EasyNet: 100 Gbps Network for HLS

EasyNet: 100 Gbps Network for HLS

Zhenhao He, Dario Korolija, Gustavo Alonso
Systems Group, Department of Computer Science, ETH Zurich, Switzerland
firstname.lastname@inf.ethz.ch

Abstract—**The massive deployment of FPGAs in data centers is opening up new opportunities for accelerating distributed applications. However, developing a distributed FPGA application remains difficult for two reasons. First, commonly available development frameworks (e.g., Xilinx Vitis) lack explicit support for networking. Developers are, thus, forced to build their own infrastructure to handle the data movement between the host, the FPGA, and the network. Second, distributed applications are made even more complex by using low level interfaces to access the network and process packets. Ideally, one needs to combine high performance with a simple interface for both point-to-point and collective operations. To overcome these inefficiencies and enable further research in networking and distributed application on FPGAs, we first show how to integrate an open-source 100 Gbps TCP/IP stack into a state-of-the-art FPGA development framework (Xilinx Vitis) without degrading its performance. Further, we provide a set of MPI-like communication primitives for both point-to-point and collective operations as a High Level Synthesis (HLS) library. Our point-to-point primitives saturate a 100 Gbps link and our collective primitives achieve low latency. With our approach, developers can write hardware kernels in high level languages with the network abstracted away behind standard interfaces. To evaluate the ease of use and performance in a real application, we distribute a K-Means algorithm with the new stack and achieve a 1.9X and 3.5X throughput increase with 2 FPGAs and 4 FPGAs respectively.**

I. Introduction

The network plays a fundamental role in data centers. Accordingly, cloud deployments have started to treat FPGAs as first-class processing components with direct network access such as in the Microsoft's Catapult [1], [2] or Amazon's AQUA [3]. As Microsoft's BrainWave [4], [5] and KV-Direct [6] projects indicate, support for data center networking is a key step to go beyond the FPGA as a co-processor.

FPGAs are increasing their programmability by providing high-level synthesis (HLS) languages to enable software developers trained in conventional programming languages such as C++ to also use FPGAs. The most common vendor development frameworks, Xilinx Vitis [7] and Intel Quartus [8], support HLS and follow a clear trend towards higher abstractions. For instance, they all now have a far simpler way to manage data movement between host, FPGA memory, and application through the use of standard host APIs and hiding the underlying software-hardware interaction. When these two trends are put together, one area emerges where there are glaring gaps in terms of support: networking. While there is a growing number of research efforts targeting data center networking protocols for FPGAs [9]–[12], tool support for data center networking is still lacking. Current development frameworks do provide basic networking functionality through low-level link and physical layer protocols [13]–[15]. Such support is not sufficient in a data center where reliability, automatic connection, and a large number of connections are needed. Only recently, support for UDP in Vitis [16] has become available. Proprietary commercial implementations of network stacks [17]–[19] exist but they are often not available to researchers, are limited in their functionality or performance (only 10 or 40 Gbps), and are not integrated with HLS. Moreover, the presence of a data center network stack is often not enough. On the CPU side, almost all distributed application development relies on high-level APIs such as OpenMP [20] or MPI [21] for communication and even more complex systems for distributed coordination, such as ZooKeeper [22]. All these platforms are widely available and open-source, greatly facilitating the creation of new distributed applications. For FPGAs to become competitive, they must provide a similar infrastructure and be integrated with such systems. Recent work has tried to provide high-level abstractions for different communication patterns [23]–[25]. However, they either target a proprietary protocol for fixed typologies or provide a minimal subset of operations, thus lacking both generality and tool support.

To address these issues, we propose EasyNet, a system aiming to reduce the programming effort for distributed FPGA applications. As a first step, we integrate a 100 Gbps open-source TCP/IP stack [10], [26] into Vitis [7], to enable HLS network programming. This integration faces two major challenges: (1) The integration should be seamlessly compatible with the original design flow of the Vitis application, meaning that the instantiation of the network stack has to be hidden from the application developer; (2) The performance of the 100 Gbps network stack should not be affected although Vitis is optimized for bulk data transfer with aligned memory address [7], while network communication often doesn't have this memory access pattern. Besides, to raise the level of abstraction further, we develop a rich set of MPI-like communication primitives for point-to-point and collective operations. The primitives hide the interaction and control management within the network layer and they can be easily invoked from an HLS C library. We show that our primitives running on FPGA clusters can easily saturate a 100 Gbps network and achieve lower latency than software MPI running on a CPU cluster. To evaluate EasyNet, we present a case study based on distributing K-Means on an FPGA cluster. Compared to a single node implementation, we show a negligible overhead from communication and achieve 1.9X and 3.5X by using 2 and 4 FPGAs, respectively. EasyNet is open-source [1].

[1]https://github.com/fpgasystems/Vitis_with_100Gbps_TCP-IP

II. Background

A. FPGA Development Framework

Vitis [7] is one of the state-of-the-art FPGA development frameworks. In Vitis, the FPGA is divided into two regions. The static region (shell), which contains common infrastructure, such as the DMA engine. The dynamic region contains the customized logic for user-defined kernels. Vitis abstracts data movement by automatically adding the necessary interconnects for these kernel interfaces to communicate with the rest of the platform. Combined with HLS, Vitis is a huge boost in productivity compared to older platforms such as Vivado [27] or SDAccel [28]. However, Vitis imposes constraints on kernel interfaces, and interconnects to the network are not supported, limiting its use in distributed applications. Thus, current development of distributed applications requires to build a customized infrastructure to handle data movement between host, FPGA memory and network using low level interfaces, which lacks support of HLS and portability.

B. Distributed Communication Primitives

Point-to-point and collective communication operations, such as broadcast and reduce, are essential in distributed applications. Distributed application development in software relies on platforms that provide such primitives behind standard interfaces such that developers handle communication only through high-level interfaces. MPI [21] is an example of such a platform providing both point-to-point and collective primitives, automatically maps the communication to the best network protocol, and, in some cases, even changes the algorithm used depending on the system size and workload. This is important as, for instance, *reduce* can be implemented either as a client-server for medium-size systems and messages, through a broadcast tree [29] for larger systems, or with a tile-based algorithm [30] to handle large message sizes.

III. Related Work

In the past, FPGAs were typically connected through point-to-point serial links with fixed topologies and with lightweight or proprietary protocols [1], [31]–[34]. Many of these implementations build on top of low-level networking IP cores provided by Xilinx and Intel, such as the Ethernet Media Access Controller (MAC) [13], [14], [35].

The situation has changed since FPGAs are deployed in the data center, where they are directly connected with data center infrastructures (e.g., high bandwidth links and network switches) [1], [36]. Therefore, there is now a trend to build infrastructure for high-performance FPGA-based NICs [37]–[39] and to develop complete network stacks on FPGAs for protocols commonly used in data centers, such as UDP [16], TCP [9], [10], [12], [17], [26], [40] or RDMA [6], [11]. As pointed out above, networking itself is not enough. Accordingly, there is also a growing interest in implementing higher-level abstractions for communication. For instance, IBM's CloudFPGA [41], [42] proposes to convert software executable MPI code to hardware synthesizable code that runs over hardware UDP stack. Eskandari et al. [23] proposed a system integrated with SDAccel [28], an older generation of the framework, that maps HLS kernels to different FPGAs by incorporating an open-source 10 Gbps TCP/IP stack [10] and providing MPI-like point-to-point primitives. There are also efforts aiming at providing MPI-like communication abstractions targeting physical/link layer protocols and fixed topologies [24], [25], [43]. Recently VNx [16] has become available, which extends Vitis with UDP, allowing the developer to interact with the UDP kernel using HLS streaming interfaces. Our goal in this paper is to seamlessly integrate TCP/IP in Vitis, a reliable protocol far more complex than UDP, and provide a rich set of communication primitives containing both point-to-point and collective operations.

A wide range of use cases can benefit from EasyNet. For instance, Zhang et al. [44] propose to use a pipelined ring-based FPGA cluster to accelerate convolutional neural networks. Owaida et al. [45] partition inference over decision tree ensembles on an FPGA cluster. In both cases, these efforts target a pre-defined cluster topology with point-to-point serial links for lack of better infrastructure. More recent work has explored DNN inference using FPGA clusters [46]–[48], offloading an SDN stack to an FPGA-based smart NIC [49], managing the scatter and gather problem in parallel data processing with FPGAs [50], or implemented key-value stores [6], [51]. Through EasyNet, such applications will become easier to develop by providing access from HLS to a 100 Gbs TCP/IP stack that is usable through a set of MPI-like interfaces without loss of performance or flexibility.

IV. Design Overview

EasyNet comprises two parts. First, it incorporates a 100 Gbps TCP/IP stack into a state-of-the-art FPGA development framework: Vitis (Section V). The seamless integration of the network stack into Vitis is challenging because Vitis limits how user kernels interface and interact with the rest of the platform. For instance, network pin assignment is not supported by Vitis. Further, memory access in Vitis is highly optimized for bulk data transfer while packet buffering in a network stack could have irregular memory access since packets with varying sizes don't come in order. Regardless of these challenges, the goal is for EasyNet to meet the following three constraints regarding the network stack. **C1:** The network infrastructure should be generic to a variety of applications rather than optimized for a concrete one. **C2:** The network infrastructure should be abstracted away from the application developer and usable from HLS. **C3:** The integration into Vitis should not reduce the performance of the network stack.

Second, EasyNet provides various communication primitives that hide the network layer and can be easily instantiated in an HLS design (Section VI). The goal is for such EasyNet primitives to satisfy the following constraints. **C4:** Each communication primitive should keep a low resource footprint while achieving high throughput or maintain low latency. **C5:** The primitives should be encapsulated in an HLS library callable as a function.

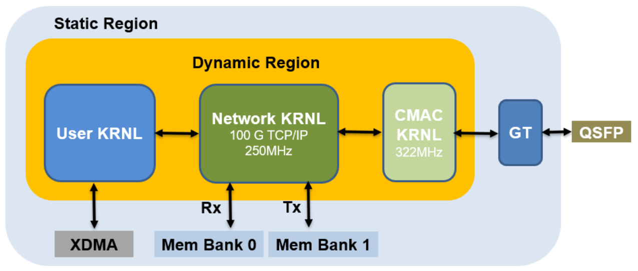


Fig. 1: Overall infrastructure architecture.

V. Network Infrastructure

A. Overall Infrastructure Architecture

The design of EasyNet aims to provide network functionality as a common infrastructure to different applications (**C1**). Therefore, we adopt a modular design principle, where different functionalities are separated into different kernels. The overall design is divided into three kernels (Figure 1).

CMAC Kernel. The CMAC kernel contains an IP block for the 100G Ethernet Subsystem, which needs to be configured for each board. A separate CMAC kernel increases the portability among different FPGA boards. This kernel bridges the whole infrastructure to the GT pins, which are the I/O pins towards the QSFP network interfaces. It also exposes two 512-bit AXI4-Stream interfaces to the network kernel for transmitting (Tx) and receiving (Rx) network packets.

Network Kernel. The network kernel contains IP blocks of an open-source 100 Gbps TCP/IP stack [52] supporting thousands of TCP/IP connections, window scaling and out-of-order packet processing. It is clocked at 250 MHz to saturate the network bandwidth. The kernel contains two 512-bit memory-mapped streaming interfaces to two memory banks, which serve as temporary buffers for re-transmission of Tx data and buffering of Rx data respectively.

User Kernel. The user kernel contains streaming interfaces to the network kernel and other interfaces that can be customized for each application. The user kernel can be written in various languages: RTL, C/C++, and OpenCL. The interconnects to the memory and the network are hidden from the user.

B. User-Network Streaming Interfaces

Interacting with the network kernel involves manipulating several streaming interfaces. Figure 2 shows the interfaces on the Tx path. The user kernel can open active connections through the *openConReq* interface providing the destination's IP address and port. Through the *openConRsp*, the user kernel will be notified whether the connection has been established and receive the session ID of the new connection. Data transmission through an established connection requires a control handshake before the transmission of the payload. The user kernel has to first provide the session ID and the length of the data to the *txDataReq* interface and then the TCP module will return a response on the *txDataRsp* indicating potential errors and the remaining buffer space for that connection. If the txDataRsp doesn't return any error, the user kernel can send the payload to the *txData* interface. Therefore, to saturate the

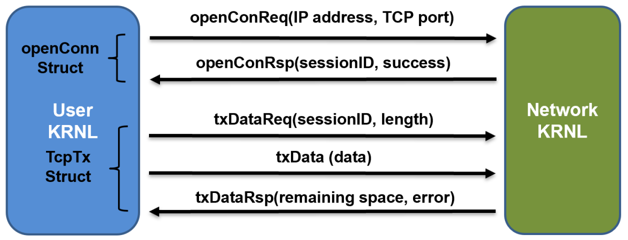

openConn Struct
User KRNL
TcpTx Struct
openConReq(IP address, TCP port)
openConRsp(sessionID, success)
txDataReq(sessionID, length)
txData (data)
txDataRsp(remaining space, error)
Network KRNL

Fig. 2: Streaming interfaces between user and network kernel.

network rate, the control handshake of the next packet needs to be overlapped with the data transmission of the current packet, imposing more difficulties on the application developer.

C. Integration into Vitis

Network Instantiation Automation. To satisfy **C2**, the major challenge is to make the connection from the CMAC kernel to the network pins without the involvement of the application developer. With Vitis, each kernel is synthesized to a Xilinx object and the interconnects for each kernel interface are automated according to a configuration file. However, connecting a kernel interface to the network pins is not allowed. Thus, we have to explicitly guide the back-end routing and placement tool [27] to make the routing connection from the CMAC kernel to the GT pins [2]. To avoid user intervention, we add an extra step in the Vitis synthesis and compilation workflow by providing a post-synthesis TCL file that automatically makes the routing connection from the CMAC kernel to the GT pins.

Maintaining Performance. To keep the performance of the original 100 Gbps implementation (**C3**), the major hurdle is the interaction between the network kernel and the memory banks. In a TCP/IP stack, the payloads are temporarily stored in a memory bank for re-transmission or buffering purposes. This requires a memory bandwidth of at least 100 Gbps if the goal is to saturate the 100 Gbps network link. However, Vitis is optimized for 64-byte aligned, sequential memory access. Unaligned memory access significantly decreases the memory bandwidth because it will trigger several aligned memory accesses. For each TCP/IP connection, an initial memory address is assigned and upcoming packets are stored with an offset from the initial memory address. First, the initial memory address is determined by the initial sequence number of the connection, which is a random number, making it most likely not aligned to 64 bytes. Second, in a default TCP/IP setting, the maximum segmentation size (MSS) is 1460 bytes, which is not multiple of 64 bytes, and network devices tend to pack small messages to match MSS to maximize network utilization. This leads to a throughput drop due to the unaligned access even if packets arrive in order and sequentially access the memory. To overcome these two inefficiencies, we first use a relative initial sequence number that results in a 64-byte aligned initial memory address and then tune the MSS of the hardware TCP/IP stack to 1408 bytes, which is the maximum multiple of 64 bytes lower than the default.

[2]We choose Vitis shell XDMA 201920.3 since only the latest shell exposes the GT pins to the dynamic region

With these changes and adaptations, EasyNet encapsulates the 100 Gbs stack behind a relatively simple interface that is usable from HLS and without the developer having to deal with the details of the network.

VI. Communication Primitives

The interaction with the TCP/IP stack is still complex due to low-level streaming interfaces and control handshakes, as previously explained in subsection V-B. To provide an even higher level of abstraction, EasyNet also includes various communication primitives that can be called as a function from an HLS C library (**C5**). These primitives have an MPI-like interface and semantics with the underlying implementation being optimized to minimize resource utilization (**C4**).

A. Connection Establishment

The *open connection* primitive takes a list of destination IP and port pairs as input arguments and issues a large amount of open connection request to the TCP/IP stack in a pipelined fashion. At the same time, it processes open connection status from the TCP/IP stack. This is useful in distributed applications to quickly connect to several nodes. Once a connection is established, the function writes the session ID, used to distinguish among different connections, into the session table. A boolean value of true is returned once all the connections are established. Similarly, the *listen port* primitive open ports according to a list of port numbers provided by the caller and returns true once all ports are successfully opened.

```
bool openConnection(int numCon, uint32_t*IP, int*port,
    ssStruct &session, opnConStruct&opnCon);
bool listenPort (int numCon, int*port, ssStruct &session,
    lstnPortStruct &lstnPort);
```

B. Point-to-point Primitives

The *send* and *receive* primitives are point-to-point operations that involve one connection between two processes. These primitives send/receive data with connection specified via session ID and support operating with either a data stream or a memory pointer. They support variable data widths while to saturate network rate requires a data width of 512 bits. One important feature is that these primitives can be used with other data processing functions in a data flow region, such that the network communication and the computation can be pipelined and overlapped for higher efficiency.

```
void send(type*data, uint64_t byte, ssStruct session,
    TcpTxStruct&TcpTx);
void recv(type*data, uint64_t byte, ssStruct session,
    TcpRxStruct&TcpRx);
```

C. Collective Primitives

Implementing collective primitives requires manipulation of several connections on each node at the same time and time-sharing of the TCP/IP stack. EasyNet implements several collective primitives: *scatter*, *broadcast*, *gather*, *reduce* and *all-reduce*. Due to space limitations, we will focus on broadcast, reduce, and all-reduce. In EasyNet, we focus on optimizing latency for medium size messages. Therefore, we adopt an all-to-one (client-to-server) implementation for reduce and a one-to-all (server-to-client) implementation for broadcast. Our implementation can be used as a basic building block for other collective algorithms, such as a tree-based algorithm [30].

To set up connections for collective operations, the server node actively establishes all the connections to the clients using the open connection primitive. Once all connections are established, both the server and the client nodes can perform duplex data transmission.

1) Broadcast: One way to broadcast data is to use the send primitive multiple times over the target data in sequence to each client node. However, this will starve the later connections until sending through the previous connection completes. To avoid this, we provide a simple yet effective broadcast primitive by interleaving the sending of small amounts of data to each client. A list of session IDs and the total number of bytes to be sent are provided when invoking the broadcast primitive. Internally, the primitive repeatedly reads a small portion of the data and stores it in a small yet fast on-chip temporary buffer. It iterates over the temporary buffer and sends data to different connections in a round-robin fashion. The primitive then reads the next data chunk once the previous chunk of data is sent to all connections. On the client-side, the application simply uses the receive primitive.

```
void broadcast(type*data, uint64_t byte, ssStructs
    sessionID, TcpTxStruct&TcpTx);
```

2) Reduce: In our implementation, all the client nodes send data to one server node for reduction. Figure 3 shows the architecture of the reduce primitive on the server node. It mainly contains two parts: (1) A gather module that collects data chunks from several connections in a round-robin manner. (2) A reducer that performs parallel reduction at line rate.

One challenge of gathering data is to preserve connection order for receiving and storing data. We deploy a finite state machine (FSM) that processes packet notifications from the TCP/IP stack and keeps track of the available data buffered in the TCP/IP stack for each connection. Then the FSM issues read request of a fixed chunk of data (1 KB) to the TCP/IP stack for each connection in a round-robin fashion. In this way, we preserve the order of the received data at the application level in the granularity of the chunk size. If the connection in turn doesn't have enough data for a read, the FSM will wait until new data for that connection arrives and we rely on the TCP/IP stack to perform flow control of the unbalanced sending rate between connections. The gathered data is multiplexed to temporary FIFOs (4KB) according to port number. The total number of FIFOs (MAX_SS in the listing) is a trade-off between the maximum amount of connections and the total resource utilization.

The reducer operates on data stored in the FIFOs in a round-robin fashion. It is equipped with arithmetic units that can perform parallel reduction at network line rate on specified data granularity (WIDTH in the listing). However, if the FIFO

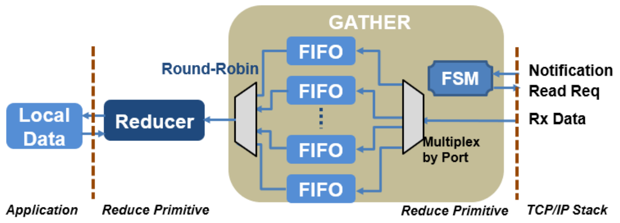


Fig. 3: Hardware architecture of reduce primitive

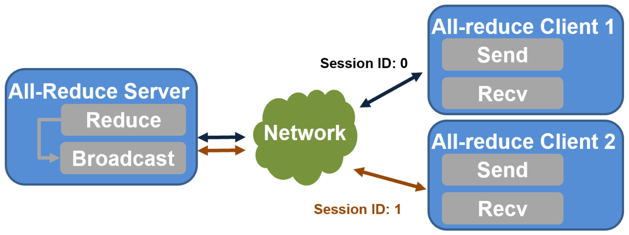


Fig. 4: Block diagram of running all-reduce on three nodes

is empty due to unbalanced connections, the reducer waits until the data arrives. Once the reduced has iterated over all the FIFOs once, it stores the result back in the data storage.

```
template<int MAX_SS, int WIDTH>
void reduce_sum(type*data, uint64_t byte, ssStruct
        session, TcpRxStruct&TcpRx);
```

3) All-Reduce: For the *all-reduce* implementation, we adopt a classical approach where an all-to-one reduce is followed by a one-to-all broadcast. In this setting, we could use previous primitives to build the all-reduce function. On the server node, the all-reduce implementation can be realized with a reduce and a broadcast primitive, while on the client-side, the operation can be realized with a send primitive followed by a receive primitive, as shown in Figure 4.

```
template<int MAX_SS, int WIDTH>
void all_reduce_sum(type*data, uint64_t byte, ssStruct
        session, TcpRxStruct&TcpRx, TcpTxStruct&TcpTx);
```

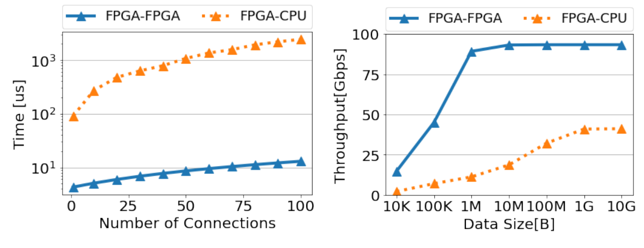
VII. Evaluation

A. Experimental Setup

We run the experiments on a cluster containing two nodes. Each nodes has a CPU with 4 Intel Xeon Gold 6234 processors and 376 GB of memory, and 2 U280 FPGAs. All the CPUs and FPGAs are connected through a 100 Gbps Cisco Nexus 9336C-FX2 network switch. As a comparison, we also run equivalent primitives with OpenMPI 3.1.4 on a cluster containing Intel Xeon E5–2609 2.40 GHz processors with 128 GB RAM and a Mellanox QDR HCA 100 Gbs NIC. OpenMPI uses TCP/IP as the underlying communication protocol. We run each experiment 5 times and report the average.

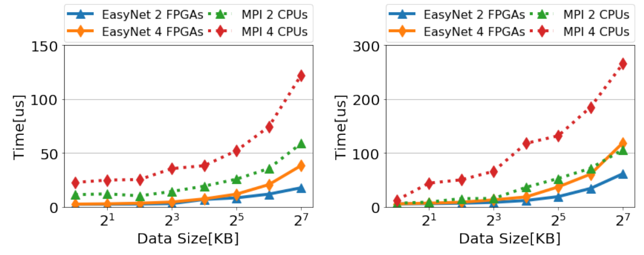
B. Micro Benchmarks

1) Latency: We measure the latency to establish a varying number of connections between two FPGAs using the open connection primitive, as shown in Figure 5a. We compare with the latency between an FPGA client and a CPU server, where the server processes are mapped to different cores. The latency



(a) Connection establishment latency (b) Point-to-point throughput

Fig. 5: Latency and throughput measurement.



(a) Broadcast (b) All-reduce

Fig. 6: Broadcast and all-reduce primitive comparison between EasyNet and MPI running with various data size

between two FPGAs is more than one order of magnitude lower in all cases, showing the advantage of a hardware network stack and the efficiency of the primitives. Besides, the round trip time(RTT) measured as a ping-pong benchmark of 64-byte data between two FPGAs is 4.3 us while the RTT between two CPUs is 56 us.

2) Throughput: We compare the throughput between 2 FPGAs and between an FPGA and a CPU using the send and receive primitives. In the latter case, we implement a receiver application on the CPU using TCP sockets, which is mapped to a single core since point-to-point operations involve single connection. As shown in Figure 5b, the throughput between 2 FPGAs approaches 100 Gbps with a small data size(1 MB), showing that both our primitives work at network rate. In contrast, with a CPU receiver, it can only achieve half of the bandwidth and it takes a larger amount of data to reach peak performance due to the packet processing overhead in the software network stack. To saturate line rate on the CPU, it requires opening more connections and larger payload size.

3) Collective Operations: We benchmark the broadcast and all-reduce primitives using 4 FPGAs and we compare them to the corresponding communication primitives in MPI, as shown in Figure 6. The evaluation is done with 32-bit fixed-point data and with SUM as the reduce operation. We observe that EasyNet achieves lower latency than its MPI counterparts running on CPUs. Further, when the data size is small, the latency is dominated by the network time. As we increase the data size, we see an almost linear increase for both primitives although the increase is more marked on the CPU case.

C. Application: K-Means

K-Means is a popular machine learning algorithm and is a common use case for hardware acceleration [53]–[62]. We examine the ease of use of EasyNet by distributing the computation of K-Means across multiple FPGAs. The algorithm mainly contains two steps in each iteration: sample assignment and center update. First, each sample is assigned to its closest center by calculating the squared Euclidean distance. For each cluster, a sum vector is used to accumulate the sample assigned to it and an assignment counter is used to count the number of assignments. Second, new centers of each cluster are calculated as the mean of the samples assigned by performing a division of the sum vector over the assignment counter.

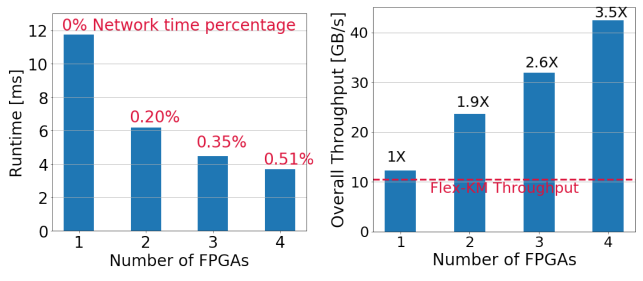
```
1  static ap_uint<32> center [DIM_MAX] [CNTR_MAX];
       //On−chip memory for center
2  static ap_uint<64> sum_n_cnt [DIM_MAX ∗
       CNTR_MAX + CNTR_MAX]; // On−chip memory for
       sum and assignment counters
3  openConnection( numCon, IP, port, &session, opnCon);
4  static ap_uint<32> byte = (DIM_MAX ∗ CNTR_MAX +
       CNTR_MAX) ∗ 8; //Set all−reduce data amount
5  read_sample_n_assign_dataflow(sample, center, sum_n_cnt,
       numSample, numCluster, numDim); //Data flow region
       to read sample and assign sample to cluster
6  all_reduce_sum<MAX_SS, WIDTH>(sum_n_cnt, byte,
       session, TcpRx, TcpTx); //Perform all−reduce
7  upd_center(sum_n_cnt, center, numCluster, numDim);
       //Update center according to aggregated sum and count
8  write_back_result(center, numCluster, numDim, result);
```

Listing 1: Code snippet of EasyNet-KM server node.

1) Single Node Baseline: We employ an existing highly-pipelined FPGA K-Means implementation: Flex-KM [63]. For the sample assignment, the design contains 16 parallel pipelines, each containing a systolic array of 16 distance processors that can process one dimension of a sample per cycle. For the center update, it contains a collector that aggregates partial sum vectors and assignment counters from each pipeline and a divider for division. For a comparison of the programming effort between HDL and HLS, we re-implement the original Flex-KM HDL design with Vitis HLS.

2) Distributed Implementation: Distributing K-Means with EasyNet (EasyNet-KM) is simple. Each compute node completes the sample assignment step on its local partition of samples, producing a partial result of sum vectors and assignment counters. An extra aggregation step is needed to aggregate all partial sum vectors and assignment counters from all nodes. Listing 1 shows the code snippet of the server node of the EasyNet-KM. Notice that only two extra functions are required (line 3 to open connections and line 6 to perform all-reduce over the network) compared to a single node implementation.

3) Experiments: We use the Forest data set [64] (581014 samples, 54 dimensions) with the number of clusters set to 7. The data set is partitioned equally across different nodes. Figure 7a and figure 7b show the runtime per iteration and the aggregated throughput of our EasyNet-KM implementation running with 1, 2 and 4 nodes. First, we observe that the network communication time is negligible compared to the computation time, proving the advantages of the low latency all-reduce primitive. Additionally, the single node throughput of our EasyNet-KM design is on a par with Flex-KM and we could only achieve a sub-linear increase of overall throughput



(a) Runtime per iteration (b) Overall throughput

Fig. 7: Runtime per iteration and overall throughput of EasyNet-KM with Forest data set

due to the serial part of the computation, such as collecting partial results from each pipeline and center update.

Table I shows lines of code for the Flex-KM in HDL and the EasyNet-KM in HLS. The advantage of programming with Vitis HLS over HDL is obvious and by adding few lines of code, we easily turned a single node application into a distributed version.

TABLE I: Lines of code.

Flex-KM(HDL)	EasyNet-KM(HLS)		
Total	Computation	Network	Total
4463	396	48	444

D. Resource Consumption

Table II shows the resource consumption of the design. The CMAC and the network kernel occupy only less than 10% LUT and BRAM. The communication primitives consume a minimal amount of resources. The K-Means kernel has low DSP usage since the Flex-KM was designed to run on an old platform (HARP2 [65]) and we follow the same configuration for an apple-to-apple comparison of the programming effort. The single-node performance of EasyNet-KM could be further optimized with more resources but we decided to focus instead on evaluating the ease of use of EasyNet.

TABLE II: Resource consumption

Resources	**LUT**	**BRAM**	**DSPs**
CMAC	15,717 (1.21%)	18 (2.24%)	0 (0%)
Network	114,699 (8.80%)	417 (7.09%)	0 (0%)
Send	2,326 (0.22%)	4 (0.29%)	0 (0%)
All-reduce	8,312 (0.81%)	83 (6.22%)	9 (0.10%)
K-Means	166,821 (14.03%)	486 (28.44%)	329 (3.65%)

VIII. Conclusion

Aiming to provide tool support and facilitate the development of distributed applications with FPGAs, EasyNet integrates a 100 Gbps network stack into a state-of-the-art FPGA development framework and provides a rich set of high-performance communication primitives using HLS. We show that with EasyNet, an application can be easily partitioned across an FPGA cluster by changing a few lines of code and achieving a performance boost with minimal communication overhead.

Acknowledgment

We would like to thank Xilinx for their generous donation of the XACC FPGA cluster at ETH Zurich on which the experiments were conducted.

REFERENCES

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

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

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

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

[5] J. Fowers, K. Ovtcharov *et al.*, "A configurable cloud-scale dnn processor for real-time ai," in *ISCA'18*.

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

[7] "Vitis application acceleration development flow documentation." [Online]. Available: https://www.xilinx.com/html_docs/xilinx2020_2/vitis_doc/kme1569523964461.html

[8] "Intel quartus prime standard edition user guide: Getting started." [Online]. Available: https://www.intel.com/content/www/us/en/programmable/documentation/yoq1529444104707.html

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

[10] D. Sidler, G. Alonso *et al.*, "Scalable 10gbps tcp/ip stack architecture for reconfigurable hardware," in *FCCM'15*.

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

[12] Y. Ji and Q. Hu, "40gbps multi-connection tcp/ip offload engine," in *WCSP'11*.

[13] "Low latency 100g ethernet intel fpga ip core user guide." [Online]. Available: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/ug/ug20085.pdf

[14] "Ultrascale+ devices integrated 100g ethernet subsystem v3.1." [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/cmac_usplus/v3_1/pg203-cmac-usplus.pdf

[15] "Aurora 64b/66b v11.2." [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/aurora_64b66b/v11_2/pg074-aurora-64b66b.pdf

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

[17] "Enyx premieres 25g tcp and udp offload engines with xilinx virtex ultrascale+ 16nm fpga on bittwares xupp3r pcie board." [Online]. Available: https://www.enyx.com/blog

[18] "Chevin technologies's tcp/ip." [Online]. Available: https://chevintechnology.com/ethernet-ip-2/ct1008-xgtcp/

[19] "Dini group's tcp/ip." [Online]. Available: https://www.synopsys.com/verification/prototyping/dini-products.html

[20] L. Dagum and R. Menon, "Openmp: an industry standard api for shared-memory programming," *IEEE Computational Science and Engineering*, 1998.

[21] W. Gropp, R. Thakur, and E. Lusk, "Using mpi-2: Advanced features of the message passing interface." Cambridge, MA, USA: MIT Press, 1999.

[22] Z. István, D. Sidler *et al.*, "Consensus in a box: Inexpensive coordination in hardware," in *NSDI'16*.

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

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

[25] M. Saldana and P. Chow, "Tmd-mpi: An mpi implementation for multiple processors across multiple fpgas," in *FPL'06*.

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

[27] "Vivado design suite hlx editions." [Online]. Available: https://www.xilinx.com/support/documentation/backgrounders/vivado-hlx.pdf

[28] "Sdaccel environment user guide." [Online]. Available: https://www.xilinx.com/html_docs/xilinx2019_1/sdaccel_doc/itd1534452174535.html

[29] R. Rabenseifner, "Optimization of collective reduction operations," *International Journal of High Performance Computing Applications*, 2005.

[30] P. Sanders, J. Speck, and J. L. Träff, "Two-tree algorithms for full bandwidth broadcast, reduction and scan," in *Elsevier Science Publishers B. V.*, 2009.

[31] S. W. Jun, M. Liu *et al.*, "A transport-layer network for distributed fpga platforms," in *FPL'15*.

[32] T. Bunker and S. Swanson, "Latency-optimized networks for clustering fpgas," in *FCCM'13*.

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

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

[35] N. Tarafdar, T. Lin *et al.*, "Enabling flexible network FPGA clusters in a heterogeneous cloud data center," in *FPGA'17*.

[36] J. Weerasinghe, R. Polig *et al.*, "Network-attached fpgas for data center applications," in *FPT'16*.

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

[38] A. Forencich, A. C. Snoeren *et al.*, "Corundum: An open-source 100-gbps nic," *FCCM'20*.

[39] N. Zilberman, Y. Audzevich *et al.*, "Netfpga sume: Toward 100 gbps as research commodity."

[40] D. Sidler, Z. István, and G. Alonso, "Low-latency TCP/IP stack for data center applications," in *FPL'16*.

[41] B. Ringlein, F. Abel *et al.*, "Programming reconfigurable heterogeneous computing clusters using mpi with transpilation," in *H2RC'20*.

[42] ——, "Zrlmpi: A unified programming model for reconfigurable heterogeneous computing clusters," in *FCCM'20*.

[43] A. Patel, C. A. Madill *et al.*, "A scalable fpga-based multiprocessor," in *FCCM'06*.

[44] C. Zhang, D. Wu *et al.*, "Energy-efficient CNN implementation on a deeply pipelined FPGA cluster," in *ISLPED '16*.

[45] M. Owaida and G. Alonso, "Application partitioning on FPGA clusters: Inference over decision tree ensembles," in *FPL'18*.

[46] J. Fowers, K. Ovtcharov *et al.*, "A configurable cloud-scale dnn processor for real-time ai," in *ISCA'18*.

[47] Y. Zhu, Z. He *et al.*, "Distributed recommendation inference on fpga clusters," *FPL'21*.

[48] W. Jiang, Z. He *et al.*, "Fleetrec: Large-scale recommendation inference on hybrid gpu-fpga clusters," *KDD'21*.

[49] D. Firestone, A. Putnam *et al.*, "Azure accelerated networking: Smartnics in the public cloud," in *NSDO'18)*.

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

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

[52] "Scalable network stack supporting tcp/ip, rocev2, udp/ip at 10-100gbit/s." [Online]. Available: https://github.com/fpgasystems/fpga-network-stack

[53] T. Saegusa and T. Maruyama, "An FPGA Implementation of K-means Clustering for Color Images Based on Kd-tree," in *FPL*, 2006.

[54] X. Wang and M. Leeser, "K-Means Clustering for Multispectral Images Using Floating-point Divide," in *FCCM*, 2007.

[55] M. Gokhale, J. Frigo *et al.*, "Experience with a Hybrid Processor: K-Means Clustering," *The Journal of Supercomputing*, 2003.

[56] M. Estlick, M. Leeser *et al.*, "Algorithmic Transformations in the Implementation of K-means Clustering on Reconfigurable Hardware," in *FPGA*, 2001.

[57] H. M. Hussain, K. Benkrid *et al.*, "Novel Dynamic Partial Reconfiguration Implementation of K-means Clustering on FPGAs: Comparative Results with GPPs and GPUs," *Int. J. Reconfig. Comput.*, 2012.

[58] Z. Wang *et al.*, "Melia: A MapReduce Framework on OpenCL-based FPGAs," *IEEE TPDS*, 2016.

[59] Z. Lin, C. Lo, and P. Chow, "K-means Implementation on FPGA for High-dimensional Data Using Triangle Inequality," in *FPL*, 2012.

[60] Q. Y. Tang and M. A. Khalid, "Acceleration of K-means Algorithm Using Altera SDK for OpenCL," *ACM TRETS*, 2016.

[61] Y. M. Choi and H. K. H. So, "Map-reduce Processing of K-Means Algorithm with FPGA-accelerated Computer Cluster," in *ASAP*, 2014.

[62] Z. He, Z. Wang, and G. Alonso, "Bis-km: Enabling any-precision k-means on fpgas," in *FPGA'20*.

[63] Z. He, D. Sidler, Z. Istvan *et al.*, "A Flexible K-Means Operator for Hybrid Databases," in *FPL'18*.

[64] J. A. Blackard and D. J. Dean, "Comparative Accuracies of Artificial Neural Networks and Discriminant Analysis in Predicting Forest Cover Types From Cartographic Variables," in *Computers and Electronics in Agriculture*, 1999.

[65] N. Oliver, R. Sharma, S. Chang *et al.*, "A Reconfigurable Computing System Based on a Cache-Coherent Fabric," in *ReConFig'11*.