A Modular Heterogeneous Stack for Deploying FPGAs and CPUs in the Data Center

Nariman Eskandari
University of Toronto
Toronto, Ontario
nariman.eskandari@mail.utoronto.ca

Naif Tarafdar
University of Toronto
Toronto, Ontario
naif.tarafdar@mail.utoronto.ca

Daniel Ly-Ma
University of Toronto
Toronto, Ontario
d.lyma@mail.utoronto.ca

Paul Chow
University of Toronto
Toronto, Ontario
pc@eecg.toronto.edu

ABSTRACT

In this work we present a heterogeneous deployment stack, called *Galapagos*, that includes the abstraction of individual nodes (FPGAs and CPUs), the communication protocols between nodes and the orchestration and connection of these nodes into clusters. The stack we create is also highly modular, allowing users to explore a design space in the implementation of their cluster such as different network protocols or communication layers. The communication layer we have currently implemented within our hardware stack, called *HUMboldt*, handles heterogeneous communication between multiple FPGAs and CPUs. We implement *HUMboldt* using High-Level Synthesis (HLS) to ensure functional portability of communicating kernels, allowing us to prototype hardware kernels in software. Our results have shown that our modular approach to this heterogeneous deployment stack has introduced very little area and latency overhead in the FPGAs and can still perform at line-rate, bottlenecked solely by the network links connecting the nodes. Our results also highlight the scalability of our design as our performance remains limited by the network links when the cluster size increases.

KEYWORDS

Abstraction layers, reconfigurable computing, deployment stack, heterogeneous computing, FPGAs, communication Layer, orchestration, high-performance computing, cloud computing

ACM Reference Format:
Nariman Eskandari, Naif Tarafdar, Daniel Ly-Ma, and Paul Chow. 2019. A Modular Heterogeneous Stack for Deploying FPGAs and CPUs in the Data Center. In *The 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '19), February 24–26, 2019, Seaside, CA, USA.* ACM, New York, NY, USA, Article 4, 10 pages. https://doi.org/10.1145/3289602.3293909

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
FPGA '19, February 24–26, 2019, Seaside, CA, USA
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6137-8/19/02...$15.00
https://doi.org/10.1145/3289602.3293909

1 INTRODUCTION

The integration of accelerators in the data center has been shown to be beneficial [1, 2] but using heterogeneity can be difficult for data center application developers and system administrators. This heterogeneity is especially difficult for application developers when integrating FPGAs as this usually requires the user to design the application and management circuitry, including the network stack and memory management. This difficulty increases substantially at scale not only for the management of the individual FPGAs but the connection and communication between them. In this work we approach the challenge of integrating FPGAs at scale through the use of a hardware stack shown in Figure 1. A hardware stack, analogous to software stacks, represents different layers of abstraction, giving users the flexibility for the amount of abstraction they require and allows them to have different implementations of individual layers of the stack. The flexibility provided by a modular implementation of a stack allows researchers to explore a design space with respect to different implementations of heterogeneous clusters easily, as the layers can be changed independent of each other as long as the interfaces between the layers remain the same. Some example explorations could be the research of the integration of future internet network protocols such as Content Centric Networks or IPV6, or using different transport layers such as TCP or UDP, or building application-specific layers.

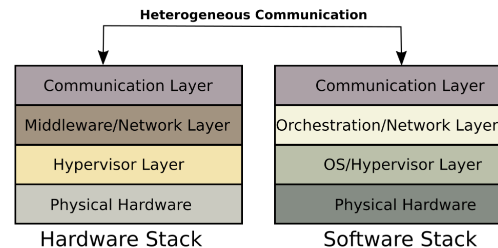

Figure 1: Our definition of a hardware deployment stack with traditional software deployment stack [3].

For both application developers and system administrators, the lack of a common communication standard among clusters of accelerators and CPUs raises challenges in creating communication links between different devices, and supporting the network connections between these devices. This communication layer is a bridge between the hardware deployment stack we made in this paper with

traditional software deployment stacks. Another desirable attribute for application developers is functional portability of a distributed application across different devices, which usually requires a developer to tailor an application specifically for each device and manage its communication. Both heterogeneity and functional portability become even more daunting at the data center scale where we can have potentially thousands of nodes interacting. We believe the challenges can be addressed with a portable communication layer across both CPUs and FPGAs, which we address in this paper.

The main contributions of this paper are as follows. First, we create our heterogeneous deployment stack by building on top of an open source FPGA orchestration tool [4]. The original tool maps streaming FPGA kernels onto FPGA abstractions and connects the multiple FPGAs together. This tool handles the abstraction of individual FPGAs and FPGA clusters. However, due to specific implementation issues, there are some scalability limitations that we first address. We also change the tool from a monolithic FPGA clustering tool to a heterogeneous deployment stack by introducing modularity at different layers such as the Communication layer and Middleware/Network layer. We demonstrate the modularity by having the same application communicate over different network protocols, such as Ethernet and TCP, without changing the application. In this paper Ethernet refers to Layer 2 Ethernet protocol within the OSI Network stack. We call our entire heterogeneous deployment stack *Galapagos*. As an example communication layer within *Galapagos* we build *HUMboldt*, which is a heterogeneous message passing communication layer, allowing messages to be sent amongst and across different FPGA kernels and CPU kernels. This is implemented as a high-level synthesisizable (HLS) and software library allowing an application developed with this library to be functionally portable across both CPUs and FPGAs. The functional portability is important because it enables application development in a pure software environment. Once correct functionality has been achieved, parts of the code can be ported to run as hardware without modifying the code.

The remainder of the paper is organized as follows. Section 2 provides background about our definition of a heterogeneous stack, different communication models and the infrastructure we build on. Section 3 explores related works in traditional software deployment stacks, other communication layers specifically on multi-FPGA clusters. Section 4 explores our modular rebuild of an FPGA cluster generator, the implementation details of our communication layer, and provides details on how to interface with our system and tool flow. Section 6 shows our results with microbenchmarks measuring the performance of our communication layer and infrastructure between FPGAs and CPUs. Lastly, we conclude our paper in Section 7 and give future work in Section 8.

2 BACKGROUND

In this section, we introduce our Hardware Stack for deploying heterogeneous applications in FPGA and CPU clusters. This stack can be seen in Figure 1. We elaborate in detail on the Communication Layer as this layer is used to bridge FPGA and CPU nodes within a cluster. We also highlight other layers that are used at the lower levels for various levels of abstraction.

Each layer of the stack represents a different view of usability for the user. Each of these layers provides a standard API to the layer above. This allows a user to have different implementations of a particular layer of the stack without a complete redesign of their system, as long as they maintain the API between these layers. The modularity of the stack allows researchers to explore a design space of different implementations of heterogeneous clusters. The bottom layer of both stacks, which is the Physical Hardware layer, represents the actual physical hardware with no abstraction. The next layer, which is the Hypervisor layer in Hardware and the OS-/Hypervisor layer in Software, represents a single node abstraction. In both hardware and software, the user is provided a set of abstractions for the I/O on the individual node. In our hardware stack there is no virtualization on the I/O as the user gets full access to the I/O, whereas software virtualization allows multiple users to share the I/O with the view that they each have the full I/O port. The layer above the Hypervisor refers to the orchestration and connection of individually abstracted nodes, which we refer to as Middleware/Networking in Hardware and Orchestration/Networking in Software. Lastly, we bridge the hardware and software stack through the use of a Communication layer. The Communication layer is highly dependent on the communication model of a given application. In this work we implement an example Communication layer but due to the modular design of our system other communication layers can be built.

2.1 Hypervisor and Middleware/Network Layer

In this section, we describe the open source framework that provides us with the Hypervisor and Middleware/Network Layer of the hardware stack. We rebuild the original Middleware/Network layer to increase its scalability and modularity. This framework is described in [4] and further extended in [5]. In these works, Tarafdar et al. introduce a multi-FPGA abstraction layer that maps a graph of streaming IP blocks connected by a large logical switch onto a multi-FPGA network-connected cluster that is provisioned from an elastic pool of cloud resources. From a high-level, the user provides a collection of kernels, a logical file describing the entire cluster and a mapping file of kernels to physical FPGAs. The high-level view of this framework is shown in Figure 2. The user is returned a network handle for their FPGA cluster.

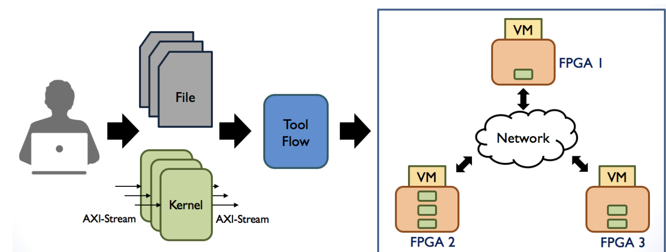

Figure 2: High-level view of original open-source framework

Each FPGA in the cluster has its physical resources abstracted with a Hypervisor. The Hypervisor exposes a control interface through PCIe and a data interface through the 10G Ethernet port,

connecting to an application region. The high-level view of the Hypervisor is shown in Figure 3. Within the application region, the framework places an interconnect on each FPGA to interface with all kernels within the cluster (either directly connects to kernels that are locally connected or encapsulates the packet with network information to make a network hop). This direct connection to all packets has an inherent limitation as the interconnect has at most 16 ports, thus limiting the total number of kernels within the entire cluster to 16. Furthermore, the modularity of this implementation is limited as the user is forced to use Ethernet packets between FPGAs, making the communication unreliable. More details about the data center infrastructure, and network architecture can be found in [4], and more details on the FPGA Hypervisor can be found in [5].

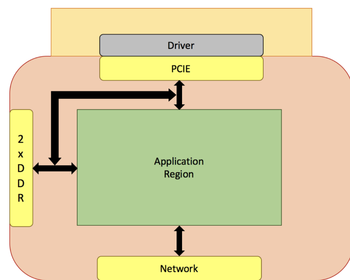

Figure 3: Hypervisor

2.2 Communication Layer

The communication layer is very application-specific as different applications exhibit different communication traffic patterns. Two popular communication models are streaming and message passing. In a streaming model, data is being sent continuously through a point-to-point channel. Some implementations of streaming communication layers include the Real-time streaming protocol (RTSP) [6] and MPEG-DASH [7]. These layers are typically built on top of network protocols like UDP, or even Ethernet as these protocols have better latency but lack reliability by default.

In a message passing model, data can be transferred between arbitrary nodes. Generally, shorter packets provide lower latency while longer packets provide better throughput because message overheads are minimized. Users must partition data into messages and include a destination address when sending, unlike a stream where it is a continuous flow of data to one or more preconfigured receivers. The most common implementation of a message passing model is the Message Passing Interface (MPI) [8]. We have decided to implement our first communication layer as a subset of MPI because there has been prior work in implementing MPI on an FPGA cluster. MPI is also a well-known standard API that is widely used in various types of HPC applications. This helps us with our goals of heterogeneity, functional portability and scalability.

Another messaging protocol that has a significant user community is ZeroMQ (0MQ) [9]. ZeroMQ uses a socket-like interface that supports multiple message patterns such as request-reply and publish-subscribe. By using a socket-like interface, it would also be easier to use it like a FIFO, or streaming interface.

3 RELATED WORK

The work we present in this paper is a heterogeneous multi-FPGA and CPU deployment stack. In this section, we explore other deployment stacks used in a homogeneous CPU environment as well as other FPGA cluster implementations and heterogeneous communication layers.

3.1 Software Deployment Stacks

Distributed and big data computing often requires the use of many compute nodes, traditionally software nodes. Figure 1 on the right shows an example of a software stack. Clusters of software nodes are grouped and connected with orchestration software such as Heat, which is part of OpenStack [10]. Orchestration software usually provisions a group of individual software nodes, and connects the software nodes between them. These nodes are often connected with popular network protocols such as IP. However on top of these interconnected software nodes, depending on the application, a user can deploy a communication layer to easily communicate between these nodes. For example, if the application describes streaming kernels then the user would want to use a communication layer amenable to streaming such as ZeroMQ [9].

3.2 FPGA Cluster Implementations for the Data Center

The flagship implementation of FPGAs in the data center is Microsoft's first version of Catapult, which includes FPGA clusters connected to a CPU as an offload engine [1]. This has limited flexibility as the FPGAs are connected to the network through the CPU. Microsoft addressed this lack of flexibility in their second iteration of Catapult [2], in which all FPGAs are connected to the network directly. To keep the network switch requirements constant, CPU network connections are made through the FPGA, as opposed to having both CPUs and FPGAs connected to the network. The work presented in this paper has a physical infrastructure similar to that of the second iteration of Catapult with network-connected FPGAs, however we also connect the CPUs directly to the network, avoiding the complexity of bypassing CPU network packets through the FPGA. We approach the challenge of heterogeneity by creating a uniform communication layer between CPUs and FPGAs to virtualize a heterogeneous network connection to look the same to both hardware and software functions. Our work also builds on top of [4], which is an orchestration layer on top of a heterogeneous network fabric, abstracting away the difficulties of configuring a multi-FPGA cluster. This is a higher level of abstraction than used in Catapult based on what is presented in the available publications.

There has been some work at a smaller scale to provide multi-FPGA orchestration. One example is the Maxeler MPC-X project [11]. A ring of eight FPGAs is connected to a network that can receive job requests as data-flow graphs. This data-flow graph is then mapped onto a set of the FPGAs available in the ring, abstracting the mapping and representation of a multi-FPGA application onto a physical multi-FPGA topology. Our work in this paper looks at a similar level of abstraction in a flexible network topology such as the one in Catapult.

3.3 Heterogeneous Communication Layers

In this section, we explore other implementations of heterogeneous communication layers. We look at two in particular: the first being TMD-MPI [12] and the second being Novo-G# [13].

3.3.1 TMD-MPI. The work presented in [12] explores an implementation of MPI within a multi-FPGA environment called TMD-MPI. TMD-MPI implements a subset of the MPI protocol to allow hardware or software processing engines on the FPGA to communicate amongst each other on the same FPGA and across multiple FPGAs. The hardware versions of the MPI functions are implemented in VHDL. In this paper, a subset of these functions are described in HLS-synthesizable C code.

TMD-MPI was created to be portable across multiple physical platforms so it was implemented in several layers, including layers that correspond to the physical setup of the network connected FPGAs. In this paper, we avoid the need to create equivalent layers as our communication layer modularly builds on an improved cluster generator tool that handles the communication of blocks within an FPGA and across multiple FPGAs in a data center.

3.3.2 Novo-G#. The work presented in [13] is a heterogeneous environment with 24 CPU servers that are connected via PCIe to FPGA boards with 4 Stratix V FPGAs. There are direct connects between the FPGAs on an individual board forming a 3D torus with a custom hardware network stack to support these direct connections. If needed, communication between host nodes can use MPI. The Novo-G# is a system that shows both the use of a custom hardware network stack for FPGA-to-FPGA communication as well as a model where accelerators are connected to host nodes and the host nodes can communicate using a standard software MPI library. In this paper, hardware and software components of the same application can communicate with the same communication layer as peers, which makes it much easier to use hardware or software interchangeably for any computing kernel.

4 IMPLEMENTATION

We rebuilt the open source framework described in [4] to improve modularity, reliability, and scalability. This improved framework provides the *Galapagos* stack with the Hypervisor and Middleware/Network Layer. These improvements allow the user to implement designs with different network protocols (e.g Ethernet or TCP) and communication layers by changing a configuration file describing the heterogeneous cluster. The user can target any number of available devices (FPGA and/or CPU) with a limit of 16 kernels per FPGA due to the number of ports on the Xilinx switch IP core. Once we addressed a few limitations of the original open-source framework, we built *HUMboldt* as a Communication Layer. Due to the modularity of the system, *HUMboldt* can fit on top of both TCP and Ethernet without changing the user application and its interfaces. *HUMboldt* maintains functional portability between hardware and software nodes. The heterogeneous connection between FPGAs and CPUs is easy to scale up as the user only needs to change two configuration files.

4.1 Galapagos Middleware/Networking Layer

The application region that can be built by the original open-source framework Middleware/Network Layer and the *Galapagos* Middleware/Network Layer can be seen in Figures 4 and 5. Both iterations of this framework take a description of a cluster composed of streaming kernels with a unique ID and maps it to multiple FPGAs. Each streaming kernel uses the AXI-stream protocol [14] with a dest field to specify which kernel the packet is destined for. A logical view of this infrastructure is a large switch connecting all the kernels within the cluster. The framework transforms this logical switch into two physical switches, with the first being an AXI-stream switch on the FPGA and the second being a top-of-rack network switch.

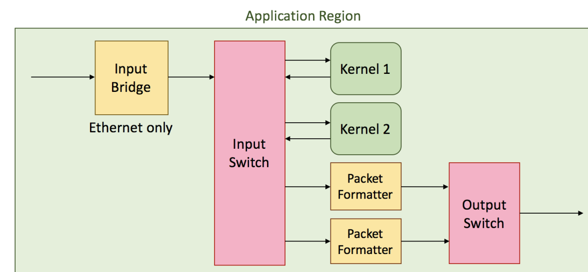

Figure 4: The original open-source framework from [4].

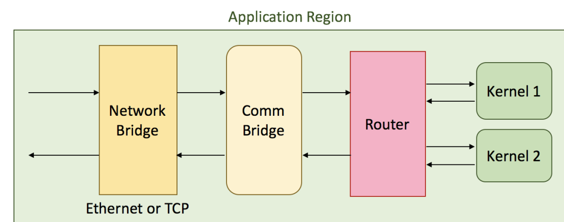

Figure 5: A high-level overview of *Galapagos.*

In the framework, described in [4] Ethernet packets are transformed into AXI-stream packets through the use of an Input Bridge. This then connects to an AXI-stream switch on the FPGA. This switch is connected to all kernels within the cluster, either directly if on the same FPGA, or through a Packet-Formatter module that encapsulates the AXI-stream packet with the appropriate Ethernet headers, and places the AXI-stream dest field in the Ethernet payload (one packet formatter for each kernel outside the FPGA). These direct connections limit scalability as the number of kernels in the cluster is limited by the 16 ports of the AXI-stream switch.

In the Middleware/Network Layer of *Galapagos*, we first addressed scalability by creating an AXI-stream router. The AXI-stream protocol is independent of higher layer protocols. The block diagram of the router is shown in Figure 6. The router on each FPGA includes a routing table indexed by the unique ID of each kernel in the entire cluster (including kernels not on this FPGA) and the network address (Ethernet or IP) of the FPGA that has each kernel. All kernels output their packets to the router that reads the AXI-stream dest field of the packet and then looks up the network

destination in the routing table. Then, the packet is either routed back to the AXI-stream switch or routed out to the network. The number of ports on this router is equal to the number of kernels only on this FPGA, as all packets leaving the FPGA share one channel. This limits us to have up to 16 kernels on a particular FPGA as opposed to the entire cluster as per the original design. The routing table is automatically generated by our modifications to the cluster generator.

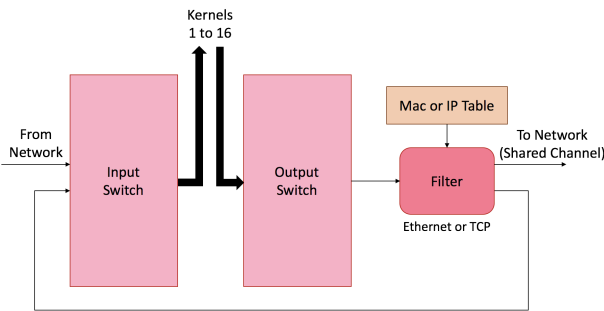

Figure 6: Router

The network bridge that is shown in Figure 5 is responsible for converting network packets into AXI-stream format and vice versa. For the Ethernet Network Bridge of Figure 5, this paper combines the Input Bridge and a modified version of the Packet Formatter of Figure 4 into a single module. The block diagram of this module is shown in Figure 7. The modified packet formatter is equipped with a look up table that has the MAC addresses for each destination kernel. Using the MAC address lookup table in the packet formatter decreases resource utilization as it only uses one packet formatter logic with a small memory instead of multiple packet formatters for each kernel outside the FPGA, and it helps the *Galapagos* to be more scalable.

In [4] the multi-FPGA communication uses the Ethernet protocol, which is not reliable. In the Middleware/Network Layer of *Galapagos*, to address reliability, an optional TCP core [15] is integrated into the framework. In this paper an additional Network Bridge is created for TCP, allowing us to standardize the interface between the Hypervisor and the Application Region. The standardization of the interface allows us to use both TCP and Ethernet interchangeably, thus addressing modularity. The block diagram of the TCP Network Bridge is illustrated in Figure 8. Observe that the interfaces are the same as for the Ethernet Network Bridge in Figure 7.

A user may wish to create a communication layer (e.g. MPI) on top of standard network layers. A communication bridge is used to transform network packets to communication layer compliant packets. In the Middleware/Network Layer of *Galapagos*, we have implemented a Comm Bridge that translates the AXI-stream packets of the Network Bridge to conform to the underlying MPI packet protocol used by HUMboldt. If we wanted to support ZeroMQ we would implement an appropriate Comm Bridge to support that protocol. Separating the functions of the Network Bridge and the Comm Bridge makes it possible to change the network protocol

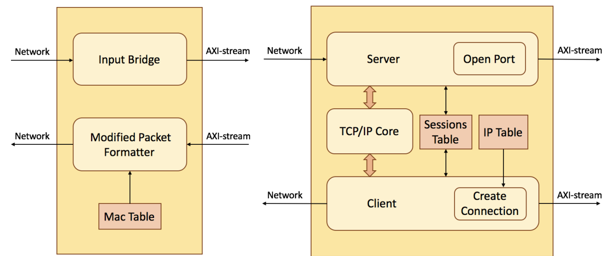

Figure 7: Ethernet Network Bridge **Figure 8: TCP Network Bridge.**

independent of the communication layer, and vice versa. We further address modularity by allowing the user to configure the network protocol (Ethernet or TCP) and communication bridge via the mapping configuration file. Details of the mapping file can be found in Section 4.3.

4.2 HUMboldt Communication Layer

Due to the layered design of *Galapagos*, the implementation of the communication layer can be any communication model as long as it adheres to the AXI-stream interface, and the appropriate communication bridge to convert AXI-stream packets into communication layer specific packets is provided. The communication layer we present here implements *HUMBoldt*, a minimal subset of MPI that is sufficient to enable basic message passing between kernels. MPI is a standard API that defines signatures for functions such as sending and receiving messages. These signatures must remain the same for all implementations of MPI. Even though the implementation for these functions vary according to platform (e.g FPGA or CPU), the standardization of the protocol for *HUMBoldt* allows for communication between heterogeneous platforms.

4.2.1 MPI Communication Layer. Here, we provide a brief explanation of MPI and the subset of the MPI library that we support, which we call Heterogeneous Uniform Messaging (*HUMboldt*). Using MPI, many parallel processes (ranks) communicating via messages can be run on multi-node platforms. We will refer to ranks as kernels to be consistent with our previous use of the term *kernels*. In MPI software implementations, such as MPICH [16] and OpenMPI [17], functions are provided to transmit data among different kernels in various ways. The minimum subset of MPI functions needed for communication and currently implemented in HUMboldt:

(1) *MPI_Init*: This function initializes the MPI environment, and does the basic setup such as network interface initialization.
(2) *MPI_Send and MPI_Recv*: These two functions are building blocks of the MPI programming model that enable data transmission among kernels. For every Send to a kernel, there must be a matching Receive on that kernel to get the data from the sender.
(3) *MPI_Finalize*: Makes sure that all kernels are done with their processes.

Other MPI functions can be added to the HUMboldt library by writing additional HLS-synthesizable functions. These functions will fit into the same HUMboldt flow.

There are two types of networks in our *HUMBoldt* communication layer. The Intra-FPGA AXI-stream network is used for kernels that are located in the same physical FPGA, and the Inter-node communication between FPGAs and CPUs uses the network. The network communication currently supports TCP or Ethernet but any network protocol with an AXI-stream interface can be used. We use the same underlying *HUMBoldt* protocol for kernels communicating within the same node (FPGA or CPU) and between nodes, as our communication bridge encapsulates a *HUMBoldt* compliant packet with the appropriate network header (e.g TCP or Ethernet) to handle the inter-node communication.

Figure 9 shows the Envelope that carries the *HUMBoldt* messages between kernels. The first two bytes correspond to the destination and source kernels. The Packet Types are *send request*, *clear to send*, *data*, or *done*, where the different packet types are used to implement the message passing protocol underlying *HUMBoldt*. The next three bytes specify the size of the message being sent. In the standard implementation of MPI, the Tag is an option for the user to tag optional metadata to transactions, and for compatibility purposes we keep this as a field in the Envelope used for *HUMBoldt*. The Data Type field helps to process the different data types properly.

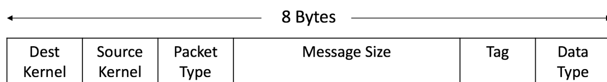

Figure 9: Envelope Packet

4.3 System Interface

This subsection specifies how the user would interface with the *Galapagos* hardware stack. A user provides application files as well as the cluster description files. The tool flow of the heterogeneous stack takes these files, and makes a cluster that all the hardware and software kernels can communicate.

In Listing 1, we illustrate an example of a common model for HPC applications in which one kernel is responsible for distributing data to several other kernels, and gathers processed data, when all kernels are done.

```
1  #include "HUMboldt.h"
2  #define MAX_ITR 10
3  #define DATA_SIZE 1000
4  #define TAG 0
5
6  int main(int argc, char* argv[])
7  {
8      HUM_Init(&argc,&argv);
9      int data_array[DATA_SIZE];
10     int size = atoi(argv[1]);
11     int rank = atoi(argv[2]);
12
13     for(int i = 0 ; i < MAX_ITR ; i++){
14         if(rank == 0){
15             for(int r = 1; r < size ; r++)
16                 HUM_Send(data_array,DATA_SIZE,
                       MPI_FLOAT,r,TAG,MPI_COMM_WORLD);
17         }
18         else{
19             HUM_Recv(data_array,DATA_SIZE,MPI_FLOAT
                   ,0,TAG,MPI_COMM_WORLD);
20         }
21         /* process data*/
22         if(rank == 0){
23             for(int r = 1 ; r < size ; r++)
24                 HUM_Recv(data_array,DATA_SIZE,
                       MPI_FLOAT,r,TAG,MPI_COMM_WORLD);
25         }
26         else
27             HUM_Send(data_array,DATA_SIZE,MPI_FLOAT
                   ,0,TAG,MPI_COMM_WORLD);
28     }
29     HUM_Finalize();
30     return 0;
31 }
```

Listing 1: HUMboldt sample code

The code in Listing 1 needs a few minor modifications to be synthesizable by Vivado HLS. For example in line 6, instead of argc and argv to input the size and kernel number (rank) we use two constant ports. These values are assigned automatically in the tool flow using the logical description file of the kernels, so there is no need for lines 10 and 11. Furthermore, some HLS pragmas should be added just for interfaces that are always the same. These changes are shown in Listing 2. These modifications can be done using a very simple script, but essentially, the same code can be run as software or implemented as hardware in an FPGA. This code demonstrates that *HUMboldt* is heterogeneous and functionally portable for different processing nodes in a cluster.

```
1 int main(const int size, const int rank)
2 {
3 #pragma HLS INTERFACE ap_ctrl_none port=return
4 #pragma HLS resource core=AXI4Stream variable=
      stream_out
5 #pragma HLS resource core=AXI4Stream variable=
      stream_in
6 #pragma HLS DATA_PACK variable  = stream_out
7 #pragma HLS DATA_PACK variable  = stream_in
```

Listing 2: HUMboldt synthesizable sample code

In addition to the code in Listing 1, cluster description files are required for *Galapagos*. Listing 3 is a sample logical file, and Listing 4 is a sample mapping file. Line 5 in Listing 3 shows an example of the *rep* field to determine the number of replications of each kernel within the entire cluster. This shows that how easy it is to have multiple replications a kernel. The *num* is an unique identifier of each kernel that can be used in mapping file.

The logical file in Listing 3 is mostly the same as the original open-source framework logical file in [4]. However, there are some small changes that are as follows. There are different naming conventions for the reset and the clock ports in the Vivado environment. In the modified logical file the user can specify the clock and reset port names (Line 6 and 7). One feature that makes this complex heterogeneous system easy to use is that a user is able to debug it by monitoring some signals. A debug capability has been added to the system, by which the signals that are marked as debug (Lines 12 and 17 of Listing 3) will be connected to a Xilinx ILA core. The Xilinx Integrated Logic Analyzer (ILA) IP core[18] is a logic analyzer that can be used to monitor the internal signals of a design, running on an FPGA. Another capability that is added to the system interface of *Galapagos* is defining a constant port that can be assigned automatically by the tool flow (Lines 19 to 23).

Listing 4 shows how the kernels, which are defined in the logical file, can be mapped into one or more FPGAs. In lines 20 to 24 it can be seen that the kernels 1-16 are mapped to a single FPGA. This

shows how easy it is to scale the system by changing some lines in the configuration files. The mapping file of the original open-source framework [4] has the same capability that *Galapagos* inherited. However, as mentioned in Section 4.1, the original framework has the limitation of 16 kernels within the entire cluster, but, *Galapagos* has no limitation for the total number of kernels. It has only the limitation of 16 kernels per FPGA.

Some additional features are added to the mapping file to support heterogeneity and modularity. For example, in Listing 4 lines 4 and 18 show two different types of nodes (software and hardware), which addresses heterogeneity and how easy it is to change a kernel from hardware to software, or vice versa. Furthermore to address modularity, the user can specify a bridge for their communication layer as shown in lines 10 to 16. If the user does not specify a bridge, then it is assumed the kernels will communicate directly via AXI-stream. Modularity within the network layer can be observed in line 19, where the user can specify the network protocol (e.g TCP or Ethernet), and network addresses as demonstrated in lines 6-7 and 25-26. The network addresses would be supplied by the manager of the data center.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <cluster>
3      <kernel> hardware_core_name
4          <num> 1 </num>
5          <rep> 96 </rep>
6          <clk> aclk </clk>
7          <aresetn> aresetn </aresetn>
8          <id_port> kernel_id </id_port>
9          <interface>
10             <direction> in </direction>
11             <name> stream_in_V </name>
12             <debug/>
13         </interface>
14         <interface>
15             <direction> out </direction>
16             <name> stream_out_V </name>
17             <debug/>
18         </interface>
19         <const>
20             <name> size </name>
21             <val> 4 </val>
22             <width> 16 </width>
23         </const>
24     </kernel>
25     <kernel> cpu
26         <num> 0 </num>
27         <rep> 1 </rep>
28     </kernel>
29 </cluster>
```

Listing 3: Sample Logical File

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <cluster>
3      <node>
4          <type> sw </type>
5          <kernel> 0 </kernel>
6          <mac_addr>  ac:c4:7a:88:c0:47 </mac_addr>
7          <ip_addr> 10.1.2.152 </ip_addr>
8      </node>
9      <node>
10         <appBridge>
11             <name> communication_bridge_eth_mpi </
                   name>
12             <to_app> to_app_V </to_app>
13             <from_app> from_app_V </from_app>
14             <to_net> to_net_V </to_net>
15             <from_net> from_net_V </from_net>
16         </appBridge>
17         <board> adm-8k5-debug </board>
18         <type> hw </type>
19         <comm> eth </comm>
20         <kernel> 1 </kernel>
21         .
22         .
23         .
24         <kernel> 16 </kernel>
25         <mac_addr>  fa:16:3e:55:ca:02 </mac_addr>
26         <ip_addr> 10.1.2.101 </ip_addr>
27     </node>
28 </cluster>
29 ~
```

Listing 4: Sample Map File

5 TOOL FLOW

To make the *Galapagos* stack work transparently and conveniently across a heterogeneous platform, a tool flow is required that takes the cluster description files (Listing 3 and Listing 4) and *HUMboldt* code (Listing 1), and creates the whole cluster automatically. Recall that one of the goals is to use identical code whether it is to run as a software kernel or as a hardware kernel. This means that the tool flow will have two paths, one to create software executables and the other to build FPGA bitstreams in a user-defined platform. Figure 10 shows the flows for hardware and software kernels.

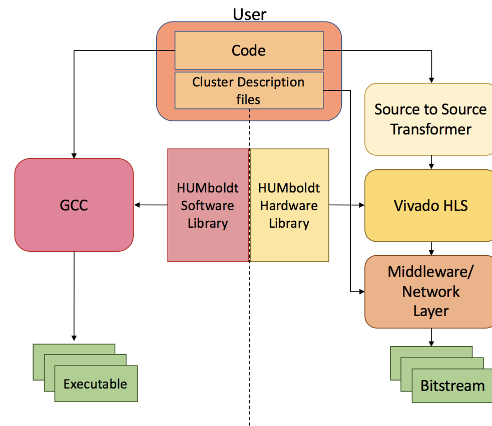

Figure 10: Software and Hardware Tool Flow

5.1 Software Kernels

Building software kernels is essentially the same as what is currently done for standard MPI software distributions. The first step is to link in the *HUMboldt* software library to the user code.

5.2 Hardware Kernels

The building of the hardware kernels requires transforming the original source code into a form that can be used with high-level synthesis (HLS). HLS essentially creates a block of hardware using software code. The hardware path of the tool flow gives these hardware blocks as well as cluster description files to the middleware/Network Layer, and it creates the bitstreams for the FPGAs in the cluster.

6 EVALUATION

In this section, the evaluation of our current platform is presented. We consider the resource utilization, latency and throughput, and scaling and heterogeneity of our platform.

The testbed that we use to run our test scenarios is a cluster of servers with Intel Xeon E5-2650 CPUs running at 2.20 GHz, each with 12 physical cores, so via hyper-threading 24 software threads can be running. The FPGAs that are located in the same network of this cluster are Xilinx Kintex UltraScale XCKU115-2-FLVA1517E devices on Alpha data ADM-PCIE-8K5 boards [19]. All network connections are 10G ethernet connected to a Dell Networking S4048-ON 10G switch. Our *HUMBoldt* implementation is used for any configurations that include a hardware node. To test the best software to software implementation, we just use MPICH, which is a mature open-source MPI implementation.

6.1 Resource Utilization of Infrastructure and Communication Layer

The resource utilization of the different layers of *Galapagos* including the Hypervisor that we got from [4] plus other parts of the Middleware/Network Layer, including off-chip memory support that we added, network bridges, communication layer bridges, and the router within the application region are shown in Table 1. The percentages, which are shown in brackets, are relative to the KU115 FPGAs. Observe that the resource utilization of *Galapagos* is about 20% when the user chooses TCP, and it is 15% when the user chooses Ethernet. The resources used here are not necessarily extra overhead as a developer would require resources to create a custom multi-FPGA interconnect as well.

Table 1: Resource Utilization of Galapagos

Galapagos Layer	LUTs	Flip-Flops	BRAMs
I) Hypervisor	95332 (14.4%)	120367 (9.1%)	255 (11.8%)
II) Network Bridge TCP	29146 (4.39%)	32582 (2.4%)	86 (4.0%)
III) Network Bridge Ethernet	582 (0.09%)	1087 (0.08%)	2 (0.09%)
IV) Communication Bridge TCP to *HUMboldt*	1039 (0.1%)	1585 (0.1%)	1 (0.046%)
V) Communication Bridge Ethernet to *HUMboldt*	729 (0.1%)	1332 (0.1%)	1 (0.046%)
VI) Router with 16 ports	5067 (0.8%)	6310 (0.5%)	1 (0.046%)
Total TCP (I + II + IV + VI)	130584 (19.7%)	160847 (12.1%)	343 (15.9 %)
Total Ethernet (I + III + V + VI)	101710 (15.3%)	129096 (9.7%)	259 (12.0%)

The other resource utilization to consider is related to the *HUMboldt* kernels. Each kernel can use any of the functions that are defined in the *HUMboldt* communication layer library. Once a *HUMboldt* function is called in the user code, the module for that function will be added to the hardware of that kernel. Multiple calls to the same function do not increase the instantiations of the hardware module. The resource utilization of each function is presented in Table 2. It can be seen that the current Send and Receive functions require minimal additional hardware resources.

Table 2: Resource Overhead of HUMBoldt Communication Layer API Functions

HUMboldt Function	LUTs	Flip-Flops	BRAMs
HUM_Send	389 (0.06%)	372 (0.03%)	0 (0%)
HUM_Recv	1180 (0.18%)	1072 (0.08%)	0 (0%)

6.2 Latency and Throughput

We have created a microbenchmark to test the send and receive functionality of our system, with one kernel sending and another kernel receiving. We change the implementation of the kernel from hardware to software and test several configurations. The configurations are as follows: software to hardware, hardware to hardware (on the same FPGA), hardware to hardware (on different FPGAs), and hardware to software. These configurations are tested with both TCP and Ethernet. Furthermore, we test the following software configurations with MPICH: software to software (on the same CPU), and software to software (on different CPUs). MPICH uses TCP for network communication. We test these using MPICH to compare the best software implementation of MPI to our *HUMboldt* communication layer. The measurements reported were averaged over many runs until results converged. For MPICH, convergence required close to a million runs due to OS and other processes effects whereas our hardware results required about 10 runs.

Figures 11 and 12 show the throughput of our benchmark. Our *HUMboldt* communication layer and the respective bridge transforming *HUMboldt* packets into network packets performs at line-rate and the effective data throughput is limited by the 10G Ethernet Core in the Hypervisor along with the respective packet headers required for TCP and Ethernet (hence the higher throughput in Ethernet than TCP). For simplicity all our kernels are connected to the same 156.25 MHz clock, which is the output of the Ethernet core running at 10G. This limits our internal FPGA throughput to 10G, however in theory this can be modified for a faster clock. Furthermore for homogeneity all kernels on the same and different nodes use the same communication protocol, however this could be further optimized by simplifying the protocol of kernels on the same node. Between hardware and software in Figure 11, we cannot scale past a 128 KB payload after which we notice packet drops using Ethernet because of the lack of reliability. This is because the software kernel cannot receive data as fast as the hardware can send it. Between two hardware kernels, we achieved close to the maximum TCP core bandwidth that is mentioned in [15]. The curves show the expected shape where the bandwidth improves as the payload size, and hardware to hardware works best when compared to links involving a software node. Note that the curve for hardware to hardware in the same FPGA is the same in both Ethernet and TCP cases because the routing is done internal to the FPGA without needing to add the Ethernet or TCP headers. The

hardware to software TCP is about half that of software to hardware. This is because the software node cannot process packets as fast as the packets received from the hardware node, thus dropping packets and initiating retransmissions.

We do not plot the latency and bandwidth between two software nodes on the same CPU because they would be difficult to show on the same graph as the others due to the scaling required and there is no additional information gained by plotting them on the same graph. We present these numbers only to illustrate the challenge of competing with MPI implemented in a shared-memory processor running at over 2 GHz. MPICH uses shared memory for the communication, much of which can fit in the cache so the bandwidth can be very high and the latency very low. The observed bandwidth for a 512 KB payload is approximately 60 GB/s with a latency of 0.21 μs. It can also be seen that MPICH shows a very high throughput for small packets and then drops off. We do not understand the inner workings of MPICH to explain this behavior, but we suspect optimizations for small packets or possibly cache effects.

The latency is shown in Table 3. We define latency as the time for sending a zero-payload size transaction, including sending an *envelope*, receiving a *clear to send*, sending a zero-payload packet, and receiving a *done*. We measure the cycle counts with a probe (Xilinx Integrated Logic Analyzer) running on the hardware with a 156.25 Mhz clock. The latency for two kernels sending and receiving an entire transaction (all four packets) on the same FPGA is deterministic and takes 29 cycles. When the receiving kernel is on another node (different FPGA or CPU) we incur an additional latency required to transform *HUMboldt* packets into network packets. Each packet would be processed through a communication bridge and a network bridge. These latencies are shown in Table 4.

Table 3: Latency of zero-payload packets

Microbenchmarks	Ethernet (μs)	TCP (μs)
Hardware to Hardware (same node)	0.2	0.2
Hardware to Hardware (different node)	5.7	15.2
Software to Hardware	27.5	48.8
Hardware to Software	34.7	81.2

Table 4: Per Packet Additional Latency

Component and Protocol	Send (cycles)	Receive (cycles)
TCP Communication Bridge	9	5
Ethernet Communication Bridge	6	5
TCP Network Bridge	177	199
Ethernet Network Bridge	7	12

On top of the internal FPGA latencies that are mentioned above there is a non-deterministic latency of the network depending on the network topology. It can be seen that whenever any communication uses a network link, the additional cycles for the bridges is very small except when TCP is used. Also, whenever there is a software node involved, it is clear that handling protocols in software is much slower than in hardware.

As a sanity check of our numbers we can make an approximate comparison to the latency number measured for Microsoft Catapult [2] where they report an FPGA to FPGA round-trip latency of 2.88μs when using their LTL communication layer over a 40G Ethernet link through a single top-of-rack switch. Table 3 shows that the FPGA to FPGA latency when using the Ethernet network link (hw to hw diff) on our platform is about 6 μs, which is the sending and receiving of four packets of 8 bytes according to the *HUMboldt* protocol. A single round-trip latency is therefore about 3 μs. On top of Ethernet, Catapult uses UDP frame encapsulation, IP routing, and adds their LTL protocol [2], which we do not have with a raw Ethernet link. Catapult is also running at 40G versus our 10G, so they are sending more bytes at a higher rate. While the round trip numbers are similar, it can be seen that they are not directly comparable. However, we argue that our implementation is within reason compared to Catapult.

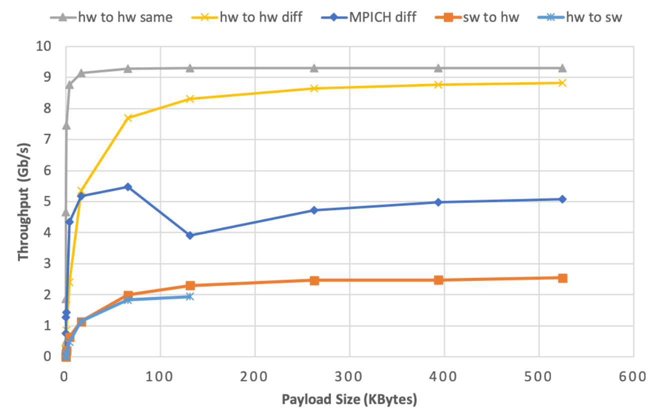

Figure 11: Ethernet Throughput.

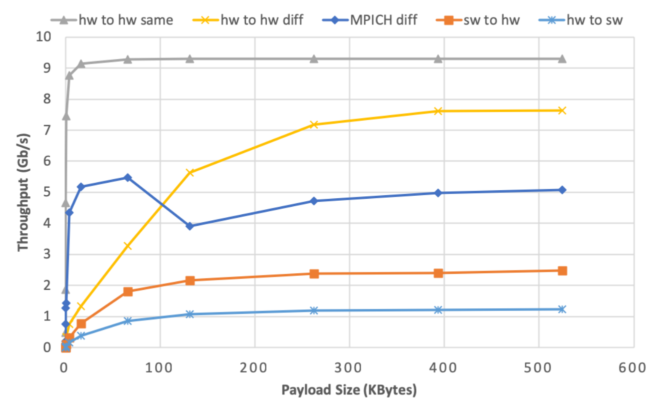

Figure 12: TCP Throughput.

6.3 Scalability, Heterogeneity and Performance

To test scalability and heterogeneity, we built a simple application proxy shown in Listing 1 that is representative of a common

computing pattern. It has a central node forking tasks to many processes and gathering the results. The goal of this application is to exercise our platform and show how easy it is to scale the application to run with different numbers of nodes. The number of nodes within our system can scale to the depth of the routing table, which is currently set to 256 but can be updated as required. Furthermore, we have a limitation of 16 kernels per FPGA. Within our experiment we easily implemented several working configurations up to 96 kernels across 6 FPGAs (limited by the number of FPGAs available in our cluster) by just making a few changes in the configuration files as described in Section 5. Our experiments have shown that scaling to 96 kernels is still limited by the network link and not by our infrastructure. We have scaled our microbenchmark experiments with 96 kernels across 6 FPGAs and have observed the same performance between two kernels. Within our experiment, we also tested heterogeneity by implementing multiple kernels in hardware and software and demonstrated functional correctness across a distributed heterogeneous application.

Our experiments have shown that with limited latency overhead (on the order of up to 200 cycles when TCP is used) and no throughput overhead we can provide scalability and heterogeneity. Furthermore, some latency overhead will be incurred by any multi-FPGA communication link as some protocol must be in place to connect the FPGAs together. The small performance overhead of our abstraction layers means that the upper bound performance of a multi-FPGA application will not be significantly affected by our abstractions. The true performance of any distributed application will be dependent mainly on the scalability of the application itself, and how well the application can map to hardware and not be impacted by our abstraction layers. Therefore, we limit our experiments to our microbenchmarking, which strictly tests the underlying communications and our ability to scale without any affects that are application dependent.

7 CONCLUSION

A heterogeneous deployment stack is necessary to give users flexible heterogeneous clusters at scale. The modularity in our stack, *Galapagos* is realized and demonstrated with the creation of the *HUMboldt* communication layer on top of multiple implementations of a networking protocol. This modularity will allow researchers to be able to experiment with different implementations of heterogeneous clusters. These layers of abstraction work at linerate, allowing users to focus on their application. We show that we can target both heterogeneity and scalability quite easily, as we can use multiple configurations that we can easily scale by changing two configuration files. We have also shown that heterogeneity can be achieved by combining a software and hardware deployment stack through a common layer, which in our case is the use of a communication layer such as *HUMboldt*. This work is open-source and can be downloaded at https://github.com/tarafdar/galapagos and https://github.com/eskandarinariman/HUMboldt.

8 FUTURE WORK

We have built *Galapagos* and *HUMboldt* to make it easier to build multi-FPGA and heterogeneous systems. To show the true power of this infrastructure, we will build some showcase applications that can leverage such a platform.

The *HUMboldt* layer currently supports message passing using a minimal subset of the MPI standard. To more fully support MPI, more functions need to be implemented, which just adds to the current library. A fully heterogeneous MPI should leverage existing software implementations, such as MPICH [16] to achieve the most efficient implementation of MPI on the software side. This could be achieved by bridging *HUMboldt* to MPICH.

To support more types of applications it would be good to add the streaming communication model to *HUMboldt*. Because of the modularity that we added to *Galapagos*, and the HLS implementation of *HUMboldt*, it will not be difficult to add streaming. Just as we have done by using MPI for message passing, it would be good to use a popular programming model such as ZeroMQ [9] to define the interfaces.

In general, since communication is often a bottleneck in multinode applications, more work needs to be done to implement efficient heterogeneous communications. The *Galapagos* platform makes it possible to do this exploration without changing the application layers so it will be easy to see the impact of different communication protocols on any applications we build.

REFERENCES

[1] Andrew Putnam et al. A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services. *ACM SIGARCH Computer Architecture News*, 42(3):13–24, 2014.
[2] Adrian Caulfield et al. Configurable Clouds. *IEEE Micro*, 37(3):52–61, 2017.
[3] Naif Tarafdar et al. Galapagos: A Full Stack Approach to FPGA Integration in the Cloud. *IEEE Micro*, 38(6):18–24, Nov 2018.
[4] Naif Tarafdar et al. Enabling Flexible Network FPGA Clusters in a Heterogeneous Cloud Data Center. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 237–246. ACM, 2017.
[5] Naif Tarafdar et al. Heterogeneous Virtualized Network Function Framework for the Data Center. In *Field Programmable Logic and Applications (FPL), 2017 27th International Conference on*, pages 1–8. IEEE, 2017.
[6] Henning Schulzrinne et al. Real Time Streaming Protocol (RTSP). Technical report, 1998.
[7] Iraj Sodagar. The MPEG-Dash standard for multimedia streaming over the internet. *IEEE MultiMedia*, (4):62–67, 2011.
[8] Marc Snir. *MPI–the Complete Reference: the MPI Core*, volume 1. MIT press, 1998.
[9] Pieter Hintjens. *ZeroMQ: Messaging for Many Applications.* " O'Reilly Media, Inc.", 2013.
[10] Kumar Rakesh et al. Open source solution for cloud computing platform using OpenStack. *International Journal of Computer Science and Mobile Computing*, 3(5):89–98, 2014.
[11] Maxeler Technologies. MPC-X Series. https://www.maxeler.com/products/mpc-xseries, 2015.
[12] Manuel Saldaña et al. MPI as a programming model for high-performance reconfigurable computers. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 3(4):22, 2010.
[13] Alan D George et al. Novo-G#: Large-scale reconfigurable computing with direct and programmable interconnects. In *High Performance Extreme Computing Conference (HPEC), 2016 IEEE*, pages 1–7. IEEE, 2016.
[14] AXI Xilinx. Reference Guide, UG761 (v13. 1). *URL http://www. xilinx. com/support/documentation/ip documentation/ug761 axi reference guide. pdf*, 2011.
[15] D. Sidler et al. Scalable 10Gbps TCP/IP Stack Architecture for Reconfigurable Hardware. In *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 36–43, May 2015.
[16] William Gropp et al. A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard. *Parallel Computing, volume=22, number=6, pages=789–828, year=1996, publisher=Elsevier.*
[17] Edgar Gabriel et al. Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation. In *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*, pages 97–104. Springer, 2004.
[18] Product Guide. LogiCORE IP Soft Error Mitigation Controller v3. 3. *Xilinx Inc*, 2016.
[19] Alpha Data. Alpha Data 8k5 boards. https://www.alpha-data.com/dcp/products.php?product=adm-pcie-8k5, 2017.