-performance networks are utilized, but a supplementary dedicated FPGA network is not in place. In these contexts, ACCL is not applicable.

## 4.2 Experiments and Results

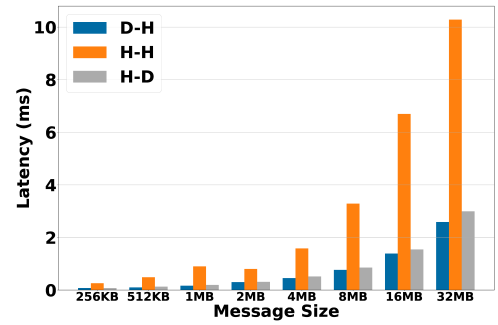
The following subsections are as follows. In section 4.2.1 we do an analysis of the benefits of using device buffers mapped to host memory space. In section 4.2.2 we break down the internode transfer latency into its parts to determine an optimal chunk size for our proposed pipelined design. In section 4.2.3 we examine the effects of P2P buffers on intranode point-to-point communication. Finally, in section 4.2.4 we compare the performance of inter-FPGA communication using application-level staging and our proposed design.

*4.2.1 Performance of Mapped Buffers.* This section quantifies the benefit of mapping device buffers to host memory and the effect of MPI performance. For this experiment, we design a benchmark that uses application-level device transfer in conjunction with MPI point-to-point calls to implement inter-FPGA communication. There are two processes, Rank 0, which will first send data and then receive data, and Rank 1, which will do the same in reverse order. The benchmark can optionally map the host buffers used for the transfer. We measure device-to-host, host-to-host, and host-to-device latency within a single node. Figures 6a and 6b show the average observed half round-trip latency of each portion of the inter-FPGA transfer for both unmapped and mapped buffers across 5 back-to-back calls of the benchmark. At the lowest message sizes, transfers between the device and host in both directions seem to perform similarly regardless of whether the device buffer is mapped. However, starting at message size 32KB and above, mapped buffers have a distinct advantage in latency. More interestingly, we see that the host-to-host transfers also perform better with mapped buffers. This emphasizes that mapping host memory to device buffers is not an optimization exclusively affecting the OpenCL runtime performance, but also parts of the application. In light of this, we used mapped buffers for the rest of our experimentation.

*4.2.2 Optimizing Pipelining Chunk Size.* In order to implement an effective design that pipelines transfers between the device and host and InfiniBand transfers, we must first observe how each part of the inter-FPGA transfer contributes to the overall latency. Using the same benchmark as before, we measure each part of the inter-FPGA transfer, each FPGA on a separate node. We present




**(a) For message sizes up to 128KB the overall latency is dominated by the staging calls.**




**(b) For message sizes above 256KB the host to host latency starts to overtake the staging latency.**

**Figure 7: Point-to-point latency broken down into device-to-host, host-to-host, and device-to-host segments**

the results of our experiment in Figure 7. At lower message sizes, the device-to-host and host-to-device transfers contribute the most to the total transfer latency. This is attributed to the PCIe interface and the device drivers. Solutions to reducing the overhead of these small transfers likely require either advanced interconnect technology or advanced transfer interfaces. For example, CXL and CCIX interconnect technologies enable cache coherence to connected devices as well as unified memory spaces between host(s) and devices. Furthermore, NVIDIA has provided protocols such as GPUDirect RDMA, GDRCopy, and CUDA IPC, all of which have been used to decrease the latency of transfers between host and device.

As message sizes grow, the host-to-host transfer latency becomes a larger proportion of the overall transfer latency. Although transfers over PCIe have a higher base latency, since later generations of the interconnect can achieve higher bandwidth than InfiniBand, it scales better. At 32KB, the host to host latency becomes larger than a transfer between device and host. This indicates pipelining can feasibly be implemented at message sizes greater than 32KB since the device-to-host can be theoretically completely overlapped with host-to-host transfers, which can be completely overlapped by host-to-device transfers. To determine the optimal chunk size for our design, we implement our proposed pipelining-based design at the MPI level. We test using chunk sizes of 32KB to 2MB, displaying the results in Figure 8. We only present results for message sizes 1MB and above since the our Eager design performs better at message sizes below that threshold. Using certain chunk sizes, especially 32KB and 64KB, result in worse performance than the baseline. This is because the overhead of orchestrating the pipeline is not offset by the overlap between the stages of communication. However, chunk sizes greater than 128KB generally produce better performance than the baseline. While there is no single chunk size that always produces the lowest latency, using a chunk size of 1MB performs the best on average.

*4.2.3 Effects of P2P Buffers.* In our testing, we find that using the `XCL_MEM_EXT_P2P_BUFFER` flag can have a significant impact on point-to-point performance, even when the P2P protocol is not being used to do the data transfer. We hypothesize that mapping BAR may have effects on the cache. To verify this, we utilize our benchmark with application-level staging in conjunction with PAPI. PAPI is an analysis tool that gives applications an easy-to-use interface to various hardware counters. The application registers which events the it would like to count, the metadata stored within an array of integers referred to as the event set. When the application wants to actually start counting events, it passes the event set into `PAPI_start`. Counting stops when the event set is passed into `PAPI_stop`, which outputs an array of counts. To support our hypothesis, we measure the average number of L1 and L2 cache misses for each message size. We place `PAPI_start` and `PAPI_stop` before and after measuring the latency of inter-FPGA transfers for each message sizes and present the results in Figure 9. At lower message sizes, the number of cache misses is similar between the test with P2P buffers and without. Once the message size grows to 32KB, a notable divergence can be seen. Tests using P2P buffers start to experience more cache misses relative to tests using non-P2P buffers. This difference grows to it's largest difference, the tests with P2P buffers suffering from 5 times more L1 cache misses. At message sizes of 2MB and above the number of cache misses between the two scenarios converges. This suggests that the negative effects of P2P buffers is negligible at this message size, thus we take these considerations when using our design.

One possible explanation for the increased cache misses could be that mapping the BAR leads to cache pollution. When the BAR is not mapped to host memory, data read and written to the device does not enter the cache hierarchy. When the BAR is mapped, reads and writes now occur within the host's memory space and thus the cache is populated with entries corresponding to these values. However, this cached memory is not useful to the MPI runtime as they are not read back within a short period of time, meaning these cached values are destined to be evicted without ever being used. Rank 0, which in our benchmark sends first and receives second, experiences more cache misses than Rank 1. This trend can be explained by the fact that receives will be more affected by pollution of the cache since more read operations happen within this phase. The send phase must be polluting the cache causing degraded performance by the following receive. We will do further analysis to verify our hypothesis.

*4.2.4 Final Point-to-Point Comparison.* In this section we detail the performance of our proposed designs against application level staging. We alter our benchmark to make all application-level staging optional. We turn application-level staging off when using our FPGA-aware MPI and turn it on when comparing to a non-FPGA aware MPI. Figure 10 and 11 demonstrate how our designs compare

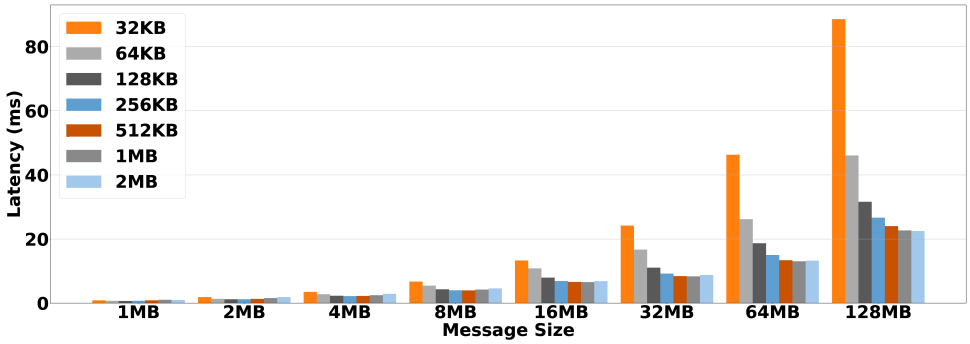



**Figure 8: Latency of pipelined design using different chunk sizes. Larger chunks generally exhibit better or equal latency to smaller chunks.**

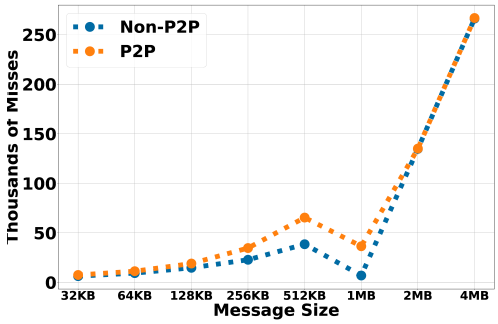
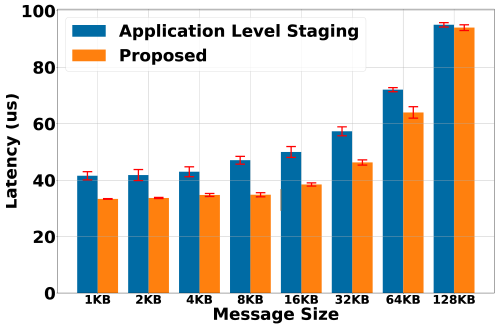
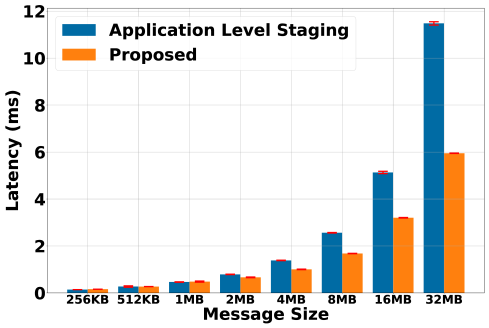



**Figure 9: Number of L1 cache misses when creating buffers with and without the P2P flag. This flag greatly affects cache performance at message sizes between 128KB and 1MB.**




**(a) For message sizes less than 64KB our design outperforms application level staging due to avoiding two extra copies.**



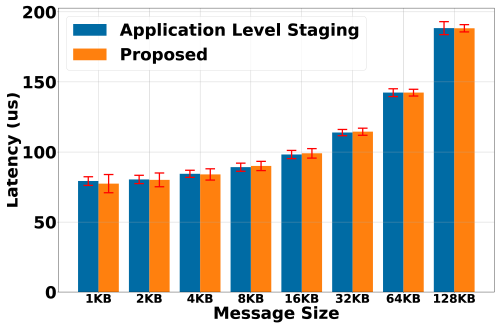

**(b) For message sizes greater than 2MB our optimized P2P design is used to avoid copies into host memory.**

**Figure 10: Intranode Point-to-point communication latency**

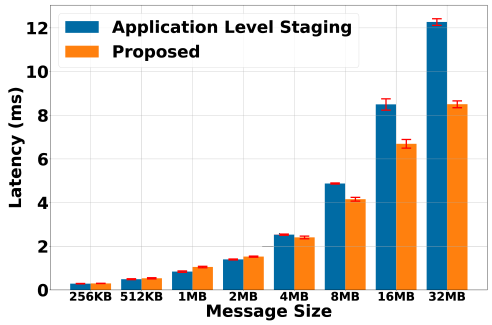
to the state of the art (i.e. application-level device transfers). The bars represent the mean latency while the error bars represent one standard deviation above and below the mean. The error bars indicate that there is very little deviation from the mean in almost all scenarios. In Figure 11a the deviation 1KB to 4KB is slightly larger than the baseline but relative to the value of the mean, these deviations are of little concern. At smaller message sizes for intranode transfers (Figure 10a), our proposed design actually shows improvements over application-level staging. This is because our design skips two copies to an application-level host buffer and instead copies straight to/from shared memory. This effectively means that there are only two copies in our shared memory Eager design vs four copies in the application-level staging, providing up to a 25% decrease in latency at the smallest messages sizes.

At larger message sizes (Figure 10b), our proposed design performs similarly to the application-level staging. However, when transferring messages 2MB and above, we are able to utilize our P2P optimization. At these larger message sizes for intranode transfers our P2P design performs significantly better than the state-of-the-art with almost as much as 33% less latency. It is important to note that these numbers are taken using a PCIe configuration that is less than ideal. Ideally, our FPGAs would be connected to the same PCIe switch, however due to resource constraints, we were unable to test with this configuration. In an ideal scenario, it is likely that our P2P design would perform even better than what we demonstrated here, especially at lower message sizes.

Looking at smaller message sizes for internode transfers (Figure 11a), our proposed design performs almost identically to the application-level staging, sometimes showing some marginal improvement. This is justified by the fact that the data movement is identical between these two cases; first, a copy happens from the device to the host, followed by a transfer over InfiniBand, and a final copy from the host to the device. However at larger message




**(a) The proposed design performs similarly to application level staging for smaller messages.**




**(b) At 2MB and above our optimized pipeline design is used to overlap staging and network transfers to reduce overall latency**

.

**Figure 11: Internode Point-to-point communication latency**

sizes, our pipelined design can be used as the transfer become less sensitive to PCIe transfer latency and becomes more bandwidth bound. Our optimized design shows up to a 33% reduction in latency at the largest message sizes.

## 5 RELATED WORK

In the HPC community, power constraint is becoming an unavoidable and challenging problem for supercomputers. More and more research has been put into FPGAs due to their energy efficiency. Even though FPGAs have never truly appeared in HPC production systems due to programmablity and design constraints, FPGA communities have gradually demonstrated the benefits of it over CPU and GPU at the application level in terms of performance and power consumption for certain application scenarios. Many such publications [2, 13, 20] discuss the current challenges of FPGA utilization and its status in HPC systems and demonstrate requirements for system architectures and interconnects to scale out FPGA resources in a distributed computing environment.

Recently, a lot of publications show the potential of FPGAs in many directions including in-switch processing, deep learning, and traditional HPC. Specifically, FPGA devices can be reconfigured to be switches with complete capabilities for handling communication, demonstrating speedups of commonly used collectives with benchmarks and mini-applications [7, 28]. Shawahna et al. [26] review the recent existing techniques for accelerating deep learning networks on FPGAs and show the potential enhancement of FPGAs for CNN acceleration. For HPC applications, Nguyen et al. [17] demonstrate the benefits of FPGA for running numerical kernels found in scientific applications.

Our paper is the first publication of an MPI design to providing support for FPGA communication without utilizing any soft logic on the FPGA nor requiring any special networking hardware. Even though communication is done explicitly through MPI primitives in contrast to several works, the syntax and semantics for programming with FPGA remain the same as with CPU, so there is no need for application-level code changes for communication tasks. However, there exists plenty of literature regarding support for FPGA communication. Notable publications are as follows: Christgau et al. [3] propose the usage of partitioned communication in FPGAs enabled by SYCL. The authors reason the full MPI implementation is too complicated for FPGAs; as a result, they limit the scope to sub-communication in which only FPGAs are involved. However, our FPGA library does not have this constraint. Haro et al. [8] propose an MPI-like extension to OmpSs, a task-based programming model developed at the Barcelona Supercomputing Center, that is capable of executing and scaling out applications on multiple FPGAs. In this work, communication is done implicitly through OpenMP-like pragma tags. Another similar work is presented by Ringlein et al. [21] in which they propose transpilation to seamlessly run MPI applications on CPU+FPGA clusters in one click without code modification. This implementation is limited by the need to specify system information to the transpiler; this means that code needs to be rebuilt for HPC jobs involving different numbers of nodes/FPGAs. Furthermore, it utilizes the soft logic on the FPGA. He et al. [9] proposed ACCL, an FPGA-specific MPI-like collective communication library over TCP/IP stack. For large messages, collectives with FPGA-driven communication, empowered by ACCL, outperform the host-based ones, empowered by OpenMPI. However, the solutions were not flexible as it was specifically designed for special systems with direct connections between FPGAs. Saldaña et al. present the work on TMD-MPI [22, 23], which is an implementation of MPI for FPGA by creating a software library for embedded processors and TMD Message Passing Engine (TMD-MPE) for hardware kernels. This engine brings MPI functionality to hardware kernels by handling MPI's protocol and packet generation. This approach not only utilizes soft logic on the FPGA but requires refactoring when migrating applications to new FPGA devices.

## 6 CONCLUSION AND FUTURE WORK

In this paper, we proposed an MPI implementation capable of doing inter-FPGA transfers. By removing application-level staging calls and doing the staging implicitly from within the MPI runtime not only does this simplify the application's code, but it also allows the MPI runtime to provide optimizations that may be difficult to implement. This enables application developers to spend less time focusing on communication code and focus more on optimizing computation, increasing productivity. Furthermore, since MPI is the defacto solution for interprocess communication in HPC workloads,

this makes it easier to alter existing CPU-based or GPU-based HPC applications. Communication code can remain untouched.

We presented our One-Shot point-to-point designs for an FPGA-aware MPI implementation and proposed two optimized designs for large message sizes. Furthermore, we executed an analysis of two optimizations provided through the OpenCL and XRT runtimes and their effects on the MPI runtime. Using these results, we implemented our FPGA-aware MPI runtime. We also developed a microbenchmark and demonstrated that our designs enable up to 25% improvement for small intranode transfers and 50% and 33% improvement for large internode and intranode transfers respectively for point-to-point communication between FPGAs.

In future work, we would like to explore accelerating existing HPC applications by developing FPGA kernels and utilizing our designs. Furthermore, we would like to provide a framework to use our MPI design in a way that enables streamed computation rather than the traditional compute-then-communicate paradigm traditionally used in MPI applications. Doing so will encourage the adoption of FPGAs in the HPC community. Adopting these devices may enable end-to-end co-design and innovative architectures to accelerate modern-day applications.

## ACKNOWLEDGMENTS

This research is supported in part by NSF grants #1818253, #1854828, #1931537, #2007991, and #2018627. We also would like to thank AMD for access to resources through the Heterogeneous Accelerated Compute Clusters (HACC) program and University of Paderborn for granting access to Noctua2.

## REFERENCES

[1] Bayatpour, M., Sarkauskas, N., Subramoni, H., Maqbool Hashmi, J., Panda, D.K.: Bluesmpi: Efficient mpi non-blocking alltoall offloading designs on modern bluefield smart nics. In: Chamberlain, B.L., Varbanescu, A.L., Ltaief, H., Luszczek, P. (eds.) High Performance Computing. pp. 18–37. Springer International Publishing, Cham (2021)

[2] Chen, D.: Fpgas in supercomputers: Opportunity or folly? In: Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays. p. 201. FPGA '19, Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3289602.3293929, https://doi.org/10.1145/3289602.3293929

[3] Christgau, S., Knaust, M., Steinke, T.: A first step towards support for mpi partitioned communication on sycl-programmed fpgas. In: IEEE/ACM International Workshop on Heterogeneous High-performance Reconfigurable Computing, H2RC@ SC 2022, Dallas, TX, USA, November, 2022 (2022)

[4] Favaro, F., Dufrechou, E., Oliver, J.P., Ezzatti, P.: Time-power-energy balance of blas kernels in modern fpgas. In: Navaux, P., Barrios H., C.J., Osthoff, C., Guerrero, G. (eds.) High Performance Computing. pp. 78–89. Springer International Publishing, Cham (2022)

[5] Freitag, T.: Acceleration of an autoencoder using a fpga-soc in a high-performance node of a distributed onboard computer (2022), https://publica.fraunhofer.de/handle/publica/430107

[6] Graph 500 green list (november 2022). https://graph500.org/?page_id$=$1128 (2022)

[7] Haghi, P., Guo, A., Xiong, Q., Yang, C., Geng, T., Broaddus, J.T., Marshall, R., Schafer, D., Skjellum, A., Herbordt, M.C.: Reconfigurable switches for high performance and flexible mpi collectives. Concurrency and Computation: Practice and Experience **34**(6), e6769 (2022)

[8] de Haro, J.M., Cano, R., Alvarez, C., Jiménez-González, D., Martorell, X., Ayguadé, E., Labarta, J., Abel, F., Ringlein, B., Weiss, B.: Ompss@ cloudfpga: An fpga task-based programming model with message passing. In: 2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS). pp. 828–838. IEEE (2022)

[9] He, Z., Parravicini, D., Petrica, L., O'Brien, K., Alonso, G., Blott, M.: Accl: Fpga-accelerated collectives over 100 gbps tcp-ip. In: 2021 IEEE/ACM International Workshop on Heterogeneous High-performance Reconfigurable Computing (H2RC). pp. 33–43. IEEE (2021)

[10] Jouppi, N.P., Young, C., Patil, N., Patterson, D., Agrawal, G., Bajwa, R., Bates, S., Bhatia, S., Boden, N., Borchers, A., et al.: In-datacenter performance analysis of a tensor processing unit. In: Proceedings of the 44th annual international symposium on computer architecture. pp. 1–12 (2017)

[11] Laboratory, N.B.C.: Mvapich: Mpi over infiniband, 10gige/iwarp and roce. http://mvapich.cse.ohio-state.edu/

[12] Laboratory, N.B.C.: OSU Micro-Benchmarks. http://mvapich.cse.ohio-state.edu/benchmarks/, [Online; accessed May 15, 2023]

[13] Lant, J., Navaridas, J., Luján, M., Goodacre, J.: Toward fpga-based hpc: Advancing interconnect technologies. IEEE Micro **40**(1), 25–34 (2020). https://doi.org/10.1109/MM.2019.2950655

[14] Lin, Y.C., Zhang, B., Prasanna, V.: Accelerating gnn training on cpu+multi-fpga heterogeneous platform. In: Navaux, P., Barrios H., C.J., Osthoff, C., Guerrero, G. (eds.) High Performance Computing. pp. 16–30. Springer International Publishing, Cham (2022)

[15] Message Passing Interface Forum: MPI: A Message-Passing Interface Standard Version 4.0 (Jun 2021), https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf

[16] Meyer, M., Kenter, T., Plessl, C.: Evaluating fpga accelerator performance with a parameterized opencl adaptation of selected benchmarks of the hpcchallenge benchmark suite. In: 2020 IEEE/ACM International Workshop on Heterogeneous High-performance Reconfigurable Computing (H2RC). pp. 10–18. IEEE (2020)

[17] Nguyen, T., Williams, S., Siracusa, M., MacLean, C., Doerfler, D., Wright, N.J.: The performance and energy efficiency potential of fpgas in scientific computing. In: 2020 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS). pp. 8–19. IEEE (2020)

[18] OpenCL Specification. https://registry.khronos.org/OpenCL/specs/3.0-unified/pdf/OpenCL_API.pdf (2022)

[19] Open MPI: Open Source High Performance Computing. http://www.open-mpi.org

[20] Plessl, C.: Bringing fpgas to hpc production systems and codes (11 2018), invited talk at the R2HC'18 workshop at SC'18

[21] Ringlein, B., Abel, F., Ditter, A., Weiss, B., Hagleitner, C., Fey, D.: Programming reconfigurable heterogeneous computing clusters using mpi with transpilation. In: 2020 IEEE/ACM International Workshop on Heterogeneous High-performance Reconfigurable Computing (H2RC). pp. 1–9. IEEE (2020)

[22] Saldana, M., Chow, P.: Tmd-mpi: An mpi implementation for multiple processors across multiple fpgas. In: 2006 International Conference on Field Programmable Logic and Applications. pp. 1–6 (2006). https://doi.org/10.1109/FPL.2006.311233

[23] Saldaña, M., Patel, A., Madill, C., Nunes, D., Wang, D., Chow, P., Wittig, R., Styles, H., Putnam, A.: Mpi as a programming model for high-performance reconfigurable computers. ACM Transactions on Reconfigurable Technology and Systems (TRETS) **3**(4), 1–29 (2010)

[24] Sanaullah, A., Herbordt, M.C.: Fpga hpc using opencl: Case study in 3d fft. In: Proceedings of the 9th International Symposium on Highly-Efficient Accelerators and Reconfigurable Technologies. pp. 1–6 (2018)

[25] Sarkauskas, N., Bayatpour, M., Tran, T., Ramesh, B., Subramoni, H., Panda, D.K.: Large-message nonblocking mpi_iallgather and mpi ibcast offload via bluefield-2 dpu. In: 2021 IEEE 28th International Conference on High Performance Computing, Data, and Analytics (HiPC). pp. 388–393 (2021). https://doi.org/10.1109/HiPC53243.2021.00054

[26] Shawahna, A., Sait, S.M., El-Maleh, A.: Fpga-based accelerators of deep learning networks for learning and classification: A review. ieee Access **7**, 7823–7859 (2018)

[27] Steiger, R.: HPCG for FPGAs: A Data-Centric Approach. B.S. thesis, ETH Zurich (2022)

[28] Stern, J., Xiong, Q., Skjellum, A., Herbordt, M.: A novel approach to supporting communicators for in-switch processing of mpi collectives. In: Workshop on Exascale MPI (2018)

[29] Top500. https://www.top500.org/lists/top500/2022/11/ (2022)

[30] Wang, H., Potluri, S., Luo, M., Singh, A.K., Sur, S., Panda, D.K.: Mvapich2-gpu: optimized gpu to gpu communication for infiniband clusters. Computer Science-Research and Development **26**(3), 257–266 (2011)