ACCL: FPGA-Accelerated Collectives over 100 Gbps TCP-IP

Author(s):
He, Zhenhao; Parravicini, Daniele; Petrica, Lucian; O'Brien, Kenneth; Alonso, Gustavo; Blott, Michaela

Publication date:
2021

Permanent link:
https://doi.org/10.3929/ethz-b-000510849

Originally published in:
https://doi.org/10.1109/H2RC54759.2021.00009

ACCL: FPGA-Accelerated Collectives over 100 Gbps TCP-IP

Zhenhao He, *Systems Group, ETH Zurich*, Zurich, Switzerland, zhe@inf.eth.ch

Daniele Parravicini, *Research Labs, Xilinx*, Dublin, Ireland, danielep@xilinx.com

Lucian Petrica, *Research Labs, Xilinx*, Dublin, Ireland, lucianp@xilinx.com

Kenneth O'Brien, *Research Labs, Xilinx*, Dublin, Ireland, kennetho@xilinx.com

Gustavo Alonso, *Systems Group, ETH Zurich*, Zurich, Switzerland, alonso@inf.eth.ch

Michaela Blott, *Research Labs, Xilinx*, Dublin, Ireland, mblott@xilinx.com

***Abstract*—Collective operations such as scatter, gather, reduce, etc are utilized broadly to implement distributed HPC applications and are the target of extensive optimization in all MPI implementations as well as dedicated collective libraries by accelerator vendors (e.g. NCCL and RCCL by NVidia and AMD respectively). We present ACCL, an open-source FPGA-accelerated collectives library designed to serve applications running primarily in Xilinx FPGAs. Compared to previous collective communication solutions for FPGA, ACCL is flexible and extensible, easily portable, and fast. We evaluate ACCL up to 8 nodes and demonstrate that ACCL outperforms OpenMPI over 100 Gbps TCP-IP for large messages.**

***Index Terms*—FPGA, collectives, MPI**

I. Introduction

Distributed and accelerated computation is essential to many modern HPC and datacenter applications. Distribution is achieved through specialized communication libraries such as MPI, which in addition to simple peer-to-peer communication primitives - send, receive - provide collective communication primitives, which perform complex communication patterns between 3 or more compute nodes.

Collectives are a powerful abstraction which enabled development of GPU-accelerated distributed applications through MPI extensions and dedicated communication libraries for GPU-to-GPU data movement. RCCL [1] and NCCL [2] are vendor-specific libraries from AMD and NVidia respectively which provide efficient implementations of a subset of MPI collectives when applications run on GPU. Optimizations include the ability to perform collectives on GPU memory directly, which greatly reduces the latency of communication since a copy to host memory is avoided.

Recently FPGAs have become available in large-scale compute infrastructure, primarily in the data-center (cloud) context (AWS, Azure) [3]–[5]. Like GPUs, FPGAs enable a programmer to offload part of their application, i.e. a computation kernel, to an accelerator card, where the kernel executes from local memory under host control.

Unlike GPUs, FPGAs have a wide array of interconnection options in addition to computational fabric, and most FPGA accelerator cards are equipped with at least one network port, capable of up to 100Gbps as is the case in high-end Xilinx Alveo accelerators. Despite this excellent opportunity for direct communication between FPGA accelerators in the context of distributed computation, to date the collective communication solutions [6]–[12] for FPGAs are incomplete, providing a limited number of communication primitives, and typically either inflexible, difficult to port between boards and user designs, or slow.

We introduce ACCL, an open-source library for FPGA-accelerated collective communication [13]. ACCL provides support for MPI-like Send, Receive, Broadcast, Scatter, (All)Gather, Reduce-Scatter and (All)Reduce. ACCL is designed to utilize direct FPGA-to-FPGA networking, and implements all communication functionality in a single Vitis [14] kernel which can be linked alongside other user kernels in an Alveo system. ACCL does not require making any changes to user compute kernels, and interfaces with these through Alveo DDR/HBM memory or AXI Streams [15]. ACCL has bindings for both Python and C++, and in addition to the above-listed collectives, provides a set of low-level communication primitives which users can utilize to implement their own collective communication patterns from Python, which can subsequently be ported to execute in-FPGA on ACCLs software-defined control plane for maximum performance.

The contributions of this paper are the following:

- We describe in detail the architecture and implementation of ACCL collectives on Alveo (Sections III and IV), and provide ACCL as open-source code.
- We analyze performance bottlenecks on 100 Gbps Ethernet, and describe optimizations where available (Section V, targeting FPGA experts looking to extend and improve ACCL),
- We compare the performance of ACCL to host-driven collective communication with OpenMPI and Mellanox 100 Gbps NICs on collective microbenchmarks with up to 8 nodes (Section VI).

Our evaluation of ACCL indicates that ACCL outperforms host-driven MPI on messages larger than 1 MBytes, especially when the to-be-collected data is produced in the FPGA itself. As such, ACCL is appropriate for applications where the majority of computation is performed on FPGA. ACCL is also sufficiently open, flexible and programmable to serve as a research platform for network traffic optimization approaches such as application-specific compression and encoding.

II. Background and Related Work

1) FPGAs in HPC: In an HPC context, FPGA acceleration is achieved utilizing PCIe-connected accelerator cards, each equipped with multiple types of on-board memory and computational resources. The Xilinx Alveo range of PCIe accelerators are each equipped with one FPGA device, which includes computational fabric (Look-up tables and DSP processors) and a small amount (tens of MB) of on-chip memory. Additionally, two Alveo boards (U280 and U50) are equipped with 8GB, 32-channel HBM memory, while the other two (U200 and U250) are equipped with 64GB, 4 channel DDR4 memory. All Alveo cards are equipped with QSFP28 transceivers and MACs capable of implementing 100 Gbps Ethernet connectivity and integrated directly into the computational fabric.

To harness the Alveo memory and computational resources, software developers utilize the Vitis and Vivado tools which enable them to describe the computational functionality to be implemented in FPGA, i.e. computational kernels, specify the desired connectivity between kernels, then compile the kernels and link them into an executable file which is utilized to configure the Alveo accelerator. Similar to a GPU acceleration flow, developers can utilize pre-defined kernels from vendor-provided acceleration libraries. However, unlike GPU kernels, FPGA kernels can exchange data not only through shared memory, but also through physically implemented stream (pipe) interfaces which carry data directly from one kernel to another, either via internal FPGA interconnect or the QSFP28 transceivers, allowing for very low latency, power efficient information exchange.

Once the Alveo is configured with the compiled kernels, the users can access, configure, and schedule work to FPGA-resident kernels using the Xilinx RunTime (XRT) library. XRT provides bindings for C++ natively and Python either natively or through the PYNQ library, which provides several Pythonic execution features such as kernel dependency scheduling through Python futures. User applications build on top of these bindings to access the FPGA computation and memory resources.

2) MPI for GPU Architectures: As the large-scale computing infrastructure becomes more heterogeneous with the emergence of compute accelerators, MPI implementations are becoming more accelerator-aware. Many MPI implementations can target GPU memory [16]–[18]. In such a GPU-extended platform, MPI relies on a commodity NIC to transfer network packets and host CPUs for orchestrating and synchronizing the data movement between network and the accelerator typically via host memory. In GPU-specialized MPI-like extensions, such as NCCL [2] and RCCL [1], the network data can be directly forwarded to the GPU memory from the NIC via the local PCIe switch instead of temporary buffering in the CPU memory, reducing latency.

3) MPI for FPGA Architectures: In contrast, the architecture of an collective implementation targeting FPGA could be very different since the FPGA can directly process the network packets, as indicated by projects such as Microsoft Catapult and Brainwave [3], [4], [19]–[22]. Therefore, it doesn't require a commodity NIC on the network data path. Collective offload to the FPGAs with large scale infrastructure is possible due an emerging research effort of communication protocol offload, e.g., UDP [23], TCP/IP [24]–[28] and RDMA [29], [30], to the FPGAs. Despite that a few research efforts have been made for collective systems targeting FPGAs, scale-out of FPGA-accelerated execution is still commonly achieved by calling a communication-specific, FPGA-agnostic library such as OpenMPI, which orchestrates host CPUs, memory and NICs to connect nodes together over Ethernet or Infiniband. This is because all existing FPGA-based collective solutions fail to satisfy the following three requirements at the same time. (1) The platform should provide *high performance* in terms of throughput (targeting 100 Gbps network bandwidth). (2) It should be *flexible* in terms of runtime configuration to run different collectives without reprogramming the FPGA. (3) It should be *portable*, which means that the collective should run on top of transmission protocol that is widely available in large-scale infrastructure, e.g. TCP/IP or RDMA, and should not be constrained to specific FPGA boards. For instance, Galapagos [9], [31] is a communication infrastructure targeting FPGA cluster with 10 Gbps hardware TCP/IP stack but they provide only send-recv primitives. EasyNet [6] is a collective library running with 100 Gbps hardware TCP/IP stack, but it doesn't have runtime flexibility. TMD-MPI [8] runs MPI implementation on top of embedded processors on FPGAs, which gives high flexibility while it targets only 1 Gbps throughput. SMI [7] propose a collective library that allows streaming message passing between computation and communication, but it is not portable since it use a customized protocol targeting FPGAs cluster with fixed topology.

TABLE I: ACCL compared with existing FPGA-based collective solutions.

Solution	Performance	Flexibility	Portability
Easynet [6]	High (~90 Gbps)	Low	High
SMI [7]	Medium (~40 Gbps)	Low	Low
Galapagos [9], [31]	Low (<10 Gbps)	Low	High
ZRLMPI [10], [32]	Low (<10 Gbps)	Low	High
TMD-MPI [8]	Low (<10 Gbps)	High	Low
ACCL (Ours)	High (~80 Gbps)	High	High

As summarized in Table I, all existing solutions have their own limitation. Therefore, we propose ACCL, which is a high performance, flexible and portable collective library for FPGAs. The purpose of ACCL is to provide the FPGA kernel(s) and runtime support for FPGA-accelerated applications to communicate without host involvement and exchange FPGA-resident data directly while maintaining MPI-like semantics familiar to HPC developers.

III. ACCL: FPGA-Accelerated Collectives

The Accelerated Communication Collective Library (ACCL) provides MPI-like primitives that allows users to orchestrate data exchange between multiple FPGAs. ACCL is

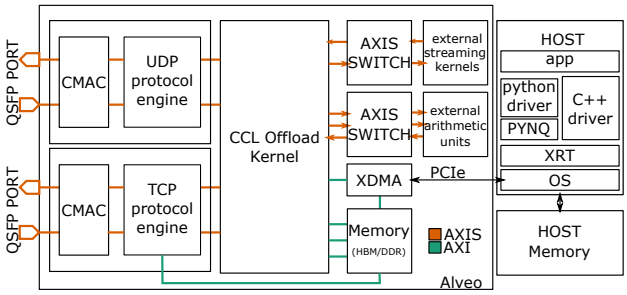


Fig. 1: System view of ACCL

a combination of software running on the host CPU, FPGA data-moving hardware, and control firmware executing on a FPGA-embedded microcontroller. Figure 1 illustrates an high level overview of the ACCL structure. In the FPGA, ACCL features a collectives offload engine (CCLO) and one or more network protocol offload engines (POE), each of which is implemented as a stand-alone Vitis kernel. The CCLO implements the collectives on top of TCP or UDP. The protocol offload engines implements the full network stack up to UDP and TCP respectively and connect directly to Ethernet ports, e.g. through Alveo Gigabit Transceivers and QSFP28 ports. For TCP, the protocol offload kernel requires access to external memory to reorder incoming packets. The host communicates with the CCLO over PCIe and Xilinx XDMA, but this complexity is hidden by XRT and our drivers. The distributed application that runs on, possibly multiple, hosts leverages the ACCL Python or C++ driver to control the CCLO. In this way, the user offloads to ACCL, via MPI-like collectives, data distribution, results collection or combination over multiple FPGAs. We provide the following collectives: send, receive, broadcast, gather, scatter, (all)gather, (all)reduce, reduce-scatter.

A. Host Software Stack

The host software stack, illustrated in Figure 1, consists of user application code which binds to ACCL drivers, through Python or C++ bindings, to initialize and invoke ACCL collectives. The drivers themselves rely on the Xilinx Run-Time (XRT) and associated libraries, and OS services, to communicate with the CCLO in hardware. ACCL is designed to provide a dedicated high-speed connection between FPGAs and does not provide general-purpose communication over TCP or UDP as a traditional network stack would. Instead, the host must provide a secondary, possibly low-speed connection to other ranks over which the user application can be launched in a distributed setting via e.g. *mpirun*.

The ACCL drivers expose three distinct APIs to the user application: a *housekeeping* API which enables CCLO configuration and monitoring, a *primitives* API consisting of simple data movement operations (send, receive, copy, reduce), and a *collectives* API which exposes the ACCL performance-optimized collectives. If users require specific collectives not supported by the ACCL collectives API, these can be assembled utilizing the primitives API in Python or C++.

We have aligned our API to MPI naming conventions to facilitate code migration to ACCL if the original application was already written using MPI for communication. As such, FPGAs involved in the communication are called ranks and are grouped in communicators. Listing 1 illustrates the three APIs - the code initializes ACCL, invokes the ACCL send/recv primitives to exchange data between ranks 0 and 1, and executes an allreduce collective on all ranks.

Listing 1: ACCL host code in Python

```
1  from pynq import Overlay, allocate
2  from mpi4py import MPI
3  #receive binfile, ranks_dict as inputs
4  ol = Overlay(binfile)
5  accl = ol.cclo
6  rank = MPI.COMM_WORLD.Get_rank()
7  bs = 16384

9  accl.setup_rx_buffers(nbufs=16, bufsize=bs,
       devicemem=ol.bank0)
10 accl.configure_communicator(ranks_dict, rank)
11 accl.open_port(); accl.open_con()

13 txb=allocate((bs,), target=ol.bank0)
14 rxb=allocate((bs,), target=ol.bank0)

16 if rank==0:
17     accl.recv(rxb, src=1, tag=1, to_fpga=True)
18 elif rank==1:
19     accl.send(txb, dst=0, tag=1, from_fpga=True)

21 ch = accl.allreduce(txb, rxb, count=256, async=
       True)
22 accl.allgather(txb, rxb, count=256, waitfor=[ch])

24 accl.deinit() #releases FPGA memory, resets CCLO
```

ACCL is initialized by functions (lines 9-11) which allocate and configure a set of buffers required for ACCL operation in the FPGA memory (detailed in Sec III-E), and construct the communicator according to rank information. In this example, we utilize the python package *mpi4py* to determine the local rank ID when the application has been launched with mpirun. All configuration information is offloaded to the FPGA so that the CCLO can rapidly access them. Afterwards, the user can issue commands to open connections between each ranks in the communicator via the protocol offload engine.

As with MPI, most ACCL collectives take two buffers as arguments, one (source buffer) holds the to-be-communicated data (line 13), while the the other (destination buffer) specifies where to store the results (line 14). Lines 16-19 implement data movement from rank 1 to rank 0 utilizing the primitives API. Each ACCL function allows users to specify whether each of the buffers reside in the host or in FPGA external memory (*to_fpga/from_fpga*). If required, the ACCL driver handles data movement between host and FPGA external memories. However, when ACCL is used in conjunction with user FPGA kernels, the additional data movement is not required, reducing latency.

Lines 21-22 execute collectives on the entire communicator, with the collectives API. When multiple collective primitives are issued in a single application, additional latency reduction can be achieved by utilizing asynchronous calls and call

chaining, a feature of XRT which the ACCL drivers expose up to application code. When an ACCL collective is called asynchronously, it returns immediately a handle to the XRT descriptor (*ch* on line 21) of the on-going collective. Chaining occurs when the invocation of one FPGA kernel depends on the completion of a different invocation to the same or other FPGA kernel. ACCL collectives also allow for these dependencies to be specified (see line 22) and passed down to the Xilinx Embedded Run-Time, a hardware scheduler which starts the collective immediately after the dependencies have been resolved, with typical latency in the nanosecond range. Using these mechanisms, users can define execution chains of arbitrary length which involve any ACCL collective and any user FPGA kernels.

B. CCLO Kernel

The CCL Offload (CCLO) kernel implements the ACCL primitives by orchestrating data movement between the network fabric, FPGA external memory, and FPGA compute kernels, with no host CPU involvement. Data movement to and from the network is accomplished through custom interface blocks to the TCP/UDP network protocol offload engines, while FPGA external memory (DDR or HBM) is read and written through DataMover engines (DMA).

Each primitive typically requires multiple transfers in specific sequences between ranks to achieve the desired result. Orchestrating the required transfers and the interaction between various subsystems at high speed is a challenge. For this reason, the CCLO kernel consists of two subsystems: a software-programmable control plane which is extremely flexible, and a high-throughput data plane consisting of DMAs, configurable routing, and pipelined arithmetic, which is fast. Different parts of the data plane can be activated at the same time in parallel, e.g. to transmit and receive data simultaneously from the network. Section IV describes the implementation of the CCLO kernel in more detail.

The CCLO data plane is datatype-agnostic for non-reduce collectives and supports in principle any user-defined datatype for sum reductions. In this work, we have provided support for single, and double precision floating point numbers, and 32- and 64-bit integers. However, the data plane can be augmented with additional arithmetic and custom streaming kernels, as illustrated in Figure 1. As such, with the addition of corresponding external logic ACCL can be extended to supports not only arbitrary datatypes, but also arbitrary reduction functions and conversions between types. Furthermore, the external data plane extension capability enables the CCLO kernel to source and sink data from/to user FPGA kernels via AXI Streams instead of external memory, reducing latency.

C. Protocol Offload Kernel

The protocol offload kernel contains hardware communication stacks that work on top of UltraScale+ Integrated 100G Ethernet Subsystem (CMAC) [33]. Currently it supports both UDP and TCP/IP and it can be extended in the future to support other protocols, such as RDMA. We utilize open-source protocol offload engines for UDP [23] and TCP/IP [24], [26], both of which can process data at 100 Gbps line rate and are integrated into the Xilinx Vitis development framework [25]. The TCP/IP network stack supports up to 1,000 connections and can be configured to support window scaling and out-of-order packet processing. The TCP/IP stack requires two hardware memory spaces, which serve as temporary buffers for re-transmission of Tx data and buffering of Rx data respectively.

We note that all of the functionality of ACCL is available with either one of the protocol offload kernels, therefore in most user systems, only one offload kernel will be present, to reduce resource utilization. However, the CCL offload kernel supports simultaneous transmission and reception from both offload kernels.

D. ACCL Message Protocol

In ACCL, we implement a light-weight communication protocol above the transport layer protocol (e.g., TCP/IP) to carry metadata information for each message. Each ACCL-emitted message consists of a 512-bit (64-byte) header and the payload consisting of user data. The header contains the rank IDs of the message source and destination, message length and tag, a sequence number which is used to keep track of the order of the messages, and padding. The 64-byte header represents one word of the CCLO datapath and is therefore convenient to insert in the message without additional logic overhead, but may reduce performance on small messages. Future versions of ACCL may implement smaller headers.

E. ACCL Memory Requirements

In the current implementation, ACCL utilizes an eager protocol to communicate between ranks, i.e. message data is sent as it becomes available. The eager protocol potentially minimizes message latency, however in a distributed setting it is possible for a message to arrive before the user at the destination rank has allocated a destination buffer for it, i.e. before the user has *posted* the corresponding receive. To handle this scenario, ranks are responsible for storing received messages until the user requests them. As a consequence, message reception is divided in two steps. First, received messages are moved in a staging area in DDR or HBM. Each message is stored individually in what we call a *spare receive buffer*. Then, when the corresponding receive is posted by the user, data is moved into the user buffer. This approach requires to allocate some memory in advance for ACCL spare buffers. The number and the size of spare buffers can be configured at initialization through the ACCL housekeeping API.

IV. Collectives Implementation

A. Data Plane Structure

Each ACCL primitive and the collectives assembled from the primitive represent sequences of operations involving the CCLO control and data planes. This section describes in detail the hardware structures and sequences of operations involved, demonstrating the flexibility of the ACCL hardware, which

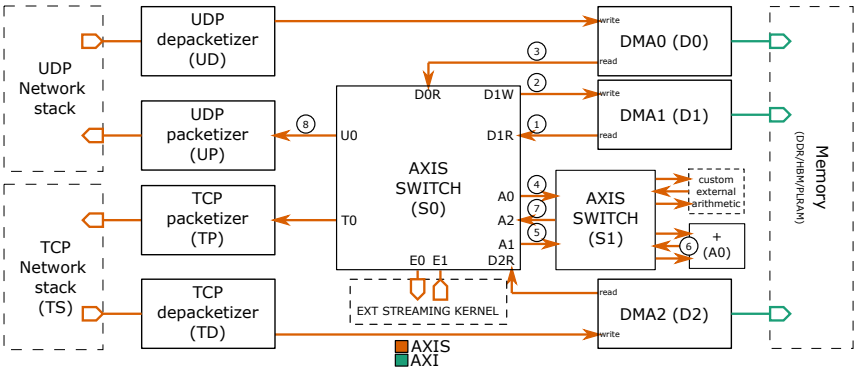


Fig. 2: Schematic view of CCL Offload Data plane and datapaths.

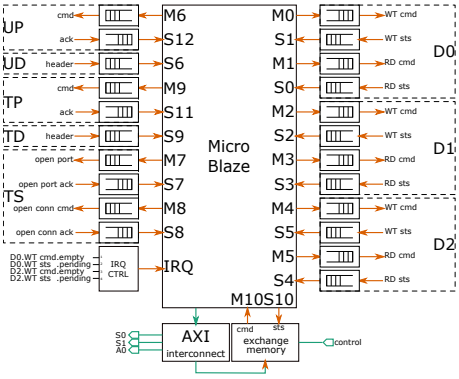


Fig. 3: Schematic view of CCL Offload Control plane.

could be extended by expert users to perform collectives not currently supported.

Figure 2 presents a high level overview of the CCLO data plane structure which consists of multiple functional units (FUs) - three AXI DataMover engines (DMA0, DMA1, DMA2), AXI Stream (AXIS) interconnects (e.g. S0, S1), an internal arithmetic unit (A0) and network interface logic (UD, UP, TP, TD). Data flows through all those components via 512 bit wide AXIS interfaces [15], that are connected together via the central AXIS Switch.

AXI DataMover engines (DMA) provide high throughput data transfer between AXI memory mapped (AXIMM) and AXI stream (AXIS) domains. Each DMA is connected to FPGA external memory via AXI interface [34] and provides two channels dedicated to read (memory to stream) and write (stream to memory) operations respectively. Each of the channels is independently controlled through a pair of AXI Streams, carrying DMA commands and command acknowledgements respectively.

The network interface logic decouples data movement and the network offload engines. The communication is accomplished by means of custom HLS blocks - for each protocol, one packetizer and one depacketizer. The packetizers (UP, TP) are responsible for inserting the ACCL message header into the stream, dividing up the stream into individual packets of pre-set lengths, and forwarding the packet to the respective protocol offload kernel. The depacketizers (UD, TD) perform the reverse operation. The AXIS Switch (S0) provides a flexible interconnection in the data plane. S0 is programmable via AXI-mapped registers to implement any connectivity pattern between its inputs and outputs. Therefore, data plane components are not arranged in a fixed datapath but rather the order in which data traverses different components is set at run-time by the control firmware. However, we highlight that received data, via UD and TD, is served by a fixed datapath that connects the depacketizers directly to the DMA engines via dedicated streams.

B. Control Plane Structure

Figure 3 illustrates the detailed structure of the control plane. Central to the control plane is a MicroBlaze single-core, FPGA-optimized microprocessor [35]. The role of the Microblaze CPU in the CCLO design is to provide the required flexibility to execute many different collectives utilizing a multitude of FUs in combination, which is difficult to achieve utilizing logic described in HLS or RTL. Firmware executing on the Microblaze is able to generate commands to the various FUs - the DMAs, (de)packetizers, switch, etc. These commands can be assembled into *data movement primitives*, e.g. copy, transmit to network, which themselves are assembled into collectives. By implementing the control in software (compiled C code), the control logic can run at fairly high frequencies (up to 250 MHz) with reasonable resource utilization [36] and can also be adapted and improved over time with relative ease, without requiring re-synthesis of the CCLO kernel. Compiling and debugging the firmware is possible utilizing Vitis and the Xilinx command-line tool XSCT, which enables all common software development features - breakpoints, step-by-step execution, profiling and others.

However, the latency of firmware-directed control is higher when compared to RTL control logic. To counteract the higher latency of the MicroBlaze we employed a pipelined control plane architecture as illustrated in Figure 3 whereby most functional blocks are controlled through hardware First-In-First-Out (FIFO) memories, decoupling control issuing from the execution. Less performance-sensitive FUs (AXI Stream switches, arithmetic) are controlled through an AXI bus using register-mapped interfaces. These AXI interfaces to the FUs are utilized at most once per collective execution, to set long-duration configurations, such as the reduction function, network MTU, etc.

Interaction with the host is achieved through a call controller and shared mailbox memory (exchange memory module), which are accessible by both the MCU and the host. The call controller is an HLS IP which implements the host-facing handshake protocol allowing the CCLO to appear to the Xilinx Run-Time (XRT) as a call-able Vitis kernel. The 4KB mailbox

memory is accessible by both the host and Microblaze and enables large and stable configuration parameters to be set by the host and used by the firmware. These structures include: a list of spare buffers to be used as receive memory by the CCLO, a list of communicators describing rank information (IP addresses, ports, session IDs), data type and compression scheme information, and error reporting registers utilized for CCLO debugging.

C. Receive Pipeline

As described in section III-E, ACCL utilizes an eager protocol for message transmission which requires receivers to typically divide message receipt in two steps. First, incoming messages are moved to a spare buffer, i.e. a staging area in DDR or HBM, via the receive datapaths from depacketizers to D0/D2 write channels. Subsequently, when the ACCL user has posted the receive or indicated the desired collective, the MicroBlaze issues commands to move data from the spare buffer to an appropriate destination. To avoid applying back-pressure to the network fabric, the write channels of the receive datapath are primed ahead-of-time by the Microblaze with commands via RX command FIFOs M0, M4. As these commands are consumed by incoming messages, the write channels issue status words into FIFOs S1, S5. Upon their reception, the firmware interprets the statuses and marks the corresponding spare buffers as reserved, to avoid issuing further commands to the same addresses. The firmware also collects received message headers from the depacketizer status FIFOs S6, S9 which are saved as spare buffer metadata. To minimize latency and avoid RX pipeline starvation, the occupancy of both command and status FIFOs are monitored by hardware and interrupts are generated to the MicroBlaze when either the command FIFOs become under-full or the status FIFOs becomes non-empty. We note that one incoming message may be split among multiple spare buffers if the message size is larger than the maximum size of a DMA transfer, which is configurable up to 8MB.

In the current implementation of the ACCL receive pipeline, simultaneous interleaved receive of multiple messages is not supported. To ensure this scenario is not encountered in practice, we implement collectives with ring-based algorithms that ensure each rank's fan-in is at most 1, as will be described in subsequent sections.

D. Data Movement Primitives

The primitive functionalities that the CCLO implements are: copying buffers to and from FPGA external memory, sending and receiving data to and from the network fabric, and performing reduction operations on data. The following detailed description refers to blocks indicated in fig. 2. In ACCL, these primtives are implemented in Microblaze firmware but are also exposed as part of the user-facing Python API, allowing users to create complex communication behaviour.

Copy: To copy a buffer from one address (addr0) to another (addr1), the MicroBlaze first activates the copy datapath connecting D1R to D1W by programming the switch S0, then emits commands to D1 channels via the corresponding FIFOs M2 and M3. D1R reads from addr0 and D1W writes to addr1. In the process, data gets loaded in stream (1) and traverses the AXI switch to stream (2). DMAs signal completion by emitting a status words into FIFOs S2 and S3. The firmware polls these FIFOs until both statuses are received.

Send: To send a buffer to a different rank over the network, the MicroBlaze configures the switch S0 to connect D1R to T0 or U0 depending on the desired protocol TCP or UDP. A packet header is sent to the relevant packetizer TP or UP and then D1 is commanded to read the required bytes from the FPGA external memory. The data read from D1 (1) flows through the S0 to the appropriate packetizer (8) which prepends the message header and segments the message into packets of size equal or less to the network MTU. The protocol offload engine handles the rest of the transmission process independently.

Receive: As the receive pipeline is automated by the mechanisms described in the previous section, the receive primitive involves identifying whether a required buffer has been received and where it is located, and if necessary waiting for it to arrive. When the receive request is issued, either by the host or by the firmware as part of a collective, the MicroBlaze will scan through the reserved spare buffers to perform matching based on the source rank and tag in the saved headers. If no match is found, the firmware re-attempts the matching until either the message arrives, or a timeout is triggered. As part of the execution of the receive primitive, the message data may be subsequently copied to a user-accessible buffer. In case the message had been segmented between multiple spare buffers, a DMA gather will be performed utilizing D1. The datapath for receive is configured exactly as it is for Copy, since the dataflow is physically identical.

Reduction: To apply a reduction function to two buffers (addr0, addr1 respectively), the MicroBlaze programs S0 to activate the reduction datapath which connects D0R and D1R DMA read channels to S0 outputs A0 and A1, corresponding to inputs to reduction logic, and output D1W to input A2, corresponding to reduction result. The MCU then commands D0, D1 to read from addr0 and addr1 respectively and D1 to write results in another address. The stream from D0(D1) read enters S0 (1)((3)) and exits at A0 (4)(A1 (5)). Note that additional routing logic e.g. S1 enables multiple arithmetic units to be instantiated and utilized for reduction. These can implement various reduction functions (sum, maximum, elementwise) or the same function for multiple data types. By default, CCLO supports sum reduction only.

E. Collective implementation

ACCL collectives are implemented by assembling the primitives described earlier. We note that in ACCL there are two methods of assembling collectives: host-side collectives are assembled by combining host-side calls to ACCL primitives API, and are therefore flexible but slow, while firmware collectives assemble primitives in firmware code, increasing performance.

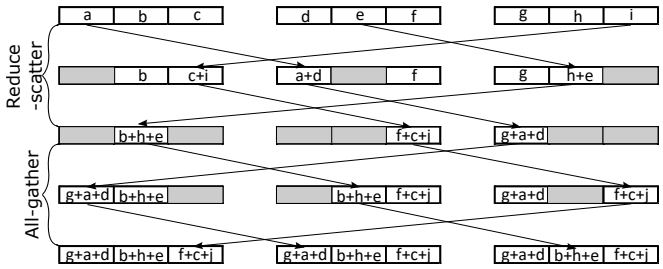


Fig. 4: Allreduce algorithm graphical representation. Grey boxes indicate non valid data.

Broadcast: The root rank j sends its buffer to all the ranks in the communicator. Each rank $i \neq j$ receives from rank j in its destination buffer.

Scatter: The root rank j divides the source buffer into n chunks. It then sends the i-th segment of the buffer to rank i. Finally it copies chunk j its destination buffer. Each rank $i \neq j$ receives from rank j in its destination buffer.

Gather: The non root ranks send their source buffer and relay the buffer of the others - using send and receive primitives- in a clockwise ring fashion toward the root rank. The root rank (with ID i) receives $n-1$ buffers and fills the destination buffer in a anticlockwise fashion from $(i-1)$-th segment to segment 0 and to segment $n-1$ to segment $(i+1)$. The root rank then copies its source buffer to destination buffer at i-th segment offset.

Reduce: Reduction is implemented by forwarding data from one rank to the other towards the root, which performs accumulation.

Allgather, Reduce-scatter, and Allreduce: We implemented a ring based version of the allreduce algorithm presented in [37]. The algorithm segments each buffer in n smaller buffers (segments or chunks) where n is equal to the number of ranks involved in the collective. The algorithm is divided in two steps: a ring-based reduce-scatter followed by a ring all-gather. The advantage of this algorithm is that in any moment, each rank processes a different part of the the buffer, so as to take advantage of all the arithmetic units available. The second benefit of this approach is a reduction by a factor of n in the size of the intermediate messages to be exchanged by the ranks. Finally, the same firmware control code can implement either all-gather (by skipping the reduce-scatter), allreduce or reduce-scatter (by skipping the all-gather).

Figure 4 shows a graphical representation of the allreduce algorithm with 3 ranks. The reduce-scatter step is to compute one of the n segments of the final buffer in each rank. In the allgather step, each rank will send its reduce-scatter result in the ring (and relays the result of the others) such that every rank ultimately gathers all the reduced segments.

V. Performance Tuning

A. Message Segmentation and Memory Organization

In the simplest implementation of the receive pipeline, the receiving data from a given message is temporarily stored in a single spare buffer which is not read for further processing until the whole message is delivered. This store-and-forward behavior of the message creates latency overhead, especially for large messages. Therefore, one optimization is to segment the original large message into several small messages, and store them into different spare buffers such that the receiving of the message segment can be overlapped with the consumption of other message segments. Theoretically, this approach gives increasing performance with finer granularity of segmentation. However, in ACCL, two factors limit the achievable speed-up through segmentation. First, the consumption of each spare buffer interrupts the MicroBlaze which needs to collect information about the received message and schedule a new write command. This creates certain amount of overhead which is difficult to avoid. Secondly, because each segment of the message is prepended its own header, as granularity increases the ration of header to payload increases and therefore the efficiency of network fabric utilization decreases. Therefore, the segmentation size of the message should be set to an adequate large number which balances out these effects to achieve optimal performance.

As we segment the message into different spare buffers, two spare buffers could be activated at the same time, receiving data from the network and reading other data for further processing. In the most simple implementation of the receive subsystem and CCLO system design, all the spare buffers are assigned to single memory channel. However, each activated buffer requires a memory bandwidth of 100 Gbps to saturate the network throughput, yielding an aggregated required bandwidth of 200 Gbps, which is beyond the typical achievable bandwidth of a single memory channel on FPGA (about 128 Gbps). Therefore in some execution scenarios the memory bandwidth limitation creates backpressure from the DMAs through the datapath and out into the protocol offload engine, creating challenge for the underlying hardware network stack. In case of UDP, this leads to unrecoverable packet loss. Despite that the TCP/IP is a reliable transmission protocol with proper flow control mechanism, packet drop could still happen when the CCLO applies backpressure, which in turn might trigger expensive re-transmission timeout in the TCP/IP.

The solution to this problem is to allocate spare buffers in multiple memory channels in an interleaved fashion, such that consecutive access of the spare buffers are served with high probability by different memory channels to minimize the interference between memory access and network protocol functionality. In the state-of-the-art FPGA boards, there are commonly more than one DDR channel (Xilinx U250 has 4 DDR channels), and some are equipped with high-bandwidth-memories (Xilinx U280 has 32 HBM channels) [38]. However, some of these channels may be required for other user computation kernels, while in the case of DDR channels, each additional channel has a resource utilization cost of approximately 50 kLUTs and increases the difficulty of timing closure, therefore we are interested in the minimum number of memory channels required to effectively remove the identified memory bottleneck.

B. Network Stack Tuning

To better cope with the potential backpressure due to spare buffer interference, the underlying hardware TCP/IP stack should also be tuned to transfer data in a more conservative way. One factor is the window scaling option, which increases the receive window size allowed above the actual size of the receiving buffers (64 KB) within the TCP/IP stack. It is widely adopted in a network with high transmission latency to compensate the bandwidth-delay-product. However, this needs to be revisited in a high bandwidth network with low latency hardware network stack. In ACCL, 100 Gbps bandwidth with about 5us round-trip-time yields 62.5 KB bandwidth-delay-product, reaching the actual receiving buffers in the network stack. Therefore, we disable window scaling to avoid the unnecessary packet drop due over-sizing the receiving window.

C. Non-arithmetic Collectives

Our optimizations have so far targeted the receive subsystem, to better deal with pressure from incoming network data. This pressure can also be reduced by modifying the scheduling of data transmission at the origin of the transaction, especially in single-source non-arithmetic collectives such as broadcast and scatter collectives. During these, the root rank sends a number of bytes to all the other ranks. The naive implementation of these collective sends the entire buffer to one destination rank before moving to the next one. To increase data locality and to decrease network bandwidth consumption we can segment the buffers into smaller chunks and transmit to each rank in a round robin fashion until there is no segment left. For the transmitter there is no significant difference in the network stack load, however at the receiver the data is spaced out over a larger time.

VI. Evaluation

The objective of our evaluation of ACCL is two-fold. Firstly, we aim to identify the effect of various ACCL tuning parameters (segment size, number of memory banks, type of memory) on the performance of ACCL. Secondly, we aim to compare ACCL to host-driven MPI in scenarios where data originates in host and FPGA memory respectively, over a range of user-configurable application parameters, notably message size and number of ranks.

A. Set-Up and Methodology

We evaluate the ACCL on the ETH Zurich Xilinx Adaptive Compute Cluster (XACC [39]), which consists of four FPGA-equipped servers, each of which has a mix of Alveo U250 and Alveo U280 cards. Each Alveo card features two 100 Gb/s Ethernet interfaces: one is connected to a Cisco Nexus 9336C-FX2 switch and the another is connected to its Alveo neighbor. Every server also has two 100 Gb/s Mellanox Network Interface Card (NIC) connected to the switch. This configuration allows users to explore arbitrary network topologies for distributed computing.

In this work we utilize both U250 and U280 Alveo cards to demonstrate the capabilities of ACCL. Each XACC server partitions its resources to several VMs such that each VM has acces to one Alveo card and one Mellanox NIC, for a total of 8 usable FPGA-accelerated virtual nodes in the XACC, equally distributed between U250 and U280 cards. The VM software environment is based around Ubuntu 18.04 and includes many software frameworks for FPGA accelerator deployment: XRT, PYNQ for Alveo, Jupyter Lab, OpenMPI.

In our evaluation we utilize only the through-switch connections and the TCP network stack to evaluate the achievable throughput and latency for Send/Recv and each supported ACCL collective and compare against the same metrics measured on OpenMPI 4.1.1 over the Mellanox 100 Gbps NICs. We utilize messages ranging in size from 1KB to 32MB, and for collectives we also evaluate communicator sizes from 3 to 8 ranks (maximum achievable at XACC) to determine the effect of scale on the throughput and latency. The maximum transfer unit is set to 2 KB for both the software and hardware TCP/IP stack. We tune the network buffers in the Linux network stack to be 32 MB to accommodate the experiment requirement. We do not enforce the implementation algorithm for collectives in OpenMPI, i.e. we let the library self-configure. ACCL utilizes rings for all gather and reduce collectives as described in previous sections. We execute each experiment 100 times in order to calculate accurate average runtimes as well as to estimate variability (jitter).

We evaluate both ACCL and OpenMPI for host-to-host and FPGA-to-FPGA communication. In all figures following, H2H and F2F indicate experiments where source and destination data buffers are in host and FPGA memory respectively. F2F is relevant for applications producing data in FPGA, while H2H is relevant for host-executing applications. In the OpenMPI F2F and ACCL H2H scenarios, XRT is utilized to move data to and from the FPGA.

TABLE II: Resource utilization of ACCL components

Component	kLUT	DSP	BRAM18	URAM
CCLO	78	56	169	0
TCP POE	111	0	813	1
UDP POE	23	0	115	0
CMAC	12	0	34	9

B. Resource utilization

The resource utilization of ACCL components is listed in Table II. The majority of resource utilization is dedicated to the protocol engines, in particular the TCP POE. The CCLO utilizes relatively few LUT and memory resources but also requires DSPs for floating point arithmetic. However, resource-conscious users can easily modify the CCLO to eliminate support for datatypes not needed in their applications. We estimate a requirement of approximately 32 DSP per reduction datatype supported. Readers should also note that only one of the UDP/TCP POEs are required for ACCL operation.

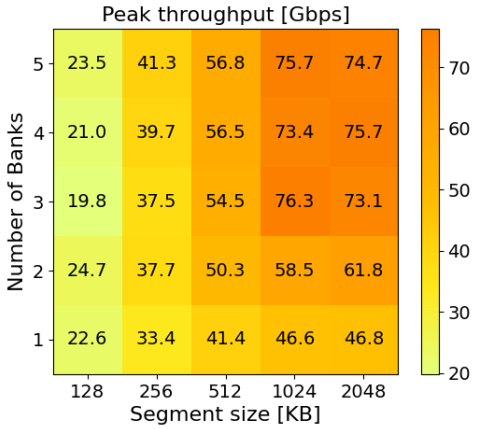


Fig. 5: Effect of segmentation and channels on throughput.

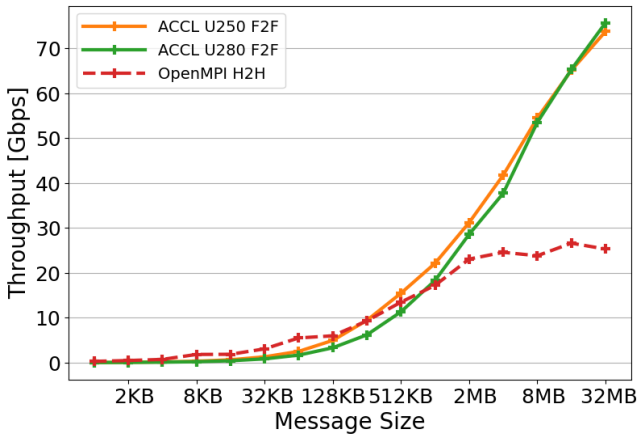


Fig. 6: Send/Recv throughput comparison.

C. Microbenchmark Results

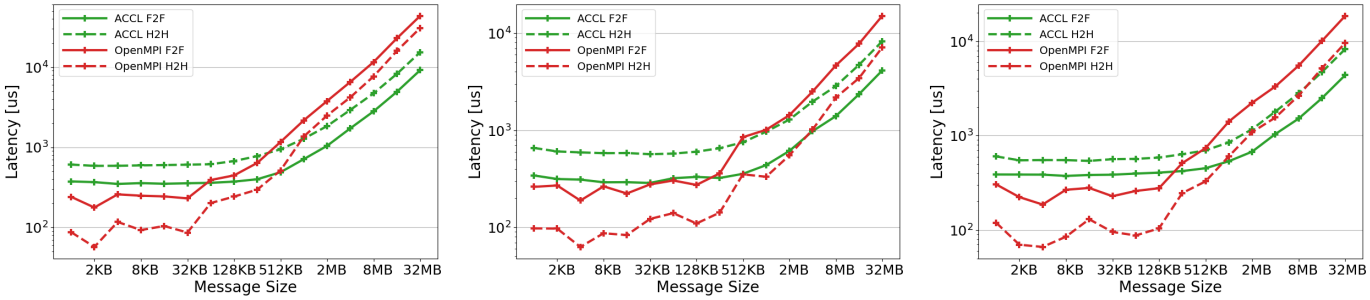
1) Send/Recv: We examine the efficiency of our control and data paths and the effects of various tuning parameters with the Send/Recv primitive. Its performance is most indicative of achievable throughput as the communication pattern is very simple - one rank sends, another receives. Figure 5 is a composite analysis of the effects of segment size and number of memory channels on achievable send/receive throughput, where each cell indicates the maximum achieved throughput for the respective combination of segment size and memory channels, at any message size. Various configurations may reach their peak at different message sizes. We utilize the Alveo U280 for this experiment.

Effect of Segmentation Size: Focusing on the horizontal axis of Figure 5, we observe that larger segments enable ACCL to achieve higher peak throughput. Although in theory finer-grained segmentation should increase performance, in practice the additional control overhead of scheduling receive buffers for the large numbers of resulting segments nullifies any gain. Smaller segments do have a slight edge for small messages, where the total number of segments per message is less. We conclude that segmentation is beneficial, and optimal segment size is 1 MByte. For 2 MByte or larger segments performance begins to decrease.

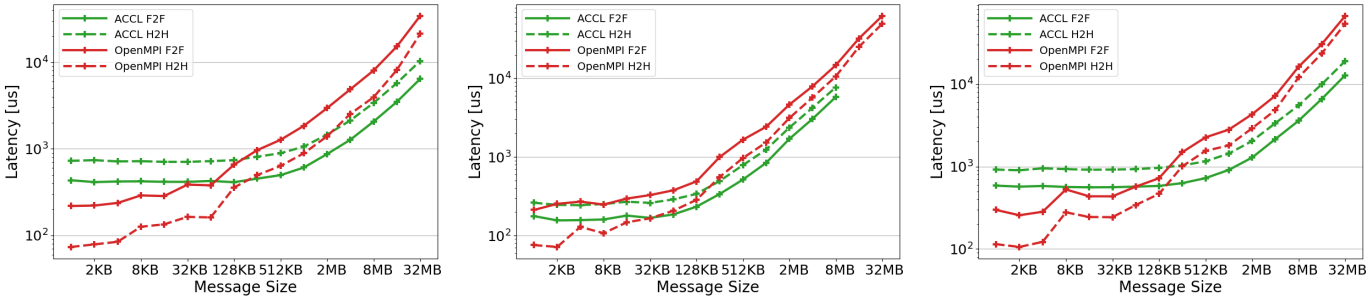
Effect of Number of Memory Channels: On the vertical axis of Figure 5 we observe the dependency of peak throughput on the number of memory channels allocated to the receive pipeline. We observe a significant speedup with the increase of the number of channels at all message sizes beyond 2MB, with peak observed throughput increasing from 50 Gbps at 1 channel to almost 80 Gbps at 3 channels and saturating thereafter, especially for large segments. This result indicates that it requires only 3 independent memory channels to reach the ACCL peak, which is practical even with non-HBM cards. As a result of this finding, our ACCL designs for the Alveo U250 utilize all four DDR interfaces: 3 for ACCL and one for the TCP protocol engine.

Portability and Comparison with OpenMPI: Figures 6 compares the throughput of ACCL with best configuration running on U280s and U250s and the software OpenMPI. Reflecting our findings in the previous sections, for both U280 and U250, the segmentation size is set to 1 MB. For the U250, we assign 3 DDR channels to ACCL while we assign 5 HBM channels on U280. Despite the board difference and slight difference in configuration, the performance on U280 and U250 is similar, demonstrating that our design is portable across different boards. We observe that ACCL delivers a higher peak throughput ($\tilde{7}6$ Gbps) than its software counterpart ($\tilde{2}5$ Gbps). We believe this is because OpenMPI uses a single TCP/IP connection for point-to-point communication, mapping to a single CPU core, and the TCP packet parsing overhead outweighs the processing capability of the single core. In contrast, despite ACCL having a single data path for all packets, it is highly pipelined to hide the packet parsing overhead, and does not require any processing by the host CPU.

2) Collectives: Figures 7 present microbenchmark results for collectives running on four nodes. ACCL results were gathered with Alveo U280 and U250, which performed similarly so U250 data was omitted for clarity. We observe that generally ACCL outperforms software MPI at large message sizes (over 1MB roughly) but not for smaller messages. This is because ACCL has larger invocation overhead, which can not be amortized with small message size. Across the range of message sizes, buffer transfers between the host and FPGA cause approximately 300 microseconds of fixed overhead for both ACCL and OpenMPI execution, which is expected since both rely on XRT for data transfers between host and FPGA. In addition, ACCL experiences an additional 150 microseconds of overhead, caused by the latency of initiating the call to the CCLO kernel in the FPGA through XRT. Among all the collectives, scatter and Gather are the most competitive for OpenMPI while Broadcast, Allgather and arithmetic collectives favor ACCL. In reduce and allreduce particularly, F2F ACCL performs better than H2H OpenMPI on message sizes over 256 bytes. This is expected given that the streaming arithmetic embedded in the ACCL FPGA kernel minimizes the latency of data manipulation for reductions and can perform



(a) Broadcast (b) Scatter (c) Gather



(d) Allgather (e) Reduce (f) Allreduce

Fig. 7: ACCL Collective Performance

parallel reduction on a wide 512-bit data path. However, for reduce with messages larger than 8MB, the CCLO is not able to keep pace with the POE which causes TCP packet loss and retransmission, leading to very high latency which we have not plotted here.

3) Scalability: Figure 8 evaluates the scalability of ACCL compared to OpenMPI when executing allreduce from 3 to 8 ranks (message size 8MB), as well as the range of execution times encountered in the 100 runs of each configuration. Given the ring algorithm employed in ACCL, we expect very little increase in runtime with increase in scale, which is what we observe. Since ACCL utlizes dedicated logic in FPGA to schedule the collectives, we observe very little execution jitter compared to OpenMPI which executes on the host CPUs. Overall runtime variability is greater for OpenMPI compared with ACCL, both between runs at different scales, and between different runs at the same scale. We believe the scale-related variability is caused by OpenMPI selecting different implementations for the allreduce at different scales.

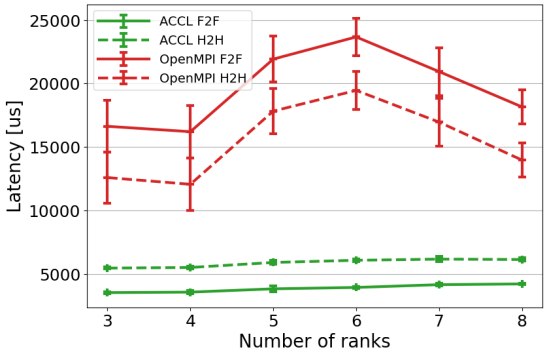


Fig. 8: Allreduce latency versus world size (8MB messages).

VII. Conclusion and Future Work

We have presented ACCL, a collective communication library executing on an FPGA and utilizing the FPGA IO to implement direct Ethernet connectivity. ACCL can be utilized to connect applications executing on FPGA with less latency than traditional approaches which require FPGA-produced data to be moved to the host before inter-node communication. Our evaluation demonstrates that, at least for message sizes larger than 1 MByte, ACCL outperforms OpenMPI and can therefore potentially improve the scalability of FPGA applications. We also observed much less execution jitter, making applications more predictable. Despite these results, ACCL still does not saturate the 100 Gbps TCP-IP link, so further performance improvement through control plane tuning is possible. Fortunately, ACCL is extremely flexible, consisting of a configurable dataplane and a programmable control plane exposed to the user through Python and C++ APIs. This makes optimisation relatively easy, and we believe ACCL will be a useful tool for future research in distributed FPGA computation. Future research will focus on applying ACCL to FPGA-accelerated applications such HPCG_FPGA [40], [41] and utilizing the ACCL streaming kernel and external arithmetic attachments to construct application-specific compressed collectives which we will apply to optimize communication in neural network training.

REFERENCES

[1] *RCCL's documentation*, AMD, 2021. [Online]. Available: https://rccl.readthedocs.io/en/rocm-4.3.0/

[2] *NVIDIA Collective Communications Library (NCCL)*, NVDIA, 2021. [Online]. Available: https://docs.nvidia.com/deeplearning/nccl/index.html

[3] A. Putnam, A. M. Caulfield *et al.*, "A reconfigurable fabric for accelerating large-scale datacenter services," in *ISCA'14*.

[4] A. Caulfield, E. Chung *et al.*, "A cloud-scale acceleration architecture," in *MICRO'16*.

[5] "Aqua (advanced query accelerator) for amazon redshift." [Online]. Available: https://aws.amazon.com/redshift/features/aqua/

[6] Z. He, D. Korolija, and G. Alonso, "Easynet: 100 gbps network for hls," 2021, international Conference on Field-Programmable Logic and Applications (FPL 2021); Conference Location: online; Conference Date: August 30 – September 3, 2021.

[7] T. De Matteis, J. de Fine Licht *et al.*, "Streaming message interface: High-performance distributed memory programming on reconfigurable hardware," in *SC '19*.

[8] M. Saldana and P. Chow, "Tmd-mpi: An mpi implementation for multiple processors across multiple fpgas," in *2006 International Conference on Field Programmable Logic and Applications*, 2006, pp. 1–6.

[9] N. Tarafdar, N. Eskandari, V. Sharma, C. Lo, and P. Chow, "Galapagos: A full stack approach to fpga integration in the cloud," *IEEE Micro*, vol. 38, no. 6, pp. 18–24, 2018.

[10] B. Ringlein, F. Abel *et al.*, "Zrlmpi: A unified programming model for reconfigurable heterogeneous computing clusters," in *FCCM'20*.

[11] O. Mencer, K. H. Tsoi *et al.*, "Cube: A 512-fpga cluster," in *SPL'09*.

[12] R. Baxter, S. Booth *et al.*, "Maxwell - a 64 fpga supercomputer," in *AHS'07*.

[13] Accelerated Collective Communication Library(ACCL). [Online]. Available: https://github.com/Xilinx/ACCL

[14] V. Kathail, "Xilinx vitis unified software platform," in *The 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2020, pp. 173–174.

[15] *AMBA 4 AXI4-Stream Protocol Specification*, ARM, 2010. [Online]. Available: https://developer.arm.com/documentation/ihi0051/a/

[16] H. Wang, S. Potluri, D. Bureddy, C. Rosales, and D. K. Panda, "Gpu-aware mpi on rdma-enabled clusters: Design, implementation and evaluation," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 10, pp. 2595–2605, 2014.

[17] C.-T. Yang, C.-L. Huang, and C.-F. Lin, "Hybrid cuda, openmp, and mpi parallel programming on multicore gpu clusters," *Computer Physics Communications*, vol. 182, no. 1, pp. 266–269, 2011.

[18] S. Potluri, H. Wang, D. Bureddy, A. K. Singh, C. Rosales, and D. K. Panda, "Optimizing mpi communication on multi-gpu systems using cuda inter-process communication," in *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*. IEEE, 2012, pp. 1848–1857.

[19] E. Chung, J. Fowers *et al.*, "Serving dnns in real time at datacenter scale with project brainwave," in *IEEE MICRO'18*.

[29] D. Sidler, Z. Wang *et al.*, "Strom: Smart remote memory," in *EuroSys '20*.

[20] C. Alvarez, Z. He *et al.*, "Specializing the network for scatter-gather workloads," in *SOCC'20*.

[21] Z. István, D. Sidler, and G. Alonso, "Caribou: Intelligent distributed storage," in *VLDB'17*.

[22] J. Lin, K. Patel *et al.*, "PANIC: A high-performance programmable NIC for multi-tenant networks," in *OSDI'20*.

[23] Xup vitis network example (vnx). [Online]. Available: https://github.com/Xilinx/xup_vitis_network_example

[24] D. Sidler, G. Alonso, M. Blott, K. Karras *et al.*, "Scalable 10Gbps TCP/IP Stack Architecture for Reconfigurable Hardware," in *FCCM'15*, 2015.

[25] D. Sidler, M. Chiosa, Z. He, M. Ruiz, K. Karras, and L. Liu. Scalable network stack supporting tcp/ip, rocev2, udp/ip at 10-100gbit/s. [Online]. Available: https://github.com/fpgasystems/fpga-network-stack

[26] D. Sidler, Z. Istvan, and G. Alonso, "Low-Latency TCP/IP Stack for Data Center Applications," in *FPL'16*, 2016.

[27] M. Ruiz, D. Sidler *et al.*, "Limago: An fpga-based open-source 100 gbe TCP/IP stack," in *FPL'19*.

[28] Li Ding, Ping Kang *et al.*, "Hardware tcp offload engine based on 10-gbps ethernet for low-latency network communication," in *FPT'16*.

[30] B. Li, Z. Ruan *et al.*, "Kv-direct: High-performance in-memory key-value store with programmable nic," in *SOSP'17*.

[31] N. Eskandari, N. Tarafdar *et al.*, "A modular heterogeneous stack for deploying fpgas and cpus in the data center," ser. FPGA'19.

[32] B. Ringlein, F. Abel, A. Ditter, B. Weiss, C. Hagleitner, and D. Fey, "Programming reconfigurable heterogeneous computing clusters using mpi with transpilation," in *H2RC'20*.

[33] Xilinx ultrascale+ integrated 100g ethernet subsystem. [Online]. Available: https://www.xilinx.com/products/intellectual-property/cmac_usplus.html

[34] *AMBA AXI and ACE Protocol Specification*, ARM, 2021. [Online]. Available: https://developer.arm.com/documentation/ihi0022/hc

[35] *Quick Start Guide:MicroBlaze Soft Processor for Vitis 2019.2*, Xilinx, 2019. [Online]. Available: https://www.xilinx.com/support/documentation/quick_start/microblaze-quick-start-guide-with-vitis.pdf

[36] (2021) Performance and resource utilization for microblaze v11.0 vivado design suite release 2021.1. Xilinx. [Online]. Available: https://www.xilinx.com/html_docs/ip_docs/pru_files/microblaze.html

[37] H. Zhao and J. Canny, "Sparse allreduce: Efficient scalable communication for power-law data," *arXiv preprint arXiv:1312.3020*, 2013.

[38] *UG1120, Alveo Data Center Accelerator Card Platforms*, Xilinx, 2021. [Online]. Available: https://www.xilinx.com/support/documentation/boards_and_kits/accelerator-cards/ug1120-alveo-platforms.pdf

[39] Xilinx, "Xilinx adaptive compute cluster (XACC) program," https://www.xilinx.com/support/university/XUP-XACC.html, accessed: 2021-01-07.

[40] High performance conjugate gradient benchmark - fpga. [Online]. Available: https://github.com/Xilinx/HPCG_FPGA

[41] A. Zeni, K. O'Brien, M. Blott, and M. D.Santambrogio, "Optimized implementation of the hpcg benchmark on reconfigurable hardware," in *European Conference on Parallel Processing*, 2021.