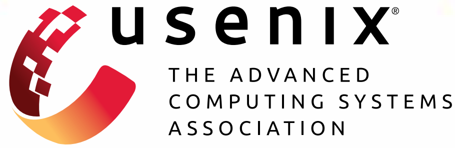


ACCL+: an FPGA-Based Collective Engine for Distributed Applications

Zhenhao He, Dario Korolija, Yu Zhu, and Benjamin Ramhorst, *Systems Group, ETH Zurich;* Tristan Laan, *University of Amsterdam;* Lucian Petrica and Michaela Blott, *AMD Research;* Gustavo Alonso, *Systems Group, ETH Zurich*

https://www.usenix.org/conference/osdi24/presentation/he

This paper is included in the Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation.

July 10–12, 2024 • Santa Clara, CA, USA

978-1-939133-40-3

Open access to the Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation is sponsored by

ARTIFACT EVALUATED usenix ASSOCIATION AVAILABLE

ACCL+: an FPGA-Based Collective Engine for Distributed Applications

Zhenhao He
Systems Group, ETH Zurich

Dario Korolija
Systems Group, ETH Zurich

Yu Zhu
Systems Group, ETH Zurich

Benjamin Ramhorst
Systems Group, ETH Zurich

Tristan Laan*
University of Amsterdam

Lucian Petrica
AMD Research

Michaela Blott
AMD Research

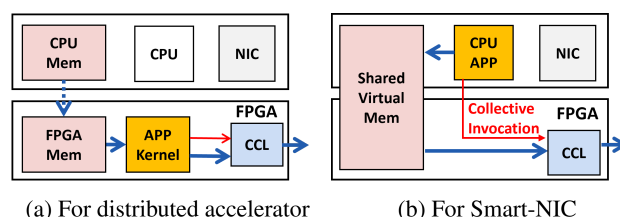
Gustavo Alonso
Systems Group, ETH Zurich

Abstract

FPGAs are increasingly prevalent in cloud deployments, serving as Smart-NICs or network-attached accelerators. To facilitate the development of distributed applications with FPGAs, in this paper we propose ACCL+, an open-source, FPGA-based collective communication library. Portable across different platforms and supporting UDP, TCP, as well as RDMA, ACCL+ empowers FPGA applications to initiate direct FPGA-to-FPGA collective communication. Additionally, it can serve as a collective offload engine for CPU applications, freeing the CPU from networking tasks. It is user-extensible, allowing new collectives to be implemented and deployed without having to re-synthesize the entire design. We evaluated ACCL+ on an FPGA cluster with 100 Gb/s networking, comparing its performance against software MPI over RDMA. The results demonstrate ACCL+'s significant advantages for FPGA-based distributed applications and its competitive performance for CPU applications. We showcase ACCL+'s dual role with two use cases: as a collective offload engine to distribute CPU-based vector-matrix multiplication, and as a component in designing fully FPGA-based distributed deep-learning recommendation inference.

1 Introduction

FPGAs are increasingly being deployed in data centers [16, 81] as Smart-NICs [29, 35, 64, 67, 103], streaming processors [31, 32, 55, 68], and disaggregated accelerators [15, 41, 61, 65, 86, 93, 115]. In scenarios where FPGAs are directly connected to the network, efficient distributed systems can be built using direct FPGA-to-FPGA communication. However, designing distributed applications with FPGAs is difficult. It requires both a network stack on the FPGA compatible with the data center infrastructure, and a higher level abstraction, e.g., *collective communication*, for more complex interaction patterns. Unlike in the software ecosystem where many such libraries exist [38, 76], there is a lack of similar resources for FPGAs. While new development platforms [53, 59] are improving FPGA programmability, and other recent efforts [10, 18, 60, 70, 72, 100] focus on virtualizing FPGA resources for abstracting data movement, they lack support for networking. This forces distributed applications on FPGAs to rely on the CPU for communication [22, 92, 116], thereby increasing the latency of data transfers between FPGAs. It has not been until recently that native networking support [14, 45, 56, 87, 95] has become available for FPGAs. But these systems lack collective communication, limiting their applicability in larger distributed use cases.



(a) For distributed accelerator (b) For Smart-NIC

Figure 1: Collective communication library (CCL) in different FPGA-accelerated systems, where the blue line indicates application data flow and the red line indicates collective invocation commands.

Implementing high-performance and versatile collective abstractions for FPGAs poses several challenges:

Challenge 1: Support of Diverse Transport Protocols. This requirement stems from the need for application-specific solutions and to ensure interoperability in mixed environments where FPGAs coexist with CPUs and accelerators. The ability to adapt to various communication protocols is crucial for integrating FPGA-based components seamlessly with other parts of a system. Existing work [25, 26, 36, 44, 73, 84, 85, 101, 104, 112] is often tailored to scenarios where FPGAs are directly connected to each other rather than connected through a data center packet-switched network. In these approaches, communication is through low-level link-layer protocols, leading to scalability and integration challenges at a data-center scale.

*Work done during internship at AMD Research

Challenge 2: Flexibility for Collective Implementation. A key challenge is to provide a flexible design in hardware allowing the selection of different collectives and algorithms at runtime. Current solutions [26,45] often integrate all possible collective modules directly into FPGA hardware, resulting in static primitives that necessitate hours-long recompilations for any changes. Alternative approaches, e.g., collectives using embedded micro-controllers (uC) [89] on FPGAs, offer more flexibility at the expense of performance.

Challenge 3: Portability Across Applications and Platforms. Portability stands out as a key challenge, as FPGAs are used in a wide variety of configurations. Figure 1a shows the FPGA collective communication library (CCL) as enabler for direct networking between FPGAs accelerators. It also demonstrates a *partitioned memory model* [54, 59, 60] for FPGA-centric applications, where an explicit memory copy is required if the data originates from CPU memory (dashed line). Figure 1b illustrates CCL's role as an collective offload engine for a CPU application with *shared virtual memory* [62,70,75,106]. This portability also raises questions about which *communication models* should be provided to the application. Should interfaces support message passing, i.e., MPI, where communication occurs between memory buffers, or streaming, where communication flows through continuous data streams?

In summary, the key question to address is *how to effectively design a portable, flexible, high-level collective abstraction on FPGAs that can support various memory models (e.g., partitioned and shared virtual memory model), communication models (e.g., message passing and streaming), and transport protocols (e.g., TCP and RDMA), while accommodating a broad spectrum of applications*. Achieving this objective is complex, given the significant impact of these configurations on runtime, interfaces, and data movement. Moreover, given FPGAs' extended compilation times and lengthy hardware debugging cycles, we need a parameterized approach that allows a FPGA-resident CCL to be modified without recompilation, in order for the CCL to be practical in real-world use. Table 1 summarizes existing FPGA-based solutions, and all existing solutions have their own limitations.

Our Contributions. To address these challenges, we introduce ACCL+, an **A**daptive **C**ollective **C**ommunication **L**ibrary on FPGAs. ACCL+ can be used to enable direct communication between FPGAs and can function as a collective offload engine for the CPU. ACCL+ provides MPI-like collective APIs with explicit buffer allocation and streaming collective APIs with direct channels to the communication layer. To achieve portability, we employ a modular system architecture which decouples platform-specific IO management and runtime from the collective implementation, incorporating platform and network protocol-specific adapters and drivers. For flexibility, we have developed a platform and protocol-independent collective offload engine that supports modifying the collective implementation without hardware recompilation. We test ACCL+ on two platforms, a commodity, partitioned memory platform (AMD Vitis [59]) and a shared virtual memory platform (Coyote [62]). We choose AMD Vitis for its recent integration of high-performance 100 Gb/s UDP [107] and TCP [45] hardware stacks, aligning with our goal of leveraging cutting-edge networking capabilities for optimal communication performance. Coyote is used due to its unique provisioning of unified and virtualized memory across CPU-FPGA boundaries [62], coupled with comprehensive network services, including RDMA. ACCL+ is also designed to minimize control overheads, improve scalability, and facilitate simulation.

Table 1: Comparison of ACCL+ with FPGA-based solutions in terms of bandwidth, flexibility in implementing different collectives, target application scenarios, and supported transport protocols.

Solution	BW (Gb)	Flex.	Application	Protocol
Easynet [45]	100	Low	FPGA	TCP
SMI [26]	40	Low	FPGA	Serial Link
Galapagos [97]	10	Low	FPGA	TCP
ZRLMPI [85]	10	Low	FPGA	UDP
TMD-MPI [89]	<10	High	FPGA	Serial Link
ACCL+ (Ours)	100	High	CPU/FPGA	UDP/TCP/RDMA

Key Results. We first evaluate ACCL+ using micro benchmarks. ACCL+ achieves a peak send/recv throughput of 95 Gbps, almost saturating the 100 Gb/s network bandwidth. We evaluate collective operations under two scenarios: FPGA-to-FPGA distributed applications with FPGA kernels directly interacting with ACCL+ (F2F), and CPU-to-CPU distributed applications with ACCL+ as a collective offload engine (H2H). ACCL+ exhibits significantly lower latency in F2F scenarios compared to software RDMA MPI for FPGA-generated data. In H2H scenarios, ACCL+ has comparable performance to software RDMA MPI for CPU-generated data while freeing up CPU cycles and reducing pressure on CPU caches. Then, we examine ACCL+ with two use-case scenarios. First, distributed vector-matrix multiplication with CPU computation and ACCL+-based reduction, where ACCL+ improves performance compared to software MPI. Secondly, we show that ACCL+ enables the distribution of an industrial recommendation model across a cluster of 10 FPGAs, achieving more than two orders of magnitude lower inference latency and more than an order of magnitude higher throughput than CPU solutions. The use case study not only highlights ACCL+'s effectiveness in different scenarios but also paves the way for future research opportunities in investigating hybrid CPU-FPGA co-design for distributed applications.

2 Background

FPGA Programming. In the past, hardware description language (HDL - Verilog, VHDL) was the sole method to pro-

gram FPGAs. With High-Level-Synthesis (HLS) [21], the programmability of FPGAs is enhanced by allowing the developers to program in C-like code with hints (pragmas) to infer parallel hardware blocks. Unfortunately, existing HLS-based libraries lack networking and collective abstractions.

Communication Models. Message passing, e.g., MPI, is a communication model for distributed programming on CPUs, whereby communicating agents exchange messages, i.e., user buffers, typically resulting from previous computation. This model can be applied to FPGAs [46, 84, 85], but a more common communication model for FPGAs is the streaming model. FPGA kernels support direct streaming interfaces, into which data can be pushed in a pipelined fashion during processing. Kernels executing on the same FPGA can stream data to each-other through low-level latency-insensitive channels, such as AXI-Stream [9]. The streaming model can be applied for communication across FPGAs, however, existing streaming communication framework [26, 33, 34] often do not have transport protocols or collective abstractions.

FPGA Development Platforms. Modern FPGA platforms adopt various virtualization methodologies [13, 83, 99] for FPGA resources. Most simplify development with a static *shell* for resource management and data movement, with some offering additional *services* like transport layer networking, and the host-device interaction relies on *runtime* libraries. This approach allows developers to concentrate on designing the application kernel. Many commodity platforms [54, 59] implement a partitioned memory model, which permits data movement from FPGA applications to FPGA memory while restricting direct access to host CPU memory. In contrast, shared virtual memory platforms, such as Coyote [62] and Optimus [70], offer a virtualized and unified memory space between CPU and FPGA.

Network-Attached FPGAs. FPGAs today feature 100 Gb/s transceivers, enabling direct processing of network data [81, 105]. FPGA-based Smart-NICs [2, 14, 28, 35, 66, 67, 98, 103, 119] perform programmable packet filtering but often leave the network stack to the CPU software, limiting their applicability to FPGA applications. Distributed machine learning [3, 12, 19, 77, 118], and data processing [24, 61, 65] applications capitalize on network-attached FPGAs. Following this trend, there is an increasing effort to develop hardware network stacks on FPGAs, such as UDP [47, 107], TCP [4, 27, 56, 87, 94], and RDMA [65, 69, 91, 95]. With the growing demands and the increasingly distributed nature of these applications, their communication patterns have become more complex, motivating the need for high-level collective abstractions on FPGAs. Therefore, simply offloading the network stack is often insufficient for complex applications in distributed settings.

3 Related Work

Collective for Accelerators. MPI implementation of collective communication are becoming more accelerator-aware, e.g., GPU-aware MPI [80, 102, 111] or FPGA-aware MPI [22]. In GPU-Direct RDMA collective libraries, e.g., NCCL [74] and RCCL [5], the network data can be directly forwarded to the GPU memory from the commodity RNIC via the shared PCIe switch, bypassing the CPU memory. However, FPGAs can connect directly to the network, and as such, the RDMA stack in ACCL+ resides completely in FPGA, eliminating the need for an external NIC. ACCL+ can provide streaming interfaces directly to FPGA kernels, in addition to the standard READs/WRITEs to memory used by commodity RNICs, therefore reducing latency by bypassing the memory hierarchy. Finally, the commodity RNICs typically lack offload capabilities, with collectives implemented on GPU cores, leading to computation and communication contention on the GPU which affects performance [51, 79]. Thus, ACCL+ could serve as a collective offload engine for GPUs in the future.

FPGA-based Collectives. Projects such as Galapagos [30, 97] and EasyNet [45] provide in-FPGA communication stacks for data exchange within a cluster, serving as a foundation for collectives without an external NIC. TMD-MPI [88, 89] orchestrates in-FPGA collectives using embedded processors, yet its bottleneck lies in control due to sequential execution in low-frequency FPGA microprocessors. Collective offload with NetFPGA [6–8] has been explored, but static collective offload engines limit flexibility and often rely on software-defined network switches for orchestration. SMI [26] proposes a streaming message-passing model, exposing streaming collective interfaces to FPGA kernels. While SMI enables kernels to initiate collectives directly, it employs dedicated FPGA logic for collective control, limiting flexibility for post-synthesis reconfiguration. In an earlier prototype, ACCL [46], we focused primarily on message-passing collectives for FPGA applications. However, the coordination of collectives required CPU involvement, it lacked significant streaming support, and was not tested at scale.

Collective Offload for CPUs. BluesMPI [11, 96] offloads collective operations to a BlueField DPU, demonstrating comparable communication latency to host-based collectives, but it does not target accelerator applications. The latency of ACCL+ targeting host data matches BluesMPI, even with BluesMPI ARM cores working at ten times the frequency.

Multi-FPGA Frameworks. Frameworks like ViTAL and its successors [112–114] propose FPGA resource virtualization and compilation flows for mapping large designs onto multiple FPGAs through latency-insensitive channels. OmpSs@cloudFPGA [25] introduces a multi-FPGA programming framework that partitions large OpenMP programs with domain-specific programs into smaller distributed parts for execution on FPGA clusters, providing communication through static, compile-time-defined send/recv and collective opera-

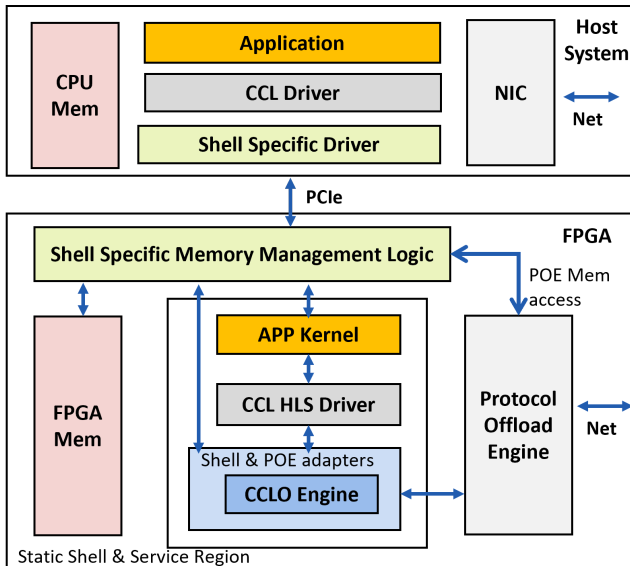


Figure 2: System overview of the FPGA-based collective communication library.

tions supporting only the unreliable UDP protocol. ElasticDF [3] and FCsN [40] present domain-specific frameworks for automatically distributing large neural network model inference across FPGAs with hardware UDP/TCP send/recv for FPGA-to-FPGA data movement. These projects are complementary to our work, and integrating ACCL+ will enhance their flexibility and performance.

4 ACCL+: An FPGA Collective Engine

ACCL+ is an FPGA-based collective abstraction designed for both FPGA and CPU applications, focusing on versatility and adaptability. Its primary goals include:

G1: Offering a standard collective API that abstracts different platforms and protocols from the application layer.

G2: Providing flexibility to dynamically select collectives and their algorithms at runtime, and to modify them without major architectural changes.

G3: Ensuring portability across various FPGA platforms and communication models for a wide range of applications.

G4: Supporting multiple transport protocols under a high-level collective abstraction.

G5: Providing high-throughput and low-latency performance for various collectives.

To achieve these goals, ACCL+ features a modular design that separates platform-specific and transport layer components from the core collective design. Its architecture includes layers of abstraction in both software and hardware, as shown in Figure 2, enabling a central CCL offload engine (CCLO) to adapt to diverse platforms and communication protocols. In this section we will describe each layer.

4.1 Application Interface

To satisfy **G1**, ACCL+ provides standard APIs for both CPU and FPGA applications. ACCL+ implements two drivers that offer similar, platform- and protocol-agnostic collective APIs for these scenarios. The ACCL+ drivers expose an MPI-like API, catering to the message-passing paradigm and facilitating the porting of existing MPI-based applications to ACCL+ collectives, and a streaming collectives API to overlap communication and computation in hardware.

ACCL+ Drivers. The host-side CCL driver allows initialization and runtime management of platform and ACCL+ data structures and hardware, as well as protocol offload engine (POE) initialization, i.e., setting up sessions for TCP or queue-pairs for RDMA. The CCL HLS driver is not capable of such initialization, and therefore the application must perform host-side initialization before any FPGA application kernels are started. We provide a more detailed description of ACCL+ initialization in Appendix A.

Listing 1: Reduce collective API in C++.

```
1  CCLRequest *reduce(BaseBuffer &buf, unsigned int
       count, unsigned int root, reduceFunction func,
       communicatorId comm_id, flagType flags);
```

MPI-like Collective API. This API require the application to store data in memory before invoking collectives. Listing 1 shows the MPI-like collective API, including arguments like datatype, buffer pointer, and element count, along with flags indicating buffer location (host or FPGA memory) and the option for synchronous calls. To facilitate portability, message passing collectives operate on an ACCL+ specific buffer class which can wrap normal C++ arrays with additional platform-specific information. Common collectives, such as reduce, broadcast, and barrier, are supported. Each MPI-like collective call in the host CCL driver has a corresponding HLS API call with a similar syntax for direct invocation from FPGA kernels.

Streaming Collective API. This API allows data to originate and terminate at the stream interfaces between the FPGA application kernels and the ACCL+ hardware, instead of in memory buffers.

Listing 2: Example kernel using streaming send in HLS.

```
1  // set up command and data interfaces
2  cclo_hls::Command cclo(cmd, sts, communicator);
3  cclo_hls::Data data(data_to_cclo, data_from_cclo);
4  // issue streaming send command without buffer argument
5  cclo.send(type, count, dst_rank);
6  // push data in streams to network without buffering
7  for (int i = 0; i < N; i++) {
8      data.push(/* generate data */); }
9  cclo.finalize(); // wait for send completion
```

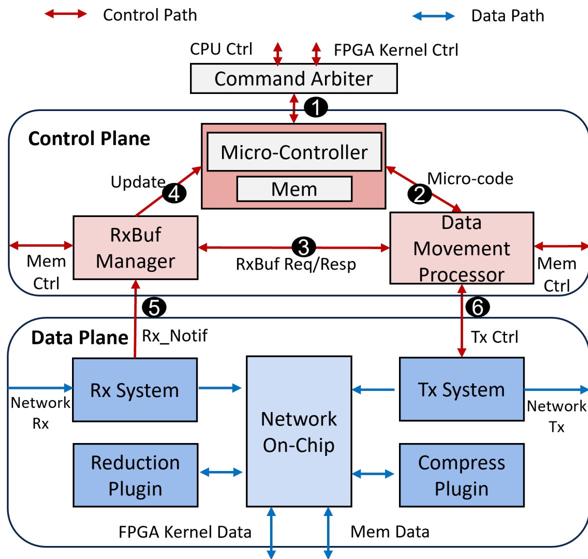


Figure 3: Hardware architecture of CCLO engine.

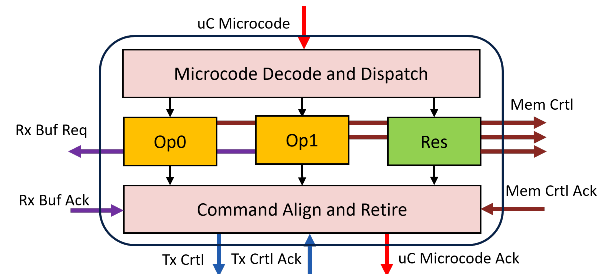


Figure 4: Architecture of the Data Movement Processor.

Listing 2 demonstrates an example FPGA kernel issuing a streaming send command to the CCLO engine (line 5) and subsequent pushes to the CCLO streaming data interface, 64B per cycle (line 8), followed by a wait for CCLO completion. The HLS-based streaming APIs are tailored for FPGA applications running in a streaming fashion and this code is synthesizable with HLS tools. HDL-based FPGA kernels can interact with the collective engine directly, through the same interfaces. Additionally, the host can also call streaming collectives via the host-side CCL driver.

4.2 CCLO Engine

Our approach to satisfying **G2**, i.e., achieving flexible collective implementation in hardware, differs from related work. One method deploys all collective modules in FPGA fabric simultaneously [26, 45], consuming extensive resources and not allowing modifications to the collective algorithms without recompiling the entire design. Another method pursues flexibility by implementing the collectives in embedded micro-controllers (uC) on FPGAs [89], which are often limited by a low clocking frequency, e.g., 200 MHz, and the sequential execution nature, thus sacrificing performance.

Our Approach. We utilize a hybrid approach that leverages the strengths of both methods. To ensure flexibility, low latency, and high throughput, the key design principle is to *decouple the CCLO logic into the flexible control plane and the parallel data processing plane*. The CCLO control plane is flexible, centered around an embedded uC [108], which enables the implementation of different collective algorithms through firmware updates without needing to refactorize the entire design and re-synthesize. The CCLO data plane contains independent latency-optimized hardware modules with wide data path for concurrent execution. Moreover, to further reduce the load on the uC, *we minimize its code footprint by offloading tasks* such as packet assembling and tag matching to hardware. Additionally, interactions with memory controllers are offloaded to dedicated hardware, preventing the uC from stalling during memory accesses. As a result, the uC handles a set of high-level data movement primitives that facilitate the implementation of the actual collective algorithms.

Figure 3 shows the overall architecture of the CCLO engine, which orchestrates the collective data movement through a set of standardized CCLO interfaces to interact with the application, the memory and the network. The CCLO accepts communication requests from the host or application kernels, communicates with the protocol offload engine, manages buffers in FPGA memory (HBM, DDR, BRAM), and manage data streams from other kernels.

4.2.1 Flexible Control Plane

The CCLO control plane contains a uC that issues high-level data movement commands to a hardware-accelerated data movement processor (DMP). The CCLO control plane also contains a RxBuf Manager (RBM), which manages temporary Rx buffers. The uC, DMP, and RBM store states in a small configuration memory implemented as FPGA BRAM. The configuration memory is also accessible by the CPU through MMIO and includes information about the communicator, e.g., session or queue pair IDs, pool of allocated Rx buffers. Besides, FIFO queues are incorporated into all command paths, allowing multiple in-flight instructions. Currently, these FIFO queues are set to a depth of 32, which can be further increased at compile time.

Collective Programming with Primitives in uC. The uC firmware implements various collective algorithms and synchronization protocols, such as eager and rendezvous, using high-level primitives. Each primitive instruction consists of three slots: two for operands (data entering CCLO) and one for the result (data exiting CCLO). This design aligns with common collective operations, e.g., reduce, which processes two inputs to produce one output. Unary operations like send can disregard one operand slot. Operand slots include opcodes and flags that define the data movement specifics, dictating

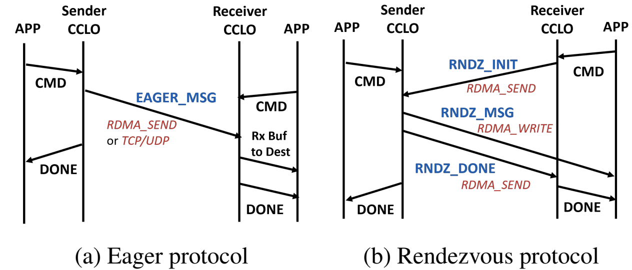
when and *where* data should move. For instance, data can be moved immediately or upon arrival, sourced from a memory buffer when using the MPI-like API, or from the data interface of the FPGA kernel when using the streaming API. Additionally, the data can either be sent to a remote node through the network or remain local as intermediate results. These elements can be combined to cover nearly all data movement needs in collective operations.

Data Movement Processor. The primary purpose of the DMP is to conceal memory access latency from the uC, ensuring that the uC does not stall for memory accesses or wait for data streams, as shown in Figure 4. Upon the receipt of the microcode generated by the uC ❷, the DMP first decodes the microcode and dispatch the code to different compute units (CUs). The DMP primarily consists of three CUs, aligning with the structure of the primitive, each responsible for controlling one or more components in the datapath. If the microcode indicates to fetch data from memory and forward it to the network, the CU issues memory requests to the target address and then issues the Tx control ❻ to the data plane, ensuring the data plane waits for incoming memory streams to forward to the network. If the operand is expected to come over network and buffered in temporary buffers, the DMP also sends out requests periodically to the RBM to check if the message has arrived ❸. The DMP operates in a pipelined fashion, and each operand slot independently interprets its instruction fields, emitting commands for corresponding datapath blocks. Upon receiving acknowledgements from datapath blocks, the DMP signals instruction completion to the uC.

RxBuf Manager. The RBM alleviates uC load by autonomously managing temporary Rx buffers and reassembling messages from network packets, especially under the eager protocol. It uses a state table in FPGA on-chip memory and a set of finite-state machines (FSMs) to handle Rx buffers. Upon notification of incoming messages ❺, RBM checks the state table using the message ID. If the message is new, it identifies a free Rx buffer from the configuration memory and issues requests to store the message there. Since messages are often split into packets that may arrive interleaved, RBM uses the state table to piece together packets into complete messages in the appropriate Rx buffer. When a full message is assembled, RBM updates the exchange memory's buffer list ❹, marking it ready for retrieval, and stores essential metadata (source ID, tag, Rx buffer address) for tag matching, facilitating buffer identification by the DMP.

4.2.2 Parallel Data Plane

Rx and Tx System. In ACCL+, we implement a message protocol that includes a signature for each message. This signature contains metadata such as message type, destination rank, length, tag, and a sequence number to track the order of messages. The Tx and Rx systems feature a 512-bit wide data path and are responsible for packetizing and depacketizing the signature along with the user payload. They also issue commands to interact with the POEs. The processes of issuing commands, inserting signatures, and parsing can vary across different synchronization protocols. Both the Rx and Tx systems incorporate a finite state machine to manage these variations appropriately.



(a) Eager protocol (b) Rendezvous protocol

Figure 5: CCLO eager and rendezvous with send/recv.

Network On Chip. All the data streams internal to the CCLO can be routed in the granularity of packets based on the dest field that comes along with the data.

Streaming Plugins. The plug-ins are utilized for applying unary and binary operations to in-flight data and can be enabled at compile time. Binary operations are typically utilized to implement reductions - sum, max, etc. Unary operators may implement compression or encryption. Each of the plug-ins is a streaming kernel and may implement more than one function, in which case the control plane will specify the desired function by setting its dest field of the plugin input stream.

4.2.3 Message Synchronization Protocol

The CCLO supports two distinct message synchronization protocols: eager and rendezvous. Thanks to its flexible design, both protocols can be tuned dynamically at runtime.

The eager protocol allows the sender CCLO to immediately send data upon receiving a command, and the receiver buffers the data in the CCLO Rx buffer before moving it to its destination (either in memory or in FPGA kernel streams depends on runtime configuration), as shown in Figure 5a. This protocol is preferred for small messages to minimize latency since there is no handshake phase and small message sizes incur little overhead. We implement the eager protocol using UDP/TCP or two-sided RDMA verb.

In contrast, the rendezvous protocol requires resolving the result buffer address before transmission, as shown in Figure 5b. Once resolved, data is directly placed into the destination, eliminating the need for temporary buffering. We use two-sided RDMA `SEND` for rendezvous handshake messages, and we use one-sided RDMA `WRITE` for actual message transmission bypassing the intervention from the receiver uC. Given that one-sided RDMA operations are transparent to the receiver uC, a key design decision is how the uC should detect message arrival. One approach is to make the uC pe-

Table 2: Algorithms used for example collectives.

Collective	Eager	Rendezvous
Bcast	One-to-all	One-to-all;Recursive doubling
Reduce	Ring	All-to-one;Binary tree
Gather	Ring	All-to-one;Binary tree
All-to-all	Linear	Linear

riodically poll the destination buffers in memory. However, this approach increases uC overhead and latency, especially since buffers may be located in various memory systems like CPU or FPGA memory. Additionally, if the destination is a streaming interface rather than a buffer, polling is not feasible. Therefore, we choose an alternative method: the sender uC dispatches a small control message using two-sided RDMA `SEND` immediately after the one-sided RDMA `WRITE`. This control message is processed by the receiver uC to confirm the completion of the data transfer. Though not depicted in Figure 3, the uC contains specific ports that directly interact with the data plane for rendezvous handshake and control messages, bypassing the RBM and DMP. These command paths also incorporate FIFO queues.

4.2.4 Collective Algorithms

We provide different implementations for various collectives, and users can define their own. Collectives are realized by specifying a communication pattern as a C function in uC firmware, and then executing this pattern through instructions in DMP and Tx/Rx System on each FPGA in the communicator. Table 2 summarizes the algorithms and communication patterns used to implement collectives. For eager protocols with unreliable transmission (e.g., UDP), we currently use simple algorithms like ring and one-to-all to minimize the chances of packet loss. Future firmware improvements can enhance POE awareness for finer-grained algorithm selection. In contrast, when using RDMA, the rendezvous protocol employs more advanced algorithms like tree or recursive doubling. The token-based flow control in RDMA makes it well-suited for these sophisticated algorithms in the rendezvous protocol. For broadcast, we implement a simple one-to-all algorithm with small rank size, while with large rank size, we adopt more advanced recursive doubling such that the data transmission is not bottlenecked at the root rank. For gather and reduce, we apply a similar strategy. With small message size, we adopt an all-to-one approach to reduce the number of intermediate hops needed. On the other hand, with larger message sizes, to avoid a potential in-cast problem with the all-to-one approach, we adopt a tree-based algorithm. Tuning of the algorithms for specific collectives can be done at runtime through configuration parameters to the CCLO engine.

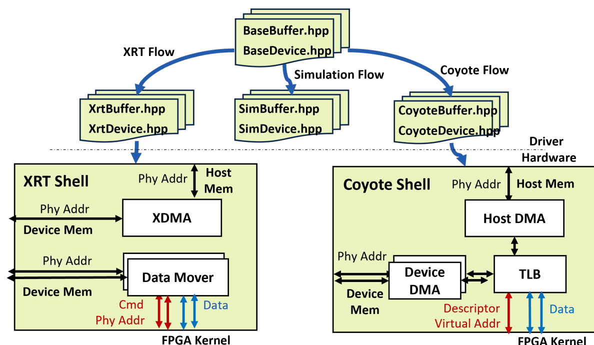


Figure 6: CCL driver for different memory managements.

4.3 ACCL+ Platform Support

A platform is defined by a software interface specification, defining how FPGA memory is allocated and manipulated, and how FPGA kernels are called, and a hardware interface specification, i.e., how FPGA kernels, including the CCLO, plug into hardware services in the FPGA. To facilitate portability between platforms and to satisfy **G3**, the ACCL+ host CCL driver layers the APIs on top of generic class types, such as BaseBuffer for memory allocation and data movement between host and FPGA, and BaseDevice for CCLO invocation. These are specialized to individual platforms through class inheritance, as illustrated in Figure 6. Each specific CCL class interfaces with platform-native drivers and employs distinct processes for handling data movement. ACCL+ supports both the commercial AMD Vitis platforms and the open-source Coyote platform [62], as well as a virtual simulation platform. New platforms can be added easily.

Integration with Coyote. Coyote utilizes a shared-memory model with a central memory management logic governed by a software-populated translation lookaside buffer (TLB). This TLB records allocated pages and facilitates virtual-physical address translation. The FPGA kernel issues memory requests through a descriptor interface, using virtual addresses directed to either host or device memory. The TLB interprets these requests, interacting with host DMA or device DMA based on the physical location of the memory page, forwarding the data to FPGA applications in a streaming manner. If a memory page is not registered during TLB lookup, it triggers an interruption to the CPU, resulting in a page fault and introducing a performance penalty. Therefore, the CCL driver, specifically the CoyoteBuffer class, eagerly maps pages to the Coyote TLBs when instantiating buffers. We modified Coyote, during integration, to increase the associativity of the TLB cache and expand the number of streaming interfaces Coyote provided to a single application region, from a single interface to three interfaces which is required by the CCLO engine. We also implemented a Coyote-specific adapter to convert from CCLO (R)DMA request syntax to Coyote-specific syntax, as indicated in Figure 7.

Integration with Vitis. Vitis platforms implement a partitioned memory model and the Xilinx Runtime (XRT) [59] is utilized by the CCL driver for low-level interaction with the platform. A XRT-controlled XDMA IP core [110] moves data between host and FPGA memory, while FPGA memory is accessed by FPGA kernels through Data Movers [109]. The CCLO memory interfaces align with the Data Mover interfaces, eliminating the need for dedicated memory interface adapters for the Vitis platform. As a result of the partitioned memory, the CCL driver explicitly migrates buffers between host and FPGA memory prior to or after the collective execution if the data originally resides in host memory - a process denoted staging. Staging creates performance penalty when ACCL+ collectives target host memory, as observed by related work on collective offload on DPUs [96]. Therefore, Vitis platforms favor distributed FPGA applications where data is streamed or resides in FPGA memory.

Simulation Platform We implemented an additional simulation platform for debugging and performance optimization. This simulation platform roughly models a Vitis platform, whereby FPGA chip interfaces (XDMA, Ethernet) are replaced by ZMQ [48] interfaces. A stand-alone simulated FPGA node is compiled to include memory and one ACCL+ CCLO Engine. The ACCL+ host driver includes dedicated buffer and device abstractions capable of connecting to the simulated node via ZMQ. ACCL+ provides convenient launch scripts to set up a simulated cluster of such simulation nodes.

The simulated nodes connect to each other through ZMQ rather than real Ethernet. While the simulated ZMQ network may lack realistic features like packet loss and reordering, it serves as a valuable functional simulation.

ACCL+ provides two simulation levels of the CCLO engine: functional simulation using compiled ACCL+ HLS source code and C firmware, and cycle-accurate (but slow) simulation using Verilog HDL generated from compiling the CCLO HLS code and firmware. For FPGA applications requiring streaming data exchange between FPGA kernels and the CCLO, we provide a bus functional model of the CCLO that connects via ZMQ to the simulated node.

4.4 Protocol Offload Engine

To satisfy **G4**, ACCL+ supports several 100 Gb/s protocol offload engines (POE) in hardware: UDP [107] and TCP [45] on Vitis platforms, and all the network services provided by Coyote. Notably, ACCL+ supports collectives with RDMA by leveraging the unified and virtualized memory space across the FPGA and the CPU provided by Coyote. All the POEs expose streaming control and data interfaces to other modules and some POEs (e.g., TCP) require direct memory access for packet buffering for re-transmission. For portability, the CCLO Engine has a set of POE-independent internal interfaces - two pairs of meta and data streaming interfaces (one for Tx and one for Rx). The meta interfaces contains various

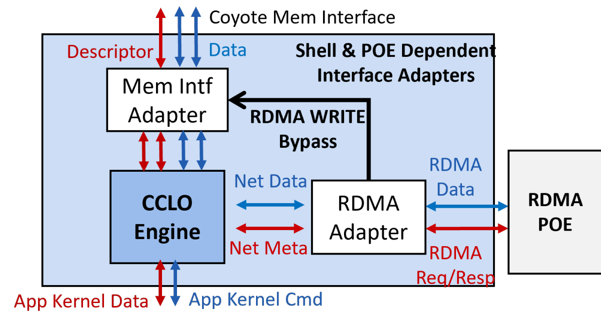


Figure 7: ACCL+ with Coyote-RDMA data path with corresponding POE and memory adapters.

sub fields to indicate the op code, data length, communication session IDs, etc. The meta interfaces are then adapted to the POE interfaces with dedicated FPGA components as exemplified in Figure 7. The selection of the POE and its adapters is a compile time parameter of the CCLO Engine.

Coyote RDMA POE. It supports standard RDMA verbs, including one-sided operations like `WRITE` and two-sided operations like `SEND`. The RDMA POE incorporates various streaming interfaces for RDMA commands, memory commands, and data. *Default Configuration:* On the initiating side of a `WRITE` operation, the RDMA POE issues memory requests directly to the Coyote memory management logic, fetching data from either host or device memory and streaming it through the network. On the passive side of `WRITE`, the data is directly written to virtualized memory. *ACCL+ Integration:* In the ACCL+-enabled configuration, the CCLO engine acts as a "bump-in-the-wire" engine between the memory management unit and the RDMA POE, as shown in Figure 7. On the initiating side of a `WRITE`, the CCLO engine issues RDMA commands and is responsible for data preparation, either fetching from memory or obtaining it from the application kernel in the form of streams. On the passive side, data bypasses the CCLO and is directly forwarded to the memory management unit. For single-sided `WRITE`, streaming into the application kernel is also possible by configuring the datapath at compile time. The CCLO engine consistently manages data and metadata streams from two-sided `SEND`. For CCL driver with RDMA, a queue pair needs to be exchanged between each node and needs to be registered to the RDMA POE.

TCP POE. The TCP POE supports up to 1,000 connections and can be configured to support window scaling and out-of-order packet processing. As a reliable transmission protocol, the TCP POE also needs to access protocol-internal buffers for re-transmission. The CCLO engine prepares and accepts all the data streams with the TCP POE. For CCL driver with TCP POE, a TCP session needs to be established between each node to construct the communicator.

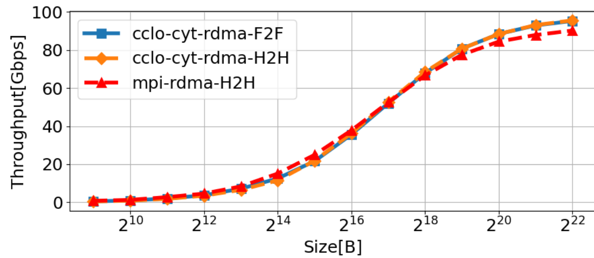


Figure 8: Send/Recv throughput comparison.

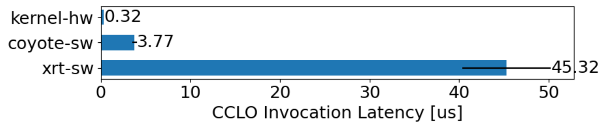


Figure 9: CCLO invocation latency from different parts.

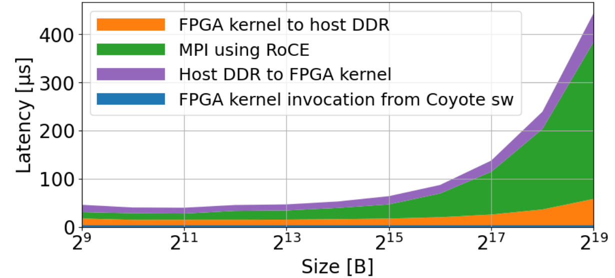


Figure 10: Latency breakdown of broadcasting FPGA produced data using software MPI with eight ranks with Coyote.

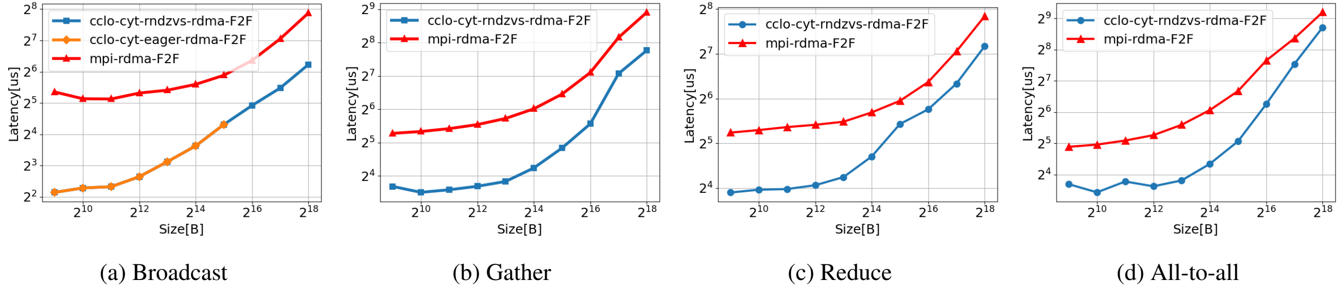
5 Microbenchmark Evaluation

We evaluate ACCL+ on a heterogeneous cluster with 10 AMD EPYC CPUs and 10 attached FPGA cards (Alveo-U55C). Each CPU is equipped with a 100 Gb/s Mellanox NIC, while each FPGA features a 100 Gb/s Ethernet interface. All devices are connected to Cisco Nexus 9336C-FX2 switches. Evaluation scenarios consider data residing on the FPGA for distributed FPGA application (suffix *F2F*) and on the CPU for distributed CPU applications (suffix *H2H*). For F2F, the FPGA application data traverses the network directly through ACCL+. As a baseline, the FPGA data initially is moved to the CPU memory and then is transmitted via a commodity NIC. In H2H, the CPU application data is transferred to the FPGA and then transmitted with ACCL+. This is compared to transmitting the CPU data directly with a commodity NIC. We use the notion of *cclo* with different suffixes to indicate different configurations of ACCL+. The focus of these experiments is evaluating RDMA running with Coyote (suffix *cyt*) due to space limitations. We nevertheless present some results with ACCL+ running TCP on top of the Vitis XRT (suffix *xrt*) platform to compare it to ACCL [16], which utilizes an embedded micro-controller to orchestrate collective operations. Experiments configure both MPI-like collectives with memory pointers and streaming collectives. For the H2H experiments, MPI-like collectives are mandatory, while the F2F experiments are configured to run with streaming collectives. ACCL+ operates at 250 MHz in micro-benchmarks, with varying frequency in the use-case study due to the design complexity. The comparisons involve MPICH 4.0.2 with TCP and OpenMPI 4.1.3 compiled with RDMA using OpenUCX 1.13.1 on the cluster CPUs and Mellanox 100 Gb/s NICs. MPI libraries self-configure for collective algorithms and synchronization protocols. Each micro benchmark experiment is averaged over 250 runs.

Send/Recv Throughput. We first evaluate pure throughput using send/recv. Figure 8 shows the throughput comparison of ACCL+ with Coyote RDMA and software MPI with RoCE backend. Notably, ACCL+ with RDMA achieves a peak throughput of 95 Gb/s, nearly saturating the network bandwidth. Compared to software MPI variants, ACCL+ exhibits comparable and slightly higher peak throughput. This is attributed to the FPGA network stack's ability to process network packets at line-rate in a pipelined fashion. Moreover, there is minimal distinction between F2F and H2H for ACCL+, thanks to the unified memory space provided by Coyote and both host memory access through PCIe and FPGA memory access offer higher bandwidth than the network.

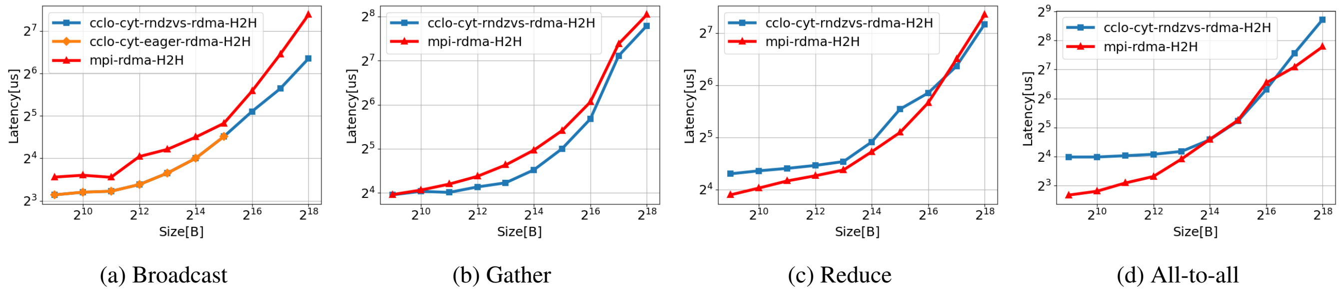
Invocation Latency. Figure 9 shows the invocation latency of the CCLO engine to execute a dummy NOP operation, which includes the time from receiving request untill the acknowledgement. For FPGA kernels that can directly interact with the CCLO engine, the invocation latency is minimal compared to software invocation from the host, showing a clear benefit of bypassing host control with FPGA-based applications. Coyote software driver contains a thin and optimized layer for invocation and scheduling and the resulting CCLO invocation time mainly consists of a PCIe write and a PCIe read latency. In contrast, the XRT invocation latency is significantly higher as it is not intended for fine-grained data movement.

FPGA-to-FPGA with Software MPI. To enable a more direct ACCL+ vs. software MPI comparison for executing collectives between kernels on FPGA, we model the execution time for MPICH- and OpenMPI-based device-to-device data movement, which includes: (1) moving data from FPGA HBM/kernel to host DDR through the PCIe, (2) executing the collective using software MPI, (3) moving data from host DDR to FPGA HBM/kernel, and (4) invoking the next computation kernel. We use the CCLO host invocation time as an approximation of the invocation time of other computation kernels. We measure the duration of each of the above steps and present a break-down of execution time of the collective with Coyote platform in Figure 10. We could observe that the PCIe transfer time is dominant for small messages while the collective time is dominant for large messages. Such



(a) Broadcast (b) Gather (c) Reduce (d) All-to-all

Figure 11: Collective latency comparison between ACCL+ RDMA and software MPI RDMA with eight ranks and device data



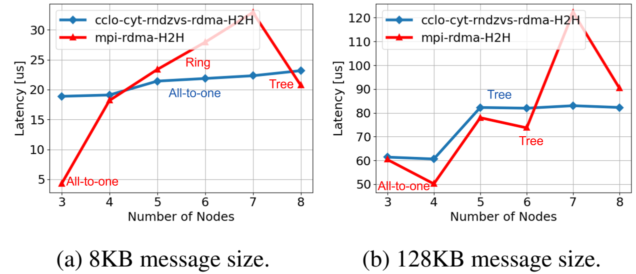
(a) Broadcast (b) Gather (c) Reduce (d) All-to-all

Figure 12: Collective latency comparison between ACCL+ RDMA and software MPI RDMA with eight ranks and host data

breakdown for XRT platform can be derived by changing the Coyote invocation latency to XRT invocation latency.

F2F Collective Latency RDMA. Figure 11 illustrates the latency of ACCL+ RDMA collectives with various message sizes on eight Alveo-U55C boards. This is compared to FPGA-to-FPGA data movement with software MPI over RDMA. For clarity, we present experiments showcasing better performance between eager and rendezvous collectives. The algorithms for each collective in ACCL+ are detailed in Table 2. Notably, ACCL+ exhibits significant performance benefits compared to its software counterpart. This advantage stems from the hardware's efficient execution of collectives and the direct network access within the FPGA device, eliminating the need for data copying to CPU memory.

H2H Collective Latency RDMA. Figure 12 illustrates a latency comparison between ACCL+ and software MPI targeting host data. The performance gains with ACCL+ vary across different collectives. Notably, for operations like broadcast and gather, ACCL+ consistently outperforms software MPI across a range of message sizes. However, for other collectives such as reduce and all-to-all, ACCL+ shows only marginal benefits and, in some cases, falls short of software MPI performance. One reason is that software MPI adapts its algorithms more finely to different configurations, whereas ACCL+ currently supports only a limited set of options. However, by offloading collective to hardware, CPU cycles could be freed for other computation tasks. Besides, by comparing ACCL+ F2F and H2H performance, we could observe that the ACCL+ collective latency has minimal difference because



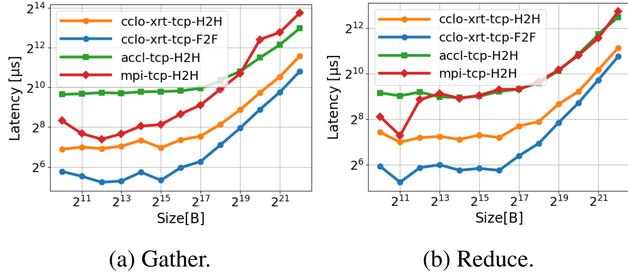
(a) 8KB message size. (b) 128KB message size.

Figure 13: Latency vs. rank sizes (Reduce).

Coyote with unified memory allows direct memory access to both host and FPGA memory.

Effect of Synchronization Protocol. Despite the simpler algorithms used by most eager collectives, such as one-to-all or ring, we observe that eager collectives can sometimes outperform rendezvous collectives with small message sizes, as seen in broadcast. This is because eager collectives do not require a handshake to resolve addresses.

Collective Algorithm and Scalability. Figure 13 illustrates the impact of algorithm selection and scalability on both ACCL+ and software MPI during collective executions. For an 8 KB message size, ACCL+'s reduce operation adopts an all-to-one algorithm, resulting in minimal latency increase across nodes. However, recognizing potential bottlenecks at the root node with this approach, ACCL+ switches to a binary tree algorithm for larger message sizes, such as 128 KB. In this case, an increase in latency is observed after



(a) Gather. (b) Reduce.

Figure 14: Comparison of collective performance between ACCL+ TCP with XRT, software MPI TCP and ACCL TCP.

four nodes, stabilizing until eight nodes due to a consistent tree depth. On the other hand, software MPI exhibits a more fine-grained approach to algorithm selection based on the scale of the message size and the number of nodes. For instance, it deploys three distinct algorithms within the 8 KB range: an all-to-one algorithm for fewer than four nodes, a ring protocol for four to eight nodes, and an optimized binomial algorithm for 8 nodes. Additionally, for larger messages, software MPI switches between an all-to-one algorithm below three nodes and a binomial tree algorithm between four and eight nodes. This fine-grained algorithmic tuning contributes to its superior performance in certain H2H scenarios. While software MPI's approach involves detailed algorithmic tuning, ACCL+'s flexible design allows for potential future enhancements through additional fine-grained tuning to further optimize performance.

XRT Platform and TCP. In Figure 14, we evaluate ACCL+ TCP with the XRT platform and compare it against software MPI with TCP. We also include a comparison with ACCL [46] collectives, which employs a similar embedded processor to orchestrate collectives and supports TCP on the XRT platform. Notably, ACCL+ TCP consistently outperforms its software counterpart across all configurations, benefiting from the line-rate processing capabilities of a hardware TCP POE. Furthermore, ACCL+ demonstrates superior performance compared to ACCL. While both ACCL+ and ACCL utilize embedded microprocessors for collective orchestration in hardware, ACCL+ distinguishes itself by offloading more tasks to the hardware data plane, such as utilizing the RxBuf Manager for packet assembling. In contrast, ACCL relies more on the microprocessor, leading to lower performance. When comparing ACCL+ TCP for serving host applications and device applications, a significant overhead is observed for host applications. This is attributed to the limitation of XRT platform, which prohibits direct data movement from the FPGA kernel to host buffers, resulting in a memory-copy overhead. Additionally, the XRT software invocation latency is notably higher, as indicated in Figure 9.

Table 3: Parameters of the target recommendation model.

Tables	Concat Vec Len	FC Layers	Embed Size
100	3200	(2048, 512, 256)	50GB

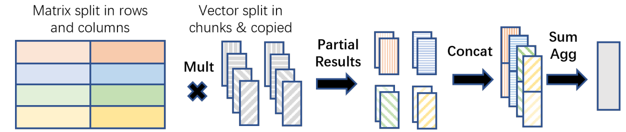


Figure 15: Checkerboard block decomposition.

6 Case Study: Deep Learning Recommendation Model

Deep Learning Recommendation Models (DLRM) are widely used in personalized recommendation systems for various scenarios [23, 37, 117]. The structure of a DLRM includes two major components: memory-bound embedding layers and computation-bound fully-connected (FC) layers. These models handle both dense and sparse features, with the latter stored as embedding vectors in tables. In inference, these vectors are accessed via indexes, resulting in multiple random memory accesses. The retrieved embedding vectors are then concatenated with dense features and passed through several FC layers to predict the click-through rate, incurring heavy computational loads due to vector-matrix multiplication.

DLRM has been a focal point for acceleration on GPUs and FPGAs, given that CPU solutions are generally constrained by both random memory access and computation [43, 49, 63]. GPU-based solutions [42, 49, 52, 58] mostly accelerate the computation-bound FC layers to gain high throughput. However, the large batch sizes required for efficient GPU computation, coupled with random memory access, often lead to increased latency (tens of milliseconds). FPGA-based techniques [57, 71] overcome the embedding lookup bottleneck by distributing tables across memory banks and enabling parallel accesses, leveraging high-bandwidth-memory (HBM) and on-chip memory (BRAM/URAM). However, this approach is constrained by the requirement for embedding tables to fit within a single FPGA's memory (e.g., 16 GB HBM on AMD Alveo-U55C), limiting the size of embedding layer. Additionally, the finite computational resources on a single node pose restrictions on overall throughput for all FC layers.

6.1 Distributed DLRM Inference

We aim to demonstrate that ACCL+ can facilitate distributing DLRM inference across FPGAs to accommodate larger embedding layers, as in many large-scale industrial settings, while at the same time achieving low latency and high throughput. Table 3 shows the detailed configuration of such an industrial-level recommendation model [58]. In such a use

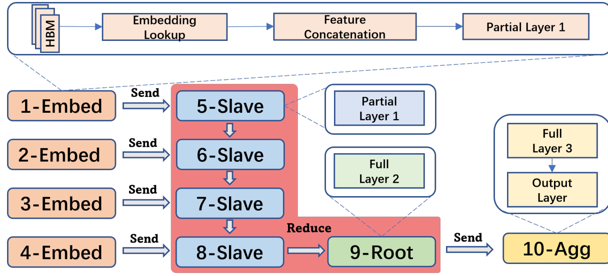


Figure 16: Conceptual design of partitioned DLRM, with $FC1$ decomposed and $FC2$, $FC3$ pipelined across nodes.

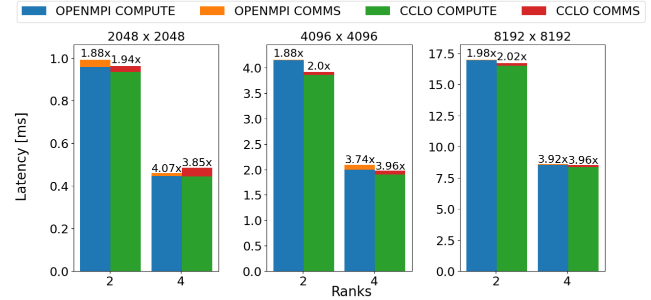


Figure 17: Speedup comparison and latency breakdown of distributed vector-matrix multiplication.

case, the embedding table does not fit into a single FPGA HBM and therefore both the embedding lookup and the computation are distributed across the network. This poses significant challenges for performance, scalability, and networking.

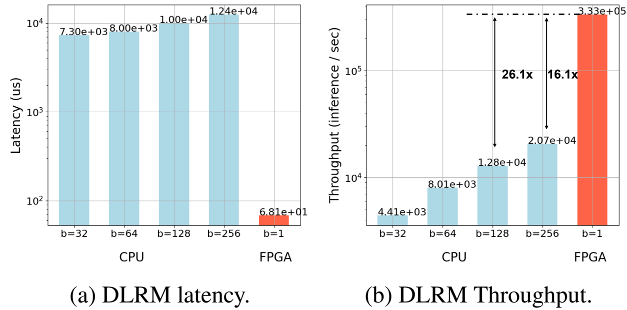
Vector-Matrix Multiplications Decomposition for DLRM. The computation pattern in DLRM inference involves a chain of three vector-matrix multiplications, with the inference output vector computed as a sequence of operations involving three matrices of FC layers ($FC1$, $FC2$, $FC3$) and a concatenated embedding vector. The concept of distributed vector-matrix multiplication has been extensively studied in literature [90] across CPUs and the same principle can be applied to an FPGA cluster. One common approach is *checkerboard block decomposition of matrix*, as shown in Figure 15. This method involves partitioning the matrix in terms of both rows and columns, while partitioning the vector ensures that processes associated with the same matrix row partition share the same sub-embedding vector. Each process can then perform partial computations, and the results belonging to the same row partition are concatenated and subsequently aggregated.

Decomposed and Pipelined Distributed DLRM. The partitioning strategy for the DLRM considers the need for balanced resource utilization, ensuring that the overall throughput is not limited by any process among all nodes. Typically, the computation load of the $FC1$ is significantly larger than subsequent layers like $FC2$ and $FC3$. To accommodate this, resource distribution should reflect the varying computation requirements. Additionally, for modern FPGAs with HBM, the capacity requires a minimum number of FPGAs to effectively store the embedding layer. A conceptual partitioned DLRM is illustrated in Figure 16. In this scenario, $FC1$ is decomposed and distributed across multiple FPGAs using the checkerboard block decomposition, and $FC2$ and $FC3$ are assigned to one FPGA each. The embedding tables are evenly distributed across nodes 1-4, with partial vectors transmitted to nodes 5-8, leveraging the network's low latency. Similarly, partial results computed on nodes 1-4 are forwarded to corresponding nodes 5-8, where an overall reduction of all partial $FC1$ results is conducted. The aggregated $FC1$ results are then forwarded to node 9 for $FC2$ computation, followed by node 10 for $FC3$ computation and final processing. Scaling resources according to the computation distribution requirements of each layer could lead to improved performance. For example, increasing the allocation of FPGAs for different layers based on their computational load. Such partitioning method requires diverse communication patterns by each node, such as send-only, send/recv, and reduction and ACCL+ provides a unified design supporting all the communication requirements of the DLRM through a standard interface. Additionally, for nodes that do not require reduction, the streaming reduction plugins of ACCL+ can be removed with a compilation flag, reducing resource consumption and improving routing and timing. Furthermore, the cross-node simulation provided by ACCL+ can facilitate the development process, reducing hardware debugging cycles.

6.2 Use Case Evaluation

Distributed FC Layer Execution on CPU. We use an illustrative example to demonstrate how ACCL+ can be utilized to improve the efficiency of distributed work executing on CPU. In this use case, we distribute an FC layer workload (matrix-vector multiplication) by partitioning the weight matrix column-wise, with each rank receiving part of the input vector and a subset of the weight matrix columns. The matrix-vector product is obtained by summing the partial rank products using the reduce collective. For the implementation, we use the highly optimized Eigen library [39], distributing it with both ACCL+ RDMA and MPI RDMA. In this experiment we do not overlap computation and communication.

The overall execution time of the distributed FC layer is compared to its single-node execution, as depicted in Figure 17, where top-of-bar numbers indicate the speed-up compared to single-node execution. We observe that utilizing ACCL+ instead of MPI for the reduction generally results in lower matrix-vector computation time. This performance increase is most likely due to reduced pressure on the CPU cache, as ACCL+ utilizes FPGA memory for all intermediate reduction data structures. The figure indicates two instances of super-linear scaling, attributed to the weight matrix partitions fitting into either L2 (8 MB) or L3 (128 MB) caches



(a) DLRM latency.

(b) DLRM Throughput.

Figure 18: ACCL+ DLRM performance comparison.

Table 4: Resource utilization.

Component	CLB kLUT	DSP	BRAM	URAM
U55C(100%)	1303	9024	2016	960
CCLO	12.1%	1.6%	5.7%	0
TCP POE	19.8%	0	10.6%	0
RDMA POE	13.0%	0	5.3%	0
DLRM FC1	278.1%	580.1%	186.3%	798.3%
DLRM FC2	29.6%	85.1%	34.2%	97.9%
DLRM FC3	6.2%	16.1%	2.2%	20.8%

on the CPU after partitioning, whereas the entire matrix did not fit in caches during single-node execution. The reduction time itself is higher in most cases due to an additional copy required to move data between Eigen result buffers and ACCL+ buffers, which can be eliminated with further optimization. Overall, distributing work with ACCL+ achieves lower latency, especially for specific configurations of FC size and number of ranks.

Distributed FPGA-based DLRM. We distribute an industrial DLRM model, as in Table 3, with ACCL+ on 10 U55C FPGAs following the same design principle as shown in Figure 16. The communication between the embedding node and the reduce slave node during each inference requires the transmission of a 3.2 KB partial embedding vector and a 4 KB partial result. Additionally, the reduction process spanning nodes 5 to 9 operates with a message size of 8 KB per inference. The achieved operating frequency is 115 MHz. We utilize 32-bit fixed-point precision for computation. All the application kernels utilize streaming collective APIs to interact with ACCL+. ACCL+ DLRM is configured with the TCP backend from XRT. Though the communication latency could be further optimized with ACCL+ RDMA, it is not on the critical path of overall latency as it is overlapped with computation. We also compare with CPU implementation [58], where the DLRM inference is run on an Intel Xeon Platinum 8259CL CPU @ 2.50 GHz (32 vCPU, Cascade Lake, SIMD supported) and 256 GB DRAM with TensorFlow Serving enabled. Figure 18(a) shows the latency comparison between ACCL+ and the CPU baseline. We evaluate various batch sizes on the CPU. On the other hand, ACCL+ works with streaming data without batching. The hardware implementation demonstrates two orders of magnitude lower latency compared to the CPU. This substantial latency reduction in the hardware implementation is attributed to the parallel arithmetic units in hardware and the significant latency introduced by random memory accesses. Figure 18(b) shows the throughput comparison. ACCL+ shows more than an order of magnitude higher throughput compared to CPU baseline.

6.3 Resource Consumption

The resource utilization of ACCL+ components and the overall utilization of DLRM across nodes are summarized in Table 4. In the ACCL+ subsystem, the majority of resources are allocated to POEs, with the TCP POE being the most resource-intensive, while the CCLO engine utilizes comparatively fewer LUT and memory resources. DLRM utilization is categorized by different layers, and the presented utilization values represent the sum across multiple FPGAs after decomposition. Note that DLRM $FC1$ utilization exceeds 100%, reflecting the decomposition across 8 FPGAs (max 800%). The primary resource bottlenecks for DLRM are URAM, serving as fast on-chip memory for storing small embedding tables, and DSP, essential for matrix computations.

7 Discussion

In this paper we have explored the design of ACCL+ targeting efficient and high-speed offload of MPI-like collective operations. However, due to its flexible and portable design, ACCL+ can be utilized in various applications and scenarios beyond the demonstrated use cases. This section explores how ACCL+ can be extended for a broader range of users applications.

Integrating ACCL+ with Machine Learning Frameworks. While in HPC it is commonplace to develop distributed applications utilizing MPI collectives explicitly, in the field of Machine Learning, codes are often written by Data Scientists who reason about distributed execution in high-level terms such as data, model, or expert parallelism [20]. Integrating ACCL+ into popular machine learning frameworks like TensorFlow [1] and PyTorch [78] is therefore essential to enable its use in ML. Our ongoing work focuses on integrating ACCL+ into PyTorch's Distributed Data Parallel (DDP) [82] module. DDP supports various communication backends for collective operations, which are invoked automatically by the PyTorch execution orchestrator to distribute work to a cluster. We aim to add ACCL+ as a new communication backend to PyTorch DDP, enabling the use of FPGA-based smartNICs to enhance collective operations in AI training and inference.

Additionally, we plan to extend ACCL+ support to other machine learning frameworks.

ACCL+ for Streaming Applications. ACCL+ can also be used for distributed applications that do not require bulk synchronous parallel collective communications, such as streaming applications. In such a scenario, one could use ACCL+ as a transport layer for model-parallel, multi-FPGA streaming accelerators, e.g., Elastic-DF [3]. ACCL+ has existing streaming primitives and collectives which could be utilized for this purpose, as demonstrated in the implementation of the DLRM in Section 6. A more flexible transport based on ACCL+ would, for example, enable higher flexibility in partitioning DNNs to multiple FPGAs.

Implementing Other Distributed Programming Models with ACCL+. The shared memory (SHMEM) programming model [17] is gaining in popularity as it becomes evident that it enables finer-grained overlap between compute and communication on GPU-accelerated systems [50]. SHMEM libraries include MPI-like collectives but add asynchronous one-sided operations (put/get) and signals. These additional operations could be implemented easily into ACCL+ with minimal firmware modifications and no hardware recompilation. Utilizing ACCL+ could reduce the latency of one-sided SHMEM operations, especially where these are used to implement complex communication sequences such as halo exchanges in stencil computations.

8 Conclusion

In this paper, we introduce ACCL+, an open-source FPGA-based collective library designed for portability across diverse platforms and communication protocols. ACCL+ offers flexibility in implementing collectives without the need for FPGA re-synthesis and demonstrates high performance as collective abstractions for FPGA-distributed applications and as a collective offload engine for CPU applications. With ACCL+, there is potential for exploring new possibilities by extending collectives across CPU and FPGA boundaries and orchestrating them for a unified application.

Acknowledgements

We would like to thank AMD for the donation of the HACC FPGA cluster at ETH Zurich on which the experiments were conducted. We thank our shepherd Ming Liu and the anonymous reviewers for their helpful feedback.

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[2] Mohammadreza Alimadadi, Hieu Mai, Shenghsun Cho, Michael Ferdman, Peter Milder, and Shuai Mu. Waverunner: An Elegant Approach to Hardware Acceleration of State Machine Replication. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX Association, 2023.

[3] Tobias Alonso, Lucian Petrica, Mario Ruiz, Jakoba Petri-Koenig, Yaman Umuroglu, Ioannis Stamelos, Elias Koromilas, Michaela Blott, and Kees Vissers. Elastic-DF: Scaling Performance of DNN Inference in FPGA Clouds through Automatic Partitioning. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 2021.

[4] Catalina Alvarez, Zhenhao He, Gustavo Alonso, and Ankit Singla. Specializing the Network for Scatter-Gather Workloads. In *Proceedings of the 11th ACM Symposium on Cloud Computing (SoCC)*. Association for Computing Machinery, 2020.

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

[6] Omer Arap, Lucas R.B. Brasilino, Ezra Kissel, Alexander Shroyer, and Martin Swany. Offloading Collective Operations to Programmable Logic. *Journal of IEEE Micro*, 2017.

[7] Omer Arap, Geoffrey M. Brown, Bryce Himebaugh, and D. Martin Swany. Software Defined Multicasting for MPI Collective Operation Offloading with the NetFPGA. In *European Conference on Parallel Processing*, 2014.

[8] Omer Arap and Martin Swany. Offloading Collective Operations to Programmable Logic on a Zynq Cluster. In *2016 IEEE 24th Annual Symposium on High-Performance Interconnects (HOTI)*, 2016.

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

[10] Mikhail Asiatici, Nithin George, Kizheppatt Vipin, Suhaib A. Fahmy, and Paolo Ienne. Virtualized Execution Runtime for FPGA Accelerators in the Cloud. *Journal of IEEE Access*, 2017.

[11] Mohammadreza Bayatpour, Nick Sarkauskas, Hari Subramoni, Jahanzeb Maqbool Hashmi, and Dhabaleswar K. Panda. BluesMPI: Efficient MPI Non-blocking Alltoall Offloading Designs on Modern BlueField Smart NICs. In Bradford L. Chamberlain, Ana-Lucia Varbanescu, Hatem Ltaief, and Piotr Luszczek,

editors, *High Performance Computing*, Cham, 2021. Springer International Publishing.

[12] Saman Biookaghazadeh, Pravin Kumar Ravi, and Ming Zhao. Toward Multi-FPGA Acceleration of the Neural Networks. *ACM Journal on Emerging Technologies in Computing Systems*, 2021.

[13] Christophe Bobda, Joel Mandebi Mbongue, Paul Chow, Mohammad Ewais, Naif Tarafdar, Juan Camilo Vega, Ken Eguro, Dirk Koch, Suranga Handagala, Miriam Leeser, Martin Herbordt, Hafsah Shahzad, Peter Hofste, Burkhard Ringlein, Jakub Szefer, Ahmed Sanaullah, and Russell Tessier. The Future of FPGA Acceleration in Datacenters and the Cloud. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 2022.

[14] Junehyuk Boo, Yujin Chung, Eunjin Baek, Seongmin Na, Changsu Kim, and Jangwoo Kim. F4T: A Fast and Flexible FPGA-Based Full-Stack TCP Acceleration Framework. In *Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA)*. Association for Computing Machinery, 2023.

[15] Irina Calciu, M. Talha Imran, Ivan Puddu, Sanidhya Kashyap, Hasan Al Maruf, Onur Mutlu, and Aasheesh Kolli. Rethinking Software Runtimes for Disaggregated Memory. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Association for Computing Machinery, 2021.

[16] Adrian M. Caulfield, Eric S. Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, Daniel Lo, Todd Massengill, Kalin Ovtcharov, Michael Papamichael, Lisa Woods, Sitaram Lanka, Derek Chiou, and Doug Burger. A cloud-scale acceleration architecture. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016.

[17] Barbara Chapman, Tony Curtis, Swaroop Pophale, Stephen Poole, Jeff Kuehn, Chuck Koelbel, and Lauren Smith. Introducing OpenSHMEM: SHMEM for the PGAS community. In *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*, 2010.

[18] Fei Chen, Yi Shan, Yu Zhang, Yu Wang, Hubertus Franke, Xiaotao Chang, and Kun Wang. Enabling FPGAs in the Cloud. In *Proceedings of the 11th ACM Conference on Computing Frontiers (CF)*. Association for Computing Machinery, 2014.

[19] Eric Chung, Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Adrian Caulfield, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Maleen Abeydeera, Logan Adams, Hari Angepat, Christian Boehn, Derek Chiou, Oren Firestein, Alessandro Forin, Kang Su Gatlin, Mahdi Ghandi, Stephen Heil, Kyle Holohan, Ahmad El Husseini, Tamas Juhasz, Kara Kagi, Ratna K. Kovvuri, Sitaram Lanka, Friedel van Megen, Dima Mukhortov, Prerak Patel, Brandon Perez, Amanda Rapsang, Steven Reinhardt, Bita Rouhani, Adam Sapek, Raja Seera, Sangeetha Shekar, Balaji Sridharan, Gabriel Weisz, Lisa Woods, Phillip Yi Xiao, Dan Zhang, Ritchie Zhao, and Doug Burger. Serving DNNs in Real Time at Datacenter Scale with Project Brainwave. *Journal of IEEE Micro*, 2018.

[20] Colossal.AI. Paradigms of Parallelism. https://colossalai.org/docs/concepts/paradigms_of_parallelism.

[21] Jason Cong, Bin Liu, Stephen Neuendorffer, Juanjo Noguera, Kees Vissers, and Zhiru Zhang. High-Level Synthesis for FPGAs: From Prototyping to Deployment. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2011.

[22] Nicholas Contini, Bharath Ramesh, Kaushik Kandadi Suresh, Tu Tran, Ben Michalowicz, Mustafa Abduljabbar, Hari Subramoni, and Dhabaleswar Panda. Enabling Reconfigurable HPC through MPI-Based Inter-FPGA Communication. In *Proceedings of the 37th International Conference on Supercomputing (ICS)*. Association for Computing Machinery, 2023.

[23] Paul Covington, Jay Adams, and Emre Sargin. Deep neural networks for youtube recommendations. In *Proceedings of the 10th ACM Conference on Recommender Systems*, 2016.

[24] Abhishek Das, David Nguyen, Joseph Zambreno, Gokhan Memik, and Alok Choudhary. An FPGA-Based Network Intrusion Detection Architecture. *IEEE Transactions on Information Forensics and Security*, 2008.

[25] Juan Miguel de Haro, Rubén Cano, Carlos Álvarez, Daniel Jiménez-González, Xavier Martorell, Eduard Ayguadé, Jesús Labarta, Francois Abel, Burkhard Ringlein, and Beat Weiss. OmpSs@cloudFPGA: An FPGA Task-Based Programming Model with Message Passing. In *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2022.

[26] Tiziano De Matteis, Johannes de Fine Licht, Jakub Beránek, and Torsten Hoefler. Streaming Message

Interface: High-Performance Distributed Memory Programming on Reconfigurable Hardware. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. Association for Computing Machinery, 2019.

[27] Li Ding, Ping Kang, Wenbo Yin, and Linli Wang. Hardware TCP Offload Engine based on 10-Gbps Ethernet for low-latency network communication. In *2016 International Conference on Field-Programmable Technology (FPT)*, 2016.

[28] Haggai Eran, Maxim Fudim, Gabi Malka, Gal Shalom, Noam Cohen, Amit Hermony, Dotan Levi, Liran Liss, and Mark Silberstein. FlexDriver: A Network Driver for Your Accelerator. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Association for Computing Machinery, 2022.

[29] Haggai Eran, Lior Zeno, Maroun Tork, Gabi Malka, and Mark Silberstein. NICA: An Infrastructure for Inline Acceleration of Network Applications. In *2019 USENIX Annual Technical Conference (ATC)*, Renton, WA, 2019. USENIX Association.

[30] Nariman Eskandari, Naif Tarafdar, Daniel Ly-Ma, and Paul Chow. A Modular Heterogeneous Stack for Deploying FPGAs and CPUs in the Data Center. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)*. Association for Computing Machinery, 2019.

[31] Yuanwei Fang, Chen Zou, and Andrew A. Chien. Accelerating Raw Data Analysis with the ACCORDA Software and Hardware Architecture. *Proceedings of the Very Large Data Base Endowment (VLDB)*, 2019.

[32] Clément Farabet, Cyril Poulet, and Yann LeCun. An FPGA-based stream processor for embedded real-time vision with Convolutional Networks. In *2009 IEEE 12th International Conference on Computer Vision Workshops, ICCV Workshops*, 2009.

[33] Kermin Fleming, Hsin-Jung Yang, Michael Adler, and Joel Emer. The LEAP FPGA operating system. In *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, 2014.

[34] Kermin Elliott Fleming, Michael Adler, Michael Pellauer, Angshuman Parashar, Arvind Mithal, and Joel Emer. Leveraging Latency-Insensitivity to Ease Multiple FPGA Design. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)*. Association for Computing Machinery, 2012.

[35] Alex Forencich, Alex C. Snoeren, George Porter, and George Papen. Corundum: An Open-Source 100-Gbps Nic. In *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2020.

[36] Shanyuan Gao, Andrew G. Schmidt, and Ron Sass. Hardware implementation of MPI_Barrier on an FPGA cluster. In *2009 International Conference on Field Programmable Logic and Applications (FPL)*, 2009.

[37] Carlos A Gomez-Uribe and Neil Hunt. The netflix recommender system: Algorithms, business value, and innovation. *ACM Transactions on Management Information Systems (TMIS)*, 2015.

[38] Richard L. Graham, Timothy S. Woodall, and Jeffrey M. Squyres. Open MPI: A Flexible High Performance MPI. In *Proceedings of the 6th International Conference on Parallel Processing and Applied Mathematics*. Springer-Verlag, 2005.

[39] Gaël Guennebaud, Benoit Jacob, et al. Eigen. *URl: http://eigen. tuxfamily. org*, 2010.

[40] Anqi Guo, Tong Geng, Yongan Zhang, Pouya Haghi, Chunshu Wu, Cheng Tan, Yingyan Lin, Ang Li, and Martin Herbordt. A Framework for Neural Network Inference on FPGA-Centric SmartNICs. In *2022 32nd International Conference on Field-Programmable Logic and Applications (FPL)*, 2022.

[41] Zhiyuan Guo, Yizhou Shan, Xuhao Luo, Yutong Huang, and Yiying Zhang. Clio: A Hardware-Software Co-Designed Disaggregated Memory System. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Association for Computing Machinery, 2022.

[42] Udit Gupta, Samuel Hsia, Vikram Saraph, Xiaodong Wang, Brandon Reagen, Gu-Yeon Wei, Hsien-Hsin S Lee, David Brooks, and Carole-Jean Wu. DeepRecSys: A System for Optimizing End-To-End At-scale Neural Recommendation Inference. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020.

[43] Udit Gupta, Carole-Jean Wu, Xiaodong Wang, Maxim Naumov, Brandon Reagen, David Brooks, Bradford Cottel, Kim Hazelwood, Mark Hempstead, Bill Jia, et al. The architectural implications of Facebook's DNN-based personalized recommendation. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2020.

[44] Pouya Haghi, Anqi Guo, Tong Geng, Justin Broaddus, Derek Schafer, Anthony Skjellum, and Martin Herbordt. A Reconfigurable Compute-in-the-Network FPGA Assistant for High-Level Collective Support with Distributed Matrix Multiply Case Study. In *2020 International Conference on Field-Programmable Technology (ICFPT)*, 2020.

[45] Zhenhao He, Dario Korolija, and Gustavo Alonso. EasyNet: 100 Gbps Network for HLS. In *2021 31st International Conference on Field-Programmable Logic and Applications (FPL)*, 2021.

[46] Zhenhao He, Daniele Parravicini, Lucian Petrica, Kenneth O'Brien, Gustavo Alonso, and Michaela Blott. ACCL: FPGA-Accelerated Collectives over 100 Gbps TCP-IP. In *2021 IEEE/ACM International Workshop on Heterogeneous High-performance Reconfigurable Computing (H2RC)*, 2021.

[47] Fernando Luis Herrmann, Guilherme Perin, Josue Paulo Jose de Freitas, Rafael Bertagnolli, and Joao Baptista dos Santos Martins. A Gigabit UDP/IP network stack in FPGA. In *2009 16th IEEE International Conference on Electronics, Circuits and Systems - (ICECS 2009)*, 2009.

[48] Pieter Hintjens. *ZeroMQ: messaging for many applications*. "O'Reilly Media, Inc.", 2013.

[49] Samuel Hsia, Udit Gupta, Mark Wilkening, Carole-Jean Wu, Gu-Yeon Wei, and David Brooks. Cross-Stack Workload Characterization of Deep Recommendation Systems. In *2020 IEEE International Symposium on Workload Characterization (IISWC)*, 2020.

[50] Chung-Hsing Hsu and Neena Imam. Assessment of nvshmem for high performance computing. *International Journal of Networking and Computing*, 2021.

[51] Changho Hwang, KyoungSoo Park, Ran Shu, Xinyuan Qu, Peng Cheng, and Yongqiang Xiong. ARK: GPU-driven Code Execution for Distributed Deep Learning. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2023.

[52] Ranggi Hwang, Taehun Kim, Youngeun Kwon, and Minsoo Rhu. Centaur: A Chiplet-based, Hybrid Sparse-Dense Accelerator for Personalized Recommendations. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020.

[53] Intel. Intel FPGA Add-on for oneAPI Base Toolkit. https://www.intel.com/content/www/us/en/developer/tools/oneapi/fpga.html.

[54] Intel. Intel Quartus Prime Standard Edition User Guide: Getting Started. https://www.intel.com/content/www/us/en/programmable/documentation/yoq1529444104707.html.

[55] Houxiang Ji, Mark Mansi, Yan Sun, Yifan Yuan, Jinghan Huang, Reese Kuper, Michael M. Swift, and Nam Sung Kim. STYX: Exploiting SmartNIC Capability to Reduce Datacenter Memory Tax. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. USENIX Association, 2023.

[56] Yong Ji and Qing-Sheng Hu. 40Gbps multi-connection TCP/IP offload engine. In *2011 International Conference on Wireless Communications and Signal Processing (WCSP)*, 2011.

[57] Wenqi Jiang, Zhenhao He, Shuai Zhang, Thomas B Preußer, Kai Zeng, Liang Feng, Jiansong Zhang, Tongxuan Liu, Yong Li, Jingren Zhou, et al. MicroRec: Efficient Recommendation Inference by Hardware and Data Structure Solutions. In *2021 4th Conference on Machine Learning and Systems (MLSys)*, 2021.

[58] Wenqi Jiang, Zhenhao He, Shuai Zhang, Kai Zeng, Liang Feng, Jiansong Zhang, Tongxuan Liu, Yong Li, Jingren Zhou, Ce Zhang, and Gustavo Alonso. Fleetrec: Large-scale recommendation inference on hybrid gpu-fpga clusters. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining (KDD)*. Association for Computing Machinery, 2021.

[59] Vinod Kathail. Xilinx Vitis Unified Software Platform. In *The 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2020.

[60] Ahmed Khawaja, Joshua Landgraf, Rohith Prakash, Michael Wei, Eric Schkufza, and Christopher J. Rossbach. Sharing, Protection, and Compatibility for Reconfigurable Fabric with AmorphOS. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, 2018.

[61] Dario Korolija, Dimitrios Koutsoukos, Kimberly Keeton, Konstantin Taranov, Dejan Milojičić, and Gustavo Alonso. Farview: Disaggregated memory with operator off-loading for database engines. In *12th Annual Conference on Innovative Data Systems Research Proceedings (CIDR)*, 2022.

[62] Dario Korolija, Timothy Roscoe, and Gustavo Alonso. Do OS abstractions make sense on FPGAs? In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 2020.

[63] Youngeun Kwon, Yunjae Lee, and Minsoo Rhu. TensorDIMM: A Practical Near-Memory Processing Architecture for Embeddings and Tensor Operations in Deep Learning. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2019.

[64] Joshua Lant, Emmanouil Skordalakis, Kyriakos Paraskevas, William B. Toms, Mikel Luján, and John Goodacre. DiAD – Distributed Acceleration for Datacenter FPGAs. In *2023 33rd International Conference on Field-Programmable Logic and Applications (FPL)*, 2023.

[65] Bojie Li, Zhenyuan Ruan, Wencong Xiao, Yuanwei Lu, Yongqiang Xiong, Andrew Putnam, Enhong Chen, and Lintao Zhang. KV-Direct: High-Performance In-Memory Key-Value Store with Programmable NIC. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)*, 2017.

[66] Junnan Li, Zhigang Sun, Jinli Yan, Xiangrui Yang, Yue Jiang, and Wei Quan. DrawerPipe: A Reconfigurable Pipeline for Network Processing on FPGA-Based SmartNIC. *Electronics*, 2020.

[67] Jiaxin Lin, Kiran Patel, Brent E. Stephens, Anirudh Sivaraman, and Aditya Akella. PANIC: A High-Performance Programmable NIC for Multi-Tenant Networks. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation (OSDI)*. USENIX Association, 2020.

[68] Junyi Liu, Aleksandar Dragojević, Shane Fleming, Antonios Katsarakis, Dario Korolija, Igor Zablotchi, Ho-Cheung Ng, Anuj Kalia, and Miguel Castro. Honeycomb: ordered key-value store acceleration on an FPGA-based SmartNIC. *IEEE Transactions on Computers (TC)*, 2023.

[69] Yuanwei Lu, Guo Chen, Bojie Li, Kun Tan, Yongqiang Xiong, Peng Cheng, Jiansong Zhang, Enhong Chen, and Thomas Moscibroda. Multi-Path Transport for RDMA in Datacenters. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, Renton, WA, 2018. USENIX Association.

[70] Jiacheng Ma, Gefei Zuo, Kevin Loughlin, Xiaohe Cheng, Yanqiang Liu, Abel Mulugeta Eneyew, Zhengwei Qi, and Baris Kasikci. A Hypervisor for Shared-Memory FPGA Platforms. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Association for Computing Machinery, 2020.

[71] Chinmay Mahajan, Ashwin Krishnan, Manoj Nambiar, and Rekha Singhal. Hetero-Rec: Optimal Deployment of Embeddings for High-Speed Recommendations. In *Proceedings of the Second International Conference on AI-ML Systems (AIMLSystems)*, New York, NY, USA, 2023. Association for Computing Machinery.

[72] Joel Mbongue, Festus Hategekimana, Danielle Tchuinkou Kwadjo, David Andrews, and Christophe Bobda. FPGAVirt: A Novel Virtualization Framework for FPGAs in the Cloud. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, 2018.

[73] Kenji Mizutani, Hiroshi Yamaguchi, Yutaka Urino, and Michihiro Koibuchi. OPTWEB: A Lightweight Fully Connected Inter-FPGA Network for Efficient Collectives. *IEEE Transactions on Computers (TC)*, 2021.

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

[75] Neal Oliver, Rahul R. Sharma, Stephen Chang, Bhushan Chitlur, Elkin Garcia, Joseph Grecco, Aaron Grier, Nelson Ijih, Yaping Liu, Pratik Marolia, Henry Mitchel, Suchit Subhaschandra, Arthur Sheiman, Tim Whisonant, and Prabhat Gupta. A Reconfigurable Computing System Based on a Cache-Coherent Fabric. In *2011 International Conference on Reconfigurable Computing and FPGAs*, 2011.

[76] Dhabaleswar Kumar Panda, Hari Subramoni, Ching-Hsiang Chu, and Mohammadreza Bayatpour. The MVAPICH project: Transforming research into high-performance MPI library for HPC community. *Journal of Computational Science*, 2021.

[77] Jongse Park, Hardik Sharma, Divya Mahajan, Joon Kyung Kim, Preston Olds, and Hadi Esmaeilzadeh. Scale-out Acceleration for Machine Learning. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. Association for Computing Machinery, 2017.

[78] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. PyTorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*, 2019.

[79] Suchita Pati, Shaizeen Aga, Mahzabeen Islam, Nuwan Jayasena, and Matthew D Sinclair. T3: Transparent Tracking & Triggering for Fine-grained Overlap of Compute & Collectives. *arXiv preprint arXiv:2401.16677*, 2024.

[80] Sreeram Potluri, Hao Wang, Devendar Bureddy, Ashish Kumar Singh, Carlos Rosales, and Dhabaleswar 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.

[81] Andrew Putnam, Adrian Caulfield, Eric Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, Eric Peterson, Aaron Smith, Jason Thong, Phillip Yi Xiao, Doug Burger, Jim Larus, Gopi Prashanth Gopal, and Simon Pope. A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture (ISCA)*. IEEE Press, 2014.

[82] PyTorch. DistributedDataParallel. https://pytorch.org/docs/stable/generated/torch.nn.parallel.DistributedDataParallel.html.

[83] Masudul Hassan Quraishi, Erfan Bank Tavakoli, and Fengbo Ren. A Survey of System Architectures and Techniques for FPGA Virtualization. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 2021.

[84] Burkhard Ringlein, Francois Abel, Alexander Ditter, Beat Weiss, Christoph Hagleitner, and Dietmar Fey. ZRLMPI: A Unified Programming Model for Reconfigurable Heterogeneous Computing Clusters. In *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*.

[85] Burkhard Ringlein, Francois Abel, Alexander Ditter, Beat Weiss, Christoph Hagleitner, and Dietmar Fey. Programming Reconfigurable Heterogeneous Computing Clusters Using MPI With Transpilation. In *2020 IEEE/ACM International Workshop on Heterogeneous High-performance Reconfigurable Computing (H2RC)*, 2020.

[86] Burkhard Ringlein, François Abel, Dionysios Diamantopoulos, Beat Weiss, Christoph Hagleitner, Marc Reichenbach, and Dietmar Fey. A Case for Function-as-a-Service with Disaggregated FPGAs. In *2021 IEEE 14th International Conference on Cloud Computing (CLOUD)*, 2021.

[87] Mario Ruiz, David Sidler, Gustavo Sutter, Gustavo Alonso, and Sergio López-Buedo. Limago: An FPGA-Based Open-Source 100 GbE TCP/IP Stack. In *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*, 2019.

[88] Manuel Saldaña, Arun Patel, Christopher Madill, Daniel Nunes, Danyao Wang, Paul Chow, Ralph Wittig, Henry Styles, and Andrew Putnam. MPI as a Programming Model for High-Performance Reconfigurable Computers. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 2010.

[89] Manuel Saldana and Paul Chow. TMD-MPI: An MPI Implementation for Multiple Processors Across Multiple FPGAs. In *2006 International Conference on Field Programmable Logic and Applications (FPL)*, 2006.

[90] Martin D. Schatz, Robert A. van de Geijn, and Jack Poulson. Parallel Matrix Multiplication: A Systematic Journey. *SIAM Journal on Scientific Computing*, 2016.

[91] Niklas Schelten, Fritjof Steinert, Anton Schulte, and Benno Stabernack. A High-Throughput, Resource-Efficient Implementation of the RoCEv2 Remote DMA Protocol for Network-Attached Hardware Accelerators. In *2020 International Conference on Field-Programmable Technology (ICFPT)*, 2020.

[92] Junnan Shan, Mihai T. Lazarescu, Jordi Cortadella, Luciano Lavagno, and Mario R. Casu. CNN-on-AWS: Efficient Allocation of Multikernel Applications on Multi-FPGA Platforms. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2021.

[93] Yizhou Shan, Will Lin, Zhiyuan Guo, and Yiying Zhang. Towards a Fully Disaggregated and Programmable Data Center. In *Proceedings of the 13th ACM SIGOPS Asia-Pacific Workshop on Systems (APSys)*. Association for Computing Machinery, 2022.

[94] David Sidler, Zsolt István, and Gustavo Alonso. Low-latency TCP/IP stack for data center applications. In *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, 2016.

[95] David Sidler, Zeke Wang, Monica Chiosa, Amit Kulkarni, and Gustavo Alonso. StRoM: Smart Remote Memory. In *Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys)*. Association for Computing Machinery, 2020.

[96] Kaushik Kandadi Suresh, Benjamin Michalowicz, Bharath Ramesh, Nick Contini, Jinghan Yao, Shulei Xu, Aamir Shafi, Hari Subramoni, and Dhabaleswar Panda. A Novel Framework for Efficient Offloading of Communication Operations to Bluefield SmartNICs. In *2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2023.

[97] Naif Tarafdar, Nariman Eskandari, Varun Sharma, Charles Lo, and Paul Chow. Galapagos: A Full Stack

Approach to FPGA Integration in the Cloud. *Journal of IEEE Micro*, 2018.

[98] Maroun Tork, Lina Maudlej, and Mark Silberstein. Lynx: A SmartNIC-Driven Accelerator-Centric Architecture for Network Servers. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Association for Computing Machinery, 2020.

[99] Anuj Vaishnav, Khoa Dang Pham, and Dirk Koch. A Survey on FPGA Virtualization. In *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, 2018.

[100] Anuj Vaishnav, Khoa Dang Pham, Dirk Koch, and James Garside. Resource Elastic Virtualization for FPGAs Using OpenCL. In *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, 2018.

[101] John Paul Walters, Xiandong Meng, Vipin Chaudhary, Tim Oliver, Leow Yuan Yeow, Bertil Schmidt, Darran Nathan, and Joseph Landman. MPI-HMMER-boost: distributed FPGA acceleration. *The Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology*, 2007.

[102] Hao Wang, Sreeram Potluri, Devendar Bureddy, Carlos Rosales, and Dhabaleswar K. Panda. GPU-Aware MPI on RDMA-Enabled Clusters: Design, Implementation and Evaluation. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 2014.

[103] Zeke Wang, Hongjing Huang, Jie Zhang, Fei Wu, and Gustavo Alonso. FpgaNIC: An FPGA-based Versatile 100Gb SmartNIC for GPUs. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. USENIX Association, 2022.

[104] Jagath Weerasinghe, Francois Abel, Christoph Hagleitner, and Andreas Herkersdorf. Enabling FPGAs in Hyperscale Data Centers. In *2015 IEEE 12th Intl Conf on Ubiquitous Intelligence and Computing and 2015 IEEE 12th Intl Conf on Autonomic and Trusted Computing and 2015 IEEE 15th Intl Conf on Scalable Computing and Communications and Its Associated Workshops (UIC-ATC-ScalCom)*, 2015.

[105] Jagath Weerasinghe, Raphael Polig, Francois Abel, and Christoph Hagleitner. Network-attached FPGAs for data center applications. In *2016 International Conference on Field-Programmable Technology (FPT)*, 2016.

[106] Gabriel Weisz, Joseph Melber, Yu Wang, Kermin Fleming, Eriko Nurvitadhi, and James C. Hoe. A Study of Pointer-Chasing Performance on Shared-Memory Processor-FPGA Systems. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*. Association for Computing Machinery, 2016.

[107] AMD Xilinx. XUP Vitis Network Example (VNx). https://github.com/Xilinx/xup_vitis_network_example.

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

[109] AMD Xilinx. AXI DataMover v5.1 LogiCORE IP Product Guide. https://docs.xilinx.com/r/en-US/pg022_axi_datamover/AXI-DataMover-v5.1-LogiCORE-IP-Product-Guide, 2023.

[110] AMD Xilinx. DMA for PCI Express (PCIe) Subsystem. https://www.xilinx.com/products/intellectual-property/pcie-dma.html, 2023.

[111] Chao-Tung Yang, Chih-Lin Huang, and Cheng-Fang Lin. Hybrid CUDA, OpenMP, and MPI parallel programming on multicore GPU clusters. *Computer Physics Communications*, 2011.

[112] Yue Zha and Jing Li. Virtualizing FPGAs in the Cloud. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Association for Computing Machinery, 2020.

[113] Yue Zha and Jing Li. Hetero-ViTAL: A Virtualization Stack for Heterogeneous FPGA Clusters. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, 2021.

[114] Yue Zha and Jing Li. When Application-Specific ISA Meets FPGAs: A Multi-Layer Virtualization Framework for Heterogeneous Cloud FPGAs. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Association for Computing Machinery, 2021.

[115] Jie Zhang, Hongjing Huang, Lingjun Zhu, Shu Ma, Dazhong Rong, Yijun Hou, Mo Sun, Chaojie Gu, Peng Cheng, Chao Shi, and Zeke Wang. SmartDS: Middle-Tier-Centric SmartNIC Enabling Application-Aware Message Split for Disaggregated Block Storage. In *Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA)*. Association for Computing Machinery, 2023.

[116] Wentai Zhang, Jiaxi Zhang, Minghua Shen, Guojie Luo, and Nong Xiao. An Efficient Mapping Approach to Large-Scale DNNs on Multi-FPGA Architectures. In *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2019.

[117] Guorui Zhou, Xiaoqiang Zhu, Chenru Song, Ying Fan, Han Zhu, Xiao Ma, Yanghui Yan, Junqi Jin, Han Li, and Kun Gai. Deep interest network for click-through rate prediction. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (KDD)*, 2018.

[118] Yu Zhu, Zhenhao He, Wenqi Jiang, Kai Zeng, Jingren Zhou, and Gustavo Alonso. Distributed Recommendation Inference on FPGA Clusters. In *2021 31th International Conference on Field-Programmable Logic and Applications (FPL)*. IEEE, 2021.

[119] Noa Zilberman, Yury Audzevich, G. Adam Covington, and Andrew W. Moore. NetFPGA SUME: Toward 100 Gbps as Research Commodity. *Journal of IEEE Micro*, 2014.

Appendices

A ACCL+ Initialization

ACCL+ is specifically designed to deliver a high-speed collective communication solution tailored for FPGAs, or to function as a specialized NIC for CPUs. To simplify the initialization process, we choose not to generalize the network stack in the hardware for general-purpose communication. Instead, we utilize the conventional NIC in the CPU system for launching ACCL+ applications in a distributed environment, such as through *mpirun*, or for establishing RDMA queue pairs for ACCL+ communicators. This NIC only involves a lower-speed connection to other ranks.

Other than the *collective API*, the CCL driver also exposes a *housekeeping API* which enables CCLO configuration and monitoring, and a *primitive API* consisting of simple data movement operations (send, receive, copy). Listing 3 illustrates the three APIs - the code initializes ACCL+, invokes the ACCL+ send/receive primitives to exchange data between ranks 0 and 1, and executes an reduce collective on all ranks.

In this example, we utilize the MPI library to determine the local rank ID (lines 6-8) when the application has been launched with mpirun. Then ACCL+ is initialized by calling the constructor function and passing the Coyote device object (line 11). Similar approach is applied for Vitis device object. Within the constructor, it also allocates and configures a set of CCLO-managed Rx buffers for collective operations in the FPGA memory, e.g., for the eager protocol. The code then constructs the communicator according to rank information and protocol type (line 15). If the protocol is TCP, the code will issue commands to open connections between each rank in the communicator via the protocol offload engine. If the protocol is RDMA, the code utilizes the commodity NIC to change queue pair information. The TCP connections and the RDMA queue pairs are generalized to session IDs in the communicator. All configuration information is offloaded to the FPGA so that the CCLO can rapidly access them. Just like MPI, ACCL+ can be configured with multiple communicators of different sizes. While not pictured here for brevity, each ACCL+ collective can specify the communicator it operates on, with COMM_WORLD being the default.

Lines 21-25 implement data movement from the buffer of rank 1 to the buffer of rank 0 utilizing the primitives API. Lines 27 execute collectives on the entire communicator using the collectives API.

```
1  #include "accl.hpp"
2  #include <mpi.h>
3  using namespace ACCL;

5  int main(int argc, char **argv) {
6      int mpi_rank;
7      MPI_Init(&argc, &argv);
8      MPI_Comm_rank(MPI_COMM_WORLD, &mpi_rank);

10     CoyoteDevice* device = new CoyoteDevice();
11     ACCL* accl = new ACCL(device);

13     std::map<int, std::string> ranks_dict = /*
           Populate rank vector*/;
14     Protocol protocol = TCP; // or RDMA
15     accl->configure_communicator(ranks_dict, mpi_rank,
           protocol);

17     const int bufsize = 64;
18     auto opbuf = accl->create_buffer<int>(bufsize);
19     auto resbuf = accl->create_buffer<int>(bufsize);

21     if (mpi_rank == 0) {
22         accl->send(opbuf, bufsize, 1); // Send to rank 1
23     } else if (mpi_rank == 1) {
24         accl->receive(opbuf, bufsize, 0); // Receive
               from rank 0
25     }

27     accl->reduce(opbuf, resbuf, bufsize, 0); // Root
           rank 0

29     opbuf->free_buffer();
30     resbuf->free_buffer();
31     delete accl;
32     delete device;
33     MPI_Finalize();
34 }
```

Listing 3: Initialization and invocation of collectives from CPU.