Direct Universal Access: Making Data Center Resources Available to FPGA

Ran Shu[1], Peng Cheng[1], Guo Chen[1,2], Zhiyuan Guo[1,3], Lei Qu[1], Yongqiang Xiong[1], Derek Chiou[4], and Thomas Moscibroda[4]

Microsoft Research[1], Hunan University[2], Beihang University[3], Microsoft Azure[4]

Abstract

FPGAs have been deployed at massive scale in data centers. Using currently available communication architectures, however, it is difficult for FPGAs to access and utilize the various heterogenous resources available in data centers (DRAM, CPU, GPU,...). In this paper, we present Direct Universal Access (DUA), a communication architecture that provides uniform access for FPGA to these data center resources.

Without being limited by machine boundaries, DUA provides global names and a common interface for communicating across various resources, the underlying network automatically routing traffic and managing resource multiplexing. Our benchmarks show that DUA provides simple and fair-share resource access with small logic area overhead ($<10\%$) and negligible latency ($<0.2\mu$s). We also build two practical multi-FPGA applications—deep crossing and regular expression matching—on top of DUA to demonstrate its usability and efficiency.

1 Introduction

Large-scale FPGA deployments in data centers [1–9] has changed the way of FPGA-based distributed systems are designed. Instead of a small number of FPGAs and limited resources (*e.g.*, only the DRAM on each FPGA board), modern FPGA applications can use heterogeneous computation/memory resources, such as CPU, GPU, host/onboard DRAM, SSD *etc.*, across large-scale data centers. The scale and diversity of resources enables the building of novel FPGA-based applications, such as cloud-scale web search ranking [10,11], storage systems [4,6], or deep learning platforms [12].

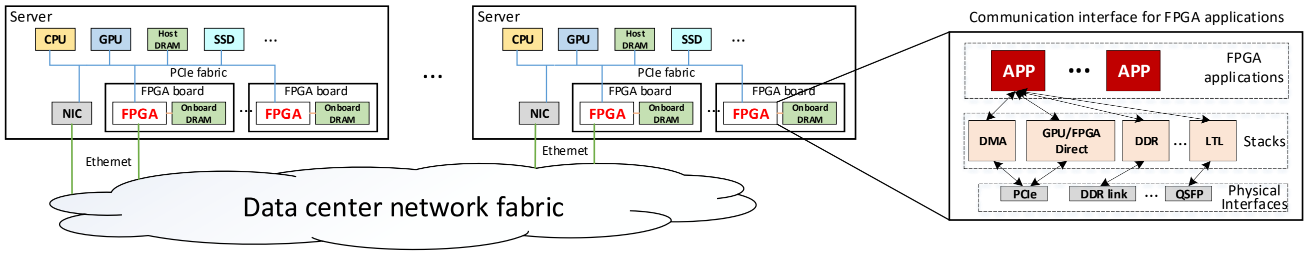
Building large-scale and diverse FPGA applications requires communication capabilities between any pair of FPGAs and other components in the data center. However, with today's technology such *FPGA communication* is highly impractical and cumbersome, posing severe challenges to designers and application developers. There are three main problems barring FPGA applications to conveniently and efficiently use data center resources (see Fig. 1(a)):

First, different resources at different locations (local/remote) are connected in different ways (*e.g.*, PCIe, network) requiring different communication stacks. This greatly increases programming complexity. For example, an FPGA application may use a custom communication stack [2] to access a local (same server) FPGA, a networking stack [11] to access a remote (different server) FPGA, GPU/FPGA Direct [13] to access a GPU, DMA to access system DRAM, DDR IP to access local DRAM, *etc.* Each of these communication stacks has a different interface (different I/O ports, functional timings, *etc.*), making it hard to understand, program, optimize, and debug.

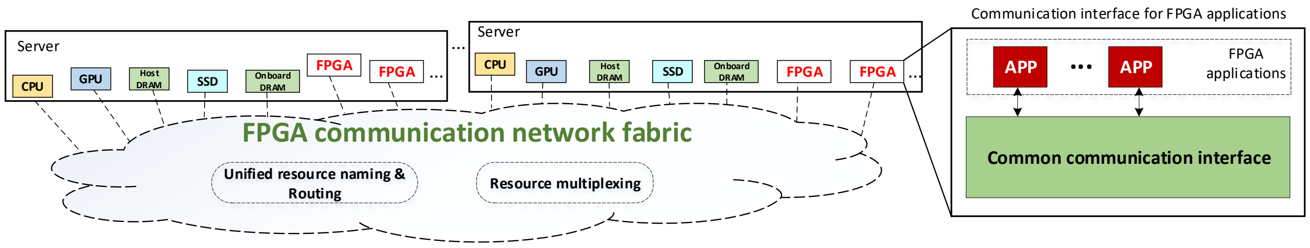
Second, most resources (*e.g.*, host DRAM, SSD) in a data center are organized in a server-centric manner. Each resource uses a dedicated name space that can only be accessed from within a host (*e.g.*, a PCIe address.) The lack of global names for resources is inefficient for FPGAs when accessing remote resources, since they first need to communicate with the remote host, and the host first has to perform the access on behalf of the requesting FPGA. If an FPGA wants to write a remote SSD, for example, it first has to transfer the data to a daemon process running on its local CPU, which passes the data to the remote CPU, which then finally writes the data to the targeted SSD. To make matters worse, developers manually write dedicated logic for each type of FPGA-to-resource communication.

Third, although FPGAs have been deployed at data center scale, current FPGA communication does not deal well with resource multiplexing. Though various resources are accessed through the same physical interface (*e.g.*, DMA and GPU/FPGA Direct both through PCIe), we are not aware of any general resource multiplexing scheme. FPGA developers need to manually handle each specific case, which is tightly coupled with the application logic. Moreover, problems become more severe when there are multiple FPGA applications using the same resource (*e.g.*, applications on two FPGAs accessing the same SSD).

Instead, FPGA developers would like an FPGA commu-



(a) Current FPGA communication architecture.


Server
CPU
GPU
Host DRAM
SSD
Onboard DRAM
FPGA
FPGA communication network fabric
Unified resource naming & Routing
Resource multiplexing
Communication interface for FPGA applications
APP
FPGA applications
Common communication interface

(b) Ideal FPGA communication architecture.

Figure 1: Comparison between a current FPGA communication architecture and an ideal communication architecture that enables FPGAs to easily access data center resources.

nication architecture as shown in Fig. 1(b). This architecture has the following desirable properties: 1) a *common communication interface* regardless of what the communication endpoints are and where they reside; 2) a *global, unified naming scheme* that can address and access all resources regardless of their location; 3) an *underlying network service* that provides routing and resource multiplexing. With such a communication architecture, FPGA applications could easily access a diverse set of resources across the data center, using the common programming interface with each resource's global name. Indeed, such a communication architecture is what developers and architects expect and is used in other distributed systems. For example, such a design has been proven successful in IP networks.

In this paper, we propose Direct Universal Access (DUA) to bring this desirable communication architecture to the FPGA world. Doing so is challenging in multiple ways, especially considering that very little, if anything, can be changed when we seek real-world deployment in existing data centers. It is impractical to require all manufacturers to support a new unified communication architecture. To circumvent this challenge, DUA chooses to abstract an overlay network on top of the existing communication stacks and physical interconnections, thereby providing a unified FPGA communication method for accessing all resources. Moreover, performance and area is often crucial in FPGA-based applications, so DUA was designed to mimimize performance and area overheads. Inspired by ideas in software defined networking [14, 15], we architect DUA into a *data plane* that is included in every FPGA, and a hybrid *control plane* including both CPU and FPGA control agents. Needless to say, designing and implementing this novel communication architecture also brings about numerous specific *technical* challenges, design choices and implementation problems. We discuss these challenges alongside our solutions in Sections 4, 5 , 6 and 7.

In summary, we make the following contributions in this paper. We introduce DUA, a communication architecture with unified naming and common interface to enable large-scale FPGA applications in a cloud environment. We implement and deploy DUA on a twenty FPGA testbed. Our testbed benchmarks show that DUA introduces negligible latency ($<0.2\mu$s) and small area ($<10\%$) overhead on each FPGA. Moreover, we demonstrate that using DUA, it is easy to build highly-efficient production-quality FPGA applications. Specifically, we implement two real-world multi-FPGA applications: Deep Crossing and regular expression matching, and show the vastly superior performance of these applications based on DUA. In addition, we also design and implement a new communication stack that supports high-performance communication between local FPGAs through PCIe, which can be integrated into DUA data plane as a highly efficient underlying stack.

2 Background

2.1 FPGA Deployments in Data Centers

The left side of Fig. 1(a) shows an overview of current FPGA data center deployments. FPGA boards connect to their host server motherboard through commodity communication interfaces such as PCIe. Each hosting server can contain one [11] or more FPGA boards [2]. Each FPGA board is typically equipped with gigabytes of onboard DRAM [2, 9, 11]. Recent deployments [11] directly connect each FPGA to the data center networking fabric, enabling it to send and receive packets without involving its host server. To meet the high data rate of physical inter-

Table 1: FPGA programming efforts to connect different communication stacks.

Resource	Communication stack	LoC
Host DRAM	DMA	294
Host CPU	FPGA host stack	205
Onboard DRAM	DDR	517
Remote FPGA	LTL	1356

faces, FPGAs use Hard IPs (HIPs) to handle physical layer protocols for each interface. Above these HIPs, FPGAs provide communication stacks that abstract access to different resources. Communication stacks may share the same HIP, e.g. DMA [16] and GPU [13] stacks both need to use the PCIe fabric. Although a server may contain multiple boards connected through PCIe [2], there are no PCIe-based stacks that support efficient and direct communication between FPGAs.

Data center FPGAs often contain a *shell* [10, 11] that contains modules that are common for all applications. For example, shells typically include communication stacks for accessing various resources (*e.g.*, PCIe, DMA, Ethernet MAC). In this way, developers only need to write their FPGA application logic and connect using these communication interfaces. The FPGA shell is similar to an operating system in the software world.

FPGAs in data centers are widely used to accelerate different applications. Applications like deep neural networks [17, 18] and bioinformatics [19] have high demand on communications between FPGAs. FPGAs for web search ranking applications [10, 11] rapidly exchange data with host and other FPGAs to generate the ranking as quick as possible. Key-value store acceleration [20] requires FPGAs to access remote FPGA's on board DRAM or even remote servers host memory. Big data analytics [21] not only require rapid coordination between computation nodes, but also need to directly fetch data from database [4, 5]. The demand of high throughput and extra low latency require FPGAs to access heterogeneous resources directly which challenges the design of FPGA communication architecture in data centers.

2.2 Existing Problems

Current FPGA communication architecture pose multiple severe problems:

Complex FPGA Application Interface: FPGA-based systems are hard to develop and deploy. One of the major reasons is that communication interfaces are hard to implement. Interfacing requires significant programming expertise and effort by application developers. To make things worse, existing stack interfaces are highly implementation specific, with substantial incompatibilities and differences between different vendors. This makes building the communication system of the application alone a major undertaking (*e.g.*, KV-Direct [20]).

To convey a concrete sense of the programming difficulties involved, consider a simple FPGA application that uses different communication interfaces to access four different resources: host DRAM, host CPU, on board DRAM, and remote FPGA. Table 1 shows the lines of Verilog code for application developers to connect each stack's interface.

Poor Resource Accessibility: Although data centers provide many computation/memory resources that potentially could be used by FPGA applications, most of these resources (even the homogeneous ones) are only named in server-local name space and work with their own software device driver/stack. There is no unified naming scheme for accessing remote resources. Without unified naming, most PCIe-based resources (*e.g.*, DRAM, SSD, GPU) can only be accessed within a server's PCIe domain, making it difficult for remote FPGAs to use. Even with latest technology like RDMA that FPGAs can use to access specific remote resources, the software driver/stack is still needed for remote communication, impacting performance.

Fixed Network Routing: In current communication architectures, FPGA applications can only access resources through limited and fixed paths. In [2], for example, FPGAs communicate with other local FPGAs through the dedicated PCIe fabric and can not access remote resources through networking. In [11, 22], FPGAs are directly connected to Ethernet through top-of-rack (ToR) switches, i.e., a pair of FPGAs can only communicate through Ethernet even when they are in the same PCIe domain. Both of these examples cannot make full use of all available bandwidth.

Also, such fixed communication architectures limit the system's scalability. For example, deploying large FPGA applications via a network-based communication architecture increases the port density of ToR switches and is a challenge to data center networking, even if most FPGAs are used for compute-intensive tasks and need only little networking bandwidth.

Poor Resource Multiplexing: To support accessing data center resources as a pool, resource multiplexing is one of the key considerations of an FPGA communication architecture. Current architectures do not handle stack multiplexing well. For example, if two applications both access local host DRAM through DMA, they need to collaboratively write a DMA multiplexer and demultiplexer. From our experience, even a simple multiplexer/demultiplexer requires 354 lines of HDL code. Moreover, currently there is no general physical interface multiplexing scheme, and it is therefore hard for current FPGA applications to simultaneously access local host DRAM and local SSD without modifying the underlying shell, since these two resources are both connected through the PCIe bus.

The Elastic Router proposed in [11] tries to solve the multiplexing problem in an FPGA environment. Currently, however, it only addresses the problem of multiplexing between multiple applications which use a common networking stack, without handling multiplexing between other stacks and between physical interfaces. Later we will see that DUA extends this with a general resource multiplexing scheme.

Inefficient Communication Stack: Existing communication architectures implement resource accessing for FPGAs in an indirect and inefficient way. Typically, FPGA applications use DMA to access local resources, which results in significant latency and low bandwidth. We note that while [23] provides a direct FPGA-to-FPGA communication mechanism through PCIe, it is inefficient. Specifically, the receiver FPGA acts as host DMA driver and first issues an DMA request. The sender FPGA treats the request as a normal DMA request and sends data. After sending data they need to synchronize the queue pointers. Since a data transmission crosses the PCIe fabric 3 times, it wastes bandwidth and has higher than necessary latency.

Furthermore, an FPGA can only access remote host DRAM by relaying data between the two sides' CPUs, reducing performance and consuming cores. We measured the performance of doing so in our testbed (see §8). Specifically, we ran two daemon processes on the local and remote server's CPU that relayed data between the local FPGA and remote DRAM through a TCP socket. Results show that for writing 256B data to the remote DRAM, the average end-to-end latency is $\sim$51.4μs. The tail latency is in the milliseconds. Using remote DMA instead of TCP may improve the performance, but in our measurement the average latency is still $\sim$20μs due to the CPU involvement (*e.g.*, initiate request, interrupt). We note that in such an application it is possible to leverage remote FPGA as a data relay for accessing remote host DRAM, through direct communication between FPGAs (*e.g.*, using LTL [11]).

3 Desired Communication Architecture

To overcome the problems outlined in the previous section, we design DUA using a familiar and concrete model: *Global names* and a *common communication interface* for FPGAs and resources regardless their location, where the underlying network automatically *routes* communication traffic according to the global name and manages the *resource multiplexing* with full utilization of *existing stacks*.

This communication architecture supports pooling data center resources for FPGAs to access. Specifically, FPGA applications can access any resource in data center using a global name and a common programming interface. The network provides a globally unified name for various kinds of resources. When getting a message, the network either routes the access to the targeted resource if it is available, or notifies the application if the resource is not available, automatically without the application being involved. Also, there is no need for applications to implement multiplexing between communication stacks or physical interconnections. DUA utilizes underlying communication when appropriate. The network automatically manages sharing and contention according to the desired policy (*e.g.*, fair sharing or priority scheduling).

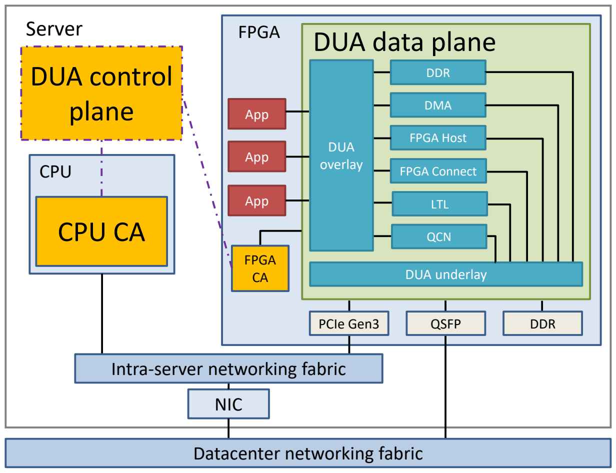


Figure 2: DUA architecture.

Networked systems communicate in exactly this way. In computer networking systems, programmers use IP addresses with TCP/UDP ports as a global name to identify a communication endpoint, and access them using a unified BSD socket interface. The network (both networking stack and fabric) automatically route the traffic through the paths calculated from routing protocols. The network also deals with resource multiplexing through techniques such as congestion control and flow control.

Of course, the communication architecture must also carefully consider security mechanisms, such that the universal access of FPGAs to resources within the data center does not damage or crash other hardware/software systems. Since FPGA-based applications often have high performance requirements, performance and resource overhead of the unified communication method must be kept low.

4 DUA Overview

Overall, DUA abstracts an overlay network for FPGA on top of existing data center network fabric. DUA provides a new communication architecture that has the desired properties mentioned before, which makes data center resources a shared pool for FPGA.

In detail, we provide an communication architecture with such overlay network, including the common communication interface and the naming scheme suitable for various applications to access different resources, and the routing, multiplexing, and resource management scheme correspondingly provided by the network.

Fig. 2 shows the system architecture of DUA. Specifically, DUA consists of a low-cost hardware *data plane* residing in every FPGA's shell, and a hybrid *control plane* including both CPU and FPGA control agents.

The DUA control plane is the brain of the overlay network. It manages all resources, assigning addresses and calculating the routing paths to them, and manages the multiplexing of resources and the network. DUA supports both connection-based and connectionless communication. The connection

UID (serverID:deviceID)	Address /port	Resource description
192.168.0.2:1	0x00000001CFFFF000	1st block of host DRAM
192.168.0.2:1	0x000000019FFFF000	2nd block of host DRAM
192.168.0.2:2	0x80000000	1st block of FPGA onboard
192.168.0.2:3	8000	1st application on FPGA
192.168.0.2:3	8001	2nd application on FPGA

Figure 3: Example resources address on server 192.168.0.2.

setup and close are processed and managed in the DUA control plane.

The DUA data plane is the actual executer of FPGA communication traffic. It stays between the FPGA applications and physical interfaces. The data plane can efficiently reuse existing communication stacks, as well as support new stacks, providing the same communication interface for various applications to access different resources.

5 DUA Communication Interface

We first describe the unified communication interface provided by DUA for accessing various resources.

5.1 Resource Address Format

DUA provides each resource a unified address, which is globally unique in the data center. Devising a totally new address format is not a wise option, since it would require both a complicated system for managing data-center-scale address spaces and changes to the existing network fabric to use that new address format. Instead, DUA leverages the current naming schemes of various resources, and combines them into a new hierarchical name.

Specifically, DUA assigns each device a unique name (UID) that extends the IP address into the intra-server network. A UID consists of two fields, *serverID:deviceID:resourceINST*. *serverID* is a globally unique ID for each server. In an Ethernet-based data center network, we leverage the server IP address as *serverID*, which a is already uniquely assigned by the network fabric. *deviceID* is a unique ID for each resource within the server (e.g. FPGA on-board DRAM, GPU, NVMe and etc.), which is newly assigned by DUA. In our current implementation, UID is designed to be 48 bits in total (32b *serverID* (length of IPv4 address in current data centers) and 16b *deviceID*).

Within each device, DUA leverages the existing addressing scheme of each resource. For example, it can be the memory address if the targeted resource is host/onboard memory, or the port number if the targeted resource is a remote FPGA application. Fig. 3 provides some examples of different resources' addresses in DUA. In §6.1 and §6.2 we will describe how it is easy to manage addresses and do routing using such an UID format.

5.2 API

DUA supports both connection-based and connectionless communication. Connectionless communication is less efficient than connection-based communication because it facilitates management issues (*e.g.*, access control) and network resource multiplexing (*e.g.*, multiplex underlying stack tunnels for the messages that have common routing paths). More details of routing and connection management will be discussed in §6.2 and §6.3.

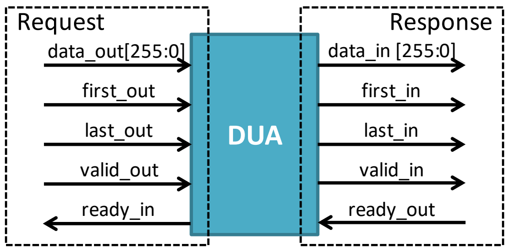


Figure 4: I/O interface for DUA.

5.2.1 Semantic Primitives

For each FPGA communication connection, applications generate communication primitives similar to BSD socket. There are two types of primitives:

1. **Connection setup/close primitives**: These primitives include CONNECT/LISTEN/ACCEPT and CLOSE, which are used by applications to setup and close a connection, respectively.

2. **Data transmission primitives**: These primitives include SEND/RECV and WRITE/READ. SEND/RECV are for sending/receiving data to/from computation resources such as other FPGA applications and CPU processes, which works as a FIFO between the two sides. Additionally, DUA supports message-based communication by adding a PUSH flag in each DUA message header. WRITE/READ are one-sided primitives for write/read data to/from memory/storage resources, which are different from WRITE/READ in BSD sockets.

5.2.2 I/O interface

We implement DUA in both Verilog and OpenCL. OpenCL is a high-level programming language for FPGAs (and GPUs.). DUA implemented in OpenCL can provide socket-like interface. However, it cost much more FPGA logic and degrade performance. Thus we only use the Verilog implementation and add a wrapper on top of it to support OpenCL. See Appendices A and B for a sample usage of the DUA interface.

Fig. 4 shows the physical I/O interface of DUA, which is full-duplex. The *request interface* is for applications to issue primitives (§5.2). The *response interface* is for applications to get primitive responses (completion information or response data).

The DUA I/O interface is the same in both direction. For each DUA message, the message header and payload use the same data wires for transmission. Note that the data signal has only 256 bits. Although a wider interface could increase the amount of data transmitted per hardware cycle, it would increase the switch fabric logic complexity and thus decrease the available clock frequency of corresponding modules. Consequently, a DUA message may be transmitted in several cycles. The first 1 or 2 cycles (depends on the message type) of data bus is message header, followed by the payload. The first/last bit indicate the first and last cycle of a message transmission. For each cycle with data to be sent or received, the valid signal is set. Note that valid can only be raised when the receiver side set the ready signal.

Next in §6 and §7, we introduce the design and implementation of control and data plane.

6 DUA Control Plane

The DUA control plane manages all resources, routing and connections. Since FPGAs are not suitable for implementing complicated control logic, we put the main control logic in the CPU control agent (CPU CA). We also implement a control agent in the FPGA (FPGA CA) which monitors local resources on its board and delivers control plane commands to the data plane.

6.1 Resource Management

Logically, the entire DUA control plane maintains the information of all available resources in data center, and assigns each resource with a UID. Thanks to the hierarchical address format (§5.1), each DUA control agent only needs to handle its local resources.

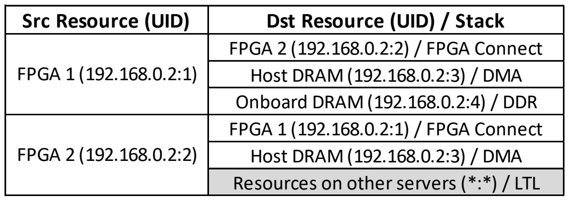
Specifically, each FPGA CA monitors all available resources on its FPGA board (*e.g.*, onboard DRAM, FPGA application). The FPGA CA does not assign addresses for those resources. Instead, each FPGA CA uploads its local resource information to the CPU CA in its host server, and the CPU CA assigns the UIDs all together. The host CPU CA gathers the resource information from all FPGA CAs, as well as all other resources such as host memory and GPU in this server.

It is straightforward to assign UIDs to local resources. As mentioned, the first UID field, *serverID*, is the server IP. Then CPU CA assigns a unique *deviceID* for each resource within this server. The CPU CA maintains the mapping from *deviceID* to different local resources, and updates the mapping once when are any resource changes, plug-in/plug-out or failures. DUA does not manage the address within each resource instead letting each device control it.

Currently, DUA does not provide a naming service. Applications directly use UID to identify resources without a name resolution service. The design of a naming service is future work.

6.2 Routing Management

To offer routing capabilities, the DUA control plane calculates the routing paths. The data plane the forwards the traffic to the target resource (directly or through other FPGAs' data plane) fully transparent to applications.



Src Resource (UID)	Dst Resource (UID) / Stack
FPGA 1 (192.168.0.2:1)	FPGA 2 (192.168.0.2:2) / FPGA Connect
	Host DRAM (192.168.0.2:3) / DMA
	Onboard DRAM (192.168.0.2:4) / DDR
FPGA 2 (192.168.0.2:2)	FPGA 1 (192.168.0.2:1) / FPGA Connect
	Host DRAM (192.168.0.2:3) / DMA
	Resources on other servers (*:*) / LTL

Figure 5: Example interconnection table on server 192.168.0.2.

Designing and implementing data center scale routing is challenging [24]. Benefiting from the hierarchical UID format, we leverage existing data center network routing capabilities. Each DUA control agent only needs to maintain interconnection information and calculate routing paths within each server.

Specifically, each CPU CA independently maintains an interconnection table for all local resources, as shown in Fig. 5. The interconnection table records the neighborhood information between FPGAs or FPGA and other resources. The first column records a source FPGA, and the second column records the local/remote resources that can be directly accessed from this FPGA through which underlying communication stack.

The interconnection table's information is updated as follows. Besides the resource information, each FPGA CA uploads the information about its communication stacks and physical interfaces to the CPU CA in its own server. Based on the uploaded information, CPU CA determines the interconnection between different FPGAs and updates the interconnection table. If an FPGA reports that it has connectivity through the data center networking fabric, the CPU CA will insert an entry for this FPGA, withan entry for each legal destination to any resources on other servers (the last row of Fig. 5).

According to the interconnection table, it is easy to calculate a routing path to targeted resources. Specifically, if an FPGA wants to communicate with some resource, DUA first checks the *serverID* and *deviceID* field in the destination resource UID, to see if this resource has a direct connection from this FPGA. If yes, DUA uses the stack recorded in the interconnection table to access the resource. If not, DUA looks up the interconnection table to find a routing path through other FPGAs.

For example, in Fig. 5, if FPGA 1 (UID *192.168.0.2:4*) wants to communicate with a remote application on FPGA 3 located on another server (say, UID *192.168.11.5:3*), the calculated routing path is from FPGA 1 to FPGA 2 via FPGA Connect, and then to FPGA 3 via LTL.

6.3 Connection Management

In DUA, every FPGA communication is abstracted as a connection. A connection is uniquely identified by a <*src UID*:*dst UID*> pair. The DUA control plane is in charge of managing all connections.

At the connection setup phase, to ensure security, DUA first checks the access control policy to see if the source FPGA application is allowed to access the destination resource. If so, the CPU CA will check the *dst UID* with the interconnection table to calculate the routing path (§6.2) and then delivers the forwarding table to the FPGA data planes along the routing path, so the data plane will forward the application traffic to the right stack. Depending on the type of routing path, CPU CA will deliver different actions to the data plane and underlying stacks. Specifically:

1) If the destination resource is directly connected, CPU CA simply delivers the corresponding forwarding table to the data plane.2) If the destination resource is not directly connected, but still within the same server, CPU CA calls the stacks in the local FPGAs along the routing path to setup a stack connection. For example in Fig. 5, if FPGA 2 initiates a connection to access the onboard DRAM of FPGA 1, CPU CA first sets up an FPGA Connect connection between FPGA 2 and FPGA 1. 3) If the destination resource is on a different server, CPU CA first calls the remote CPU CA to collaboratively setup a connection tunnel between the two remote FPGAs (*e.g.*, LTL connection). If necessary, CPU CA also sets up stack tunnels between each sides' local FPGAs.

If the above procedures all succeed, the DUA connection is established and the application is notified. Also, the active DUA connection is maintained in the control plane. Note that some underlying stacks do not support a large number of concurrent connections (*e.g.*, LTL currently only supports 64). For multiple DUA connections with common routing paths, DUA supports connections multiplexing the same tunnel connection (*e.g.*, two DUA connections share an LTL tunnel connection) to solve this problem. Moreover, DUA sets up multiple tunnels for each traffic class to simplify traffic scheduling.

When an application closes a connection, the DUA control plane closes the stack tunnel connections along the path (if no one is multiplexing them), and deletes the corresponding forwarding tables in data plane. If any failures of the data path (*e.g.*, targeted resource, physical interface, communication stack) is detected, the control plane immediately disconnects all affected DUA connections, and notifies the application.

7 DUA Data Plane

As shown in Fig. 2, the DUA data plane resides between FPGA applications and the physical interfaces. It consists of three components: *overlay*, *stack*, and *underlay*. DUA overlay acts as a router, transferring data between different applications and communication stacks. Below the overlay, DUA leverages all existing (or future new) stacks to efficiently access target resources. DUA underlay connects between the stacks and physical interfaces, which provides efficient multiplexing on physical interfaces for different stacks.

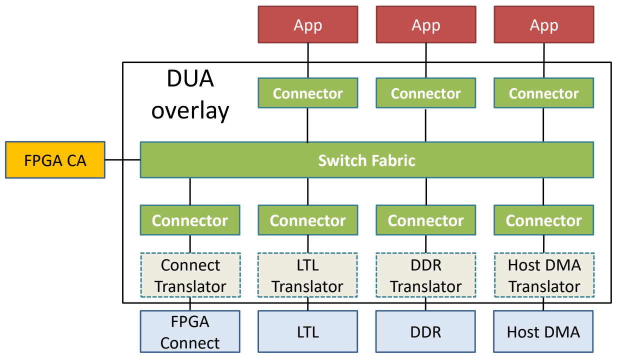
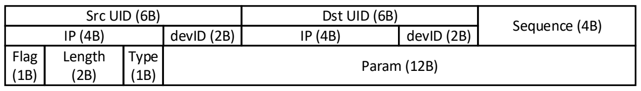


Figure 6: DUA overlay components.



Src UID (6B)		Dst UID (6B)		Sequence (4B)
IP (4B)	devID (2B)	IP (4B)	devID (2B)	

Flag (1B)	Length (2B)	Type (1B)	Param (12B)

Figure 7: DUA message header format.

7.1 DUA Overlay

To efficiently transfer data between multiple different applications and stacks, we use an "Internet-router-like" architecture to implement the DUA overlay module. Specifically, there are three components inside the overlay, *connector*, *switch fabric* and *stack translator*, as shown in Fig. 6.

7.1.1 Connector

Connectors reside between application/stack and switch fabric, playing a role similar to line cards in Internet routers. Specifically, connector performs the following tasks:

1) Translating data (from application or stack) from/to I/O interface to/from DUA messages: The I/O interface described in §5.2 is actually implemented in connectors, receiving data both from applications or stacks. A DUA message is the data transmission unit inside the overlay (like IP packets). Its header format is shown in Fig. 7. Connector encapsulates data into DUA messages in cut-through mode with the corresponding header fields filled. Also, when connecter receives a DUA message from the switch fabric, it translates it back to the I/O interface signals. One thing to note is that for connection setup/close primitives passed from the I/O interface, connector encapsulates a special message and passes it to the FPGA control agent, notifying the control plane to setup/close the connection.

2) Maintaining and looking up the forwarding table: The forwarding table stores the mapping of destination UID to the switch output port. After message encapsulation, the connector needs to lookup the forwarding table to determine the switch output port to forward the message to the destination connector through the switch fabric. The forwarding table is computed by the control plane and delivered to DUA connector (§6.2). To eliminate the contention between connectors during forwarding table lookup, each connector only maintains its own forwarding table and performs lookups independently. Note that only entries for active connections and permitted connectionless message routes are delivered to the data plane, so this table is not large. In our current im-

plementation, the table can store 32 forwarding entries and the area cost is very low (see §8.1).

3) Access control: Connector is also responsible for security checks whenever there is data coming in. Specifically, for connection-based communications, after a connection has passed the security check and has been successfully setup by the control plane (see §6.3), the control plane adds this connection to the routing control table in the connectors along the path. For connectionless communications, the control plane sets the routing table according to polices that determine which path is allowed for these messages. Only data from these legal connections or paths are transmitted by DUA connectors.

4) Transport Control: DUA adopts an end-to-end transport control design. Thus this feature is only enabled for connectors that attach to applications. We do not reimplementing TCP or RDMA on FPGA, instead, DUA leverages LTL [11] as the transport protocol.

7.1.2 Switch Fabric

The switch fabric performs the same role as its counterpart in Internet routers, switching messages from the incoming connecter to the destination connector. Specifically, DUA overlay adopts a crossbar switch fabric. To minimize memory overhead, we do not buffer any application or stack data in the switch fabric. Instead, we make our switch fabric lossless, and utilize the application data buffer or stack data buffer to store data that is going to be transmitted. Once the output is blocked, the switch fabric will back pressure to the input connector, and unset the *Ready* signal of I/O interface.

In our current implementation, the underlying stacks (LTL, FPGA Connect, DMA, FPGA Host and DDR) are all reliable, as such, the lossless fabric ensures the DUA data transmission primitives (§5.2) are also reliable. Note that in real hardware implementations, although we do not buffer data, in order to achieve full pipelining, we need to cache 256b data (one cycle of data from the I/O interface) at each input port. And to remove head-of-line blocking among different ports, we implement a Virtual Output Queue (VOQ) as in elastic router [11] at each input port of the switch fabric.

7.2 Communication Stacks

In our current implementation, we integrate four existing communication stacks (LTL, DMA, FPGA-Host and DDR) into the DUA data plane using *stack translators*. Note that DUA leverages LTL as its end-to-end transport protocol. LTL here only provides reliable communication in data center network, its end-to-end congestion control protocol is disabled. In addition, we design and implement a new stack called *FPGA Connect* that provides high-performance intra-FPGA communication through PCIe for improving the communication efficiency mentioned in §2.2.

7.2.1 Stack Translators

Stack translators translate between DUA interface (§5.2) and the actual interface of underlying stacks. After control plane sets up the connection, it delivers the corresponding translation tables to the stack translators along the routing path. Translation tables record the mapping of DUA message header and underlying stack header. Whenever receiving data from connector, the translator encapsulates the data into stack interface according to the translation table. If control plane decides to multiplex stack tunnels, stack translator encapsulates multiple DUA connections' data into the same stack connection. On the other end, when receiving data from stacks, translator translates it back into DUA interface and passes it to the connector for further routing.

Taking stack translator for memory stacks DDR/DMA as example, it converts DUA operations to memory operations. For instance, when the stack translator receives data with *Type* READ from DUA connector (*i.e.*, DMA/DDR read initiated by applications), it calls the DMA/DDR stack to issue a read request, with the memory address set accordingly to the address in DUA message header. Also, the DUA message header is stored for sending the READ response back to the application through DUA. After it gets the response, stack translator calls DUA interface to send data back.

Similar to the forwarding table, the translation table also only stores entries for active connections. Currently we implement a table with size for 32 entries.

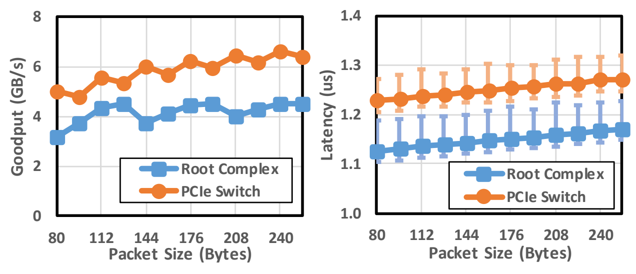
7.2.2 FPGA Connect Stack

FPGA Connect Stack enables direct communication between multiple applications on different FPGAs through PCIe within a single server. Here we introduce the design and implementation details of FPGA Connect Stack.

Challenge: There are three major challenges. *1) Differentiating different inter-FPGA connections:* One naive solution is to use a large number of physical memory addresses to receive packets of each connection. That not only needs a large amount of memory address space but also introduces address management overhead. *2) Head-of-line (HOL) blocking:* PCIe is a lossless fabric and back pressure is adopted. Due to application processing limitations and PCIe bandwidth sharing, the available rates of each connection can be different. Without an explicit flow control, the slower connection will saturate the buffer on both sender HIPs and receiver which will delay other transmissions. *3) Bufferbloat:* If packets are sent to PCIe HIP in a best-effort manner, the buffer inside HIP will quickly fill up which causes further delays.

Design: FPGA Connect provides SEND and RECV operations to users. In order to differentiate different connections and minimize physical memory address waste, FPGA Connect adds a packet header in PCIe packet's payload containing both sender's and receiver's port, and uses one identity single-packet-size [1] physical memory address as receive address for each board.

[1]The packet size mentioned in this paper is the PCIe TLP layer payload size.



(a) Throughput (b) Latency

Figure 8: Performance of FPGA connect stack using different TLP packet sizes.

We provide a simple token-based flow control to avoid HOL blocking and bufferbloat. FPGA connect sends one token request, which is a normal data packet with a special bit, in each RTT to ask the receiver for the available token if it keeps sending. Receiver responds with an ACK that includes the available token for this connection once it receives this request. The sender uses this token to set the size of the sending window for the next RTT. The receiver only keeps the available connection number and assigns the available token based on the algorithm of [25] to keep low buffer in HIPs.

PCIe provides WRITE and READ operation primitives for data transmission. According to our measurements, PCIe peer-to-peer READ throughput is 20%-40% lower than WRITE because of hardware limitations. Therefore, FPGA Connect only uses WRITE as the data transmission primitive for performance reason.

Implementation: In our implementation, FPGA Connect has a 16 bits header including 8 bits destination connection ID, 2 bits type and 6 reserved bits. Packets with *Type = 0x01* are token requests, and those with *Type = 0x11* are ACKs. Other packets are normal data packets. We leverage CPU software for connection setup, release and fail-over.

Evaluation: The typical PCIe network topology is a tree structure. The root node is called root complex which is in the CPU. Devices (e.g. FPGA, GPU and NVMe) are directly connected to it. As the number of PCIe lanes provided by root complex is limited, the number of devices connected to root complex and the peer to-peer bandwidth among devices is limited. Thus, some data center servers use PCIe switch to support more devices and improve peer-to-peer performance which provides high density of heterogeneous computation capacity. Thus we test the performance of FPGA Connect on two platforms. One provides connection through the PCIe switch and the other through the root complex. For our testbed, the maximum packet payload size is 256B. Although the FPGA can send 256B packets, Root complex forces segmentation of packets to align in 64B units (the PCIe switch does not segment packets). We have measured the performance of FPGA Connect with different packet sizes using our testbed described in §8.1. Fig. 8 shows the results. FPGA Connect achieves 6.66 GB/s peak throughput when the packet size is 240B (the peak throughput is limited by TLP implementation issues in our FPGA shell). Consequently, in order to reduce the header overhead and achieving higher throughput, we choose 240B as our maximum packet size. As for latency, FPGA Connect provides latency as low as 1.1∼1.33 us. In our testbed, PCIe switch offers better throughput but slightly higher latency than root complex.

7.3 DUA Underlay

The DUA underlay resides between the stacks and physical interfaces, managing hard IPs and resource sharing among stacks, protecting DUA stacks against outside attacks and avoiding failed stacks sending packets outside the FPGA. All these features are managed by policies configured by the control plane. Each physical interface has a separate underlay module. The upstream and downstream interface are the same to provide seamless integration with stacks. Therefore, existing stacks need no modification when attached to DUA underlay.

The DUA underlay achieves these goals by setting up a virtual transaction layer, which provides multiplexing and security protection without proscribing a stack interface abstraction. The virtual transaction layer works by checking, modifying, and, if necessary, dropping traffic generated by or routed to the stacks to prevent causing a physical interface (or even the whole network) into an error condition.

When data flows into DUA underlay from stacks, all packages are passed through a filter which validates them as well-formed per the rules configured by the control plane. If stack traffic violates any security rules or physical interface restrictions, the packet is dropped and the violation is reported to FPGA CA. Then, when data flows from virtual transaction layer to physical interfaces, the DUA underlay works as a multiplexer, take the responsibility of managing multiple connections for supporting multiple users. DUA underlay scheduling the data to the physical interface using polices like fair-sharing, weighted sharing, strict priority etc. In our implementation, we use fair-sharing. To avoid wasting bandwidth, we implement a shallow input FIFO for each stack in the underlay. The scheduler fairly schedules data from non-empty FIFOs only.

When receiving data from a physical interface, the DUA underlay works as a demultiplexer. It demultiplexes the incoming data to the corresponding stack through virtual transaction layer according to the data header.

8 Evaluation

Testbed Setup: As shown in Fig. 9, we build a testbed consisting of two Supermicro SYS-4028GR-TR2 servers, 20 FPGAs and one Arista 7060X switch. Every 5 FPGAs are inserted under the same PCIe switch and only one FPGA under each PCIe switch is connected to the Arista switch. All FPGAs in the testbed are the same as in [11], which is an Altera Stratix V D5, with 172.6K ALMs of programmable logic, one 4 GB DDR3-1600 DRAM channel, two PCIe Gen 3 x8

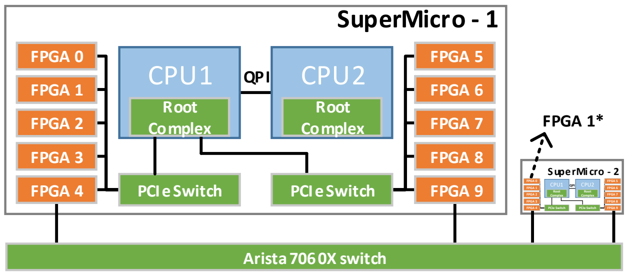


Figure 9: DUA experiment testbed

Component			ALMs	
DUA overlay	Switch fabric	2 ports	1272	0.74%
		4 ports	3227	1.88%
		8 ports	9366	5.45%
	Connector		3011	1.75%
	Stack translator	FPGA Connect	138.4	0.08%
		LTL	255.4	0.15%
		DMA	115.7	0.07%
		DDR	190.3	0.11%
DUA underlay	Stacks: FPGA Connect, LTL, DMA, DDR PHY interfaces: PCIe, DDR, QSFP		431.7	0.25%

Figure 10: FPGA area cost of different components in DUA data plane.

HIPs and two independent 40 Gb Ethernet QSFP+. Note that in all the following experiments, we only enable one HIP and one QSFP+ in each FPGA shell. In addition, because SuperMicro does not provide two PCIe slots directly connected to the root complex, we use a Dell R720 server to test the performance under root complex (experiments in §7.2.2). Other experiments are on the SuperMicro servers. Servers' OS is Windows Server 2012 R2, and each server has a 40Gbps NIC connected to the switch.

8.1 System Micro Benchmark

We first show that DUA only consumes little FPGA area. Then we show that the DUA switch fabric and routing table achieve high throughput and low latency. Finally we show that DUA incurs little latency overhead and handles the multiplexing of communication stacks and applications well.

8.1.1 FPGA Area Cost

Fig. 10 shows the FPGA resource consumption for implementing DUA. Here we only list logic resource overhead (in ALMs) since DUA does not buffer data and BRAM cost is negligible. When connecting four stacks and no application, the total ALMs consumed by DUA overlay (including 4-port switch, 4 connectors and 4 stack translators) is only 9.29%. When increasing the number of switch ports to 8 (4 ports for applications), the overlay still only costs 19.86% logic area in our middle-end FPGA. The underlay consumes only 0.25% logic resources when connecting 4 stacks and 3 physical interfaces. Compared to the logic resources consumed by the existing underlying communication stacks and physical interfaces (in total $>$17%), such overhead incurred by DUA is moderate and acceptable. With more advanced FPGAs, such logic area cost will become negligible (*e.g.*, latest Stratix 10 5500 FPGA has 10x logic resource [26]).

Table 2: Throughput and latency of switch fabric.

Number of ports	Latency (ns)			Throughput (GBps)
	min	avg	max	
4	53	326	1086	9.6
8	56	649	1903	9.6

Table 3: Area cost and max frequency of routing table.

Number of entries	ALMs (per port)		Fmax (MHz)
32	1435	0.83%	490.20
64	2810	1.63%	423.91
128	5571	3.23%	378.93

8.1.2 Switch Fabric Performance

We conduct an experiment to evaluate the performance of the switch fabric in DUA overlay. In this test, all switch ports are attached to an application which acts as traffic generator and result checker. All applications send messages to a single port to construe a congested traffic scenario. Message length is varied from 32B to 4KB. The switch fabric works at 300 MHz, thus the ideal throughput is 9.6 GBps. We measure the latency and output throughput during a test lasting for 2 hours. Table 2 shows the result with different number of switch ports. Throughput achieves the theoretical maximum and Latency is low.

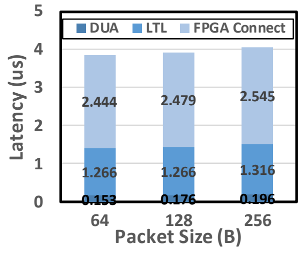
8.1.3 Routing Table Performance

We implement parallel matching engines for each table entry. Each entry comparison takes 1 cycle, and the matching result is calculated by a priority selector in another cycle. Note that this implementation has a two cycles constant latency. On the other hand, the routing table is well pipelined, so in every cycle it can accept a message and look up its output port. Thus, the message-per-second throughput is the same as the clock frequency. Table 3 shows the area cost and max frequency of the routing table with different number of entries. Our implementation with typical 32 entries consumes 0.83% of ALMs, that is 3.3%-6.6% for a typical 4-8 port implementation. The max frequency is high enough for serving the shortest DUA message at a rate of over 10GBps per port. As the number of entries increases to 64 and 128, area cost increases linearly and the frequency dose not decrease much.

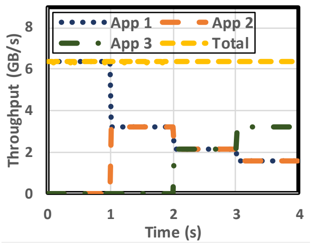
8.1.4 Latency Overhead

We use FPGA 1 to send data through DUA to FPGA 1* in Fig. 9. Specifically, DUA first transmits data from FPGA 1 to FPGA 4 through FPGA Connect, and then to FPGA 4* (the 4th FPGA on Server 2) through LTL, and then to FPGA 1* through FPGA Connect. We measure the end-to-end communication latency including DUA and all the traversing stacks, as well as the break-down latency for each stack.

Fig. 11(a) shows the average latency of each part. Under various packet sizes, DUA only incurs less than 0.2μs



(a) Latency overhead



(b) Handling multiplexing

Figure 11: DUA only adds little latency overhead and handles multiplexing well.

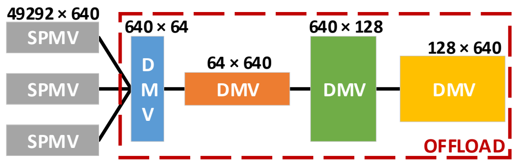


Figure 12: Deep Crossing Model in our experiment

latency, which is a negligible overhead compared with the other stacks' latency in the end-to-end communication. Note that DUA is fully pipelined and can achieve line rate, incurring no throughput overhead.

8.1.5 Handling Multiplexing

We build three applications (App 1,2,3) on the same FPGA all using DUA, to test whether DUA can handle multiplexing well. Specifically, App 1 starts from second 0, keeps writing data to host DRAM. At second 1, App 2 also starts to write data to host DRAM. At second 2, App 3 starts to send data to another local FPGA through FPGA Connect. In this scenario, all three applications multiplex the same PCIe physical interface, App 1 and 2 multiplex the same DMA stack, and DMA stack and FPGA Connect stack multiplex the same physical PCIe interface. DUA adopts fair scheduling policy for all DUA connections, and changes to weighted share scheduling (share ratio 1:1:2) starting from second 3.

Fig. 11(b) shows the results. During the experiments, the total throughput of all applications always achieves the maximum throughput of the PCIe physical interface (§7.2.2). When new applications join, DUA successfully balances the throughput between them. Specifically, App 1 and 2 each achieve ∼3.2GBps between second 1 and 2, all three applications get ∼2.2GBps between second 2 and 3. Also, when we change the scheduling policy to 1:1:2 weighted share at second 3, the three applications quickly get their respective expected throughput.

8.1.6 Deep Crossing

8.2 Applications Built With DUA

In this section, we present two applications built on DUA, demonstrating that DUA can ease the process of building high-performance multi-FPGA applications.

Deep crossing, a deep neural network, was proposed in [27] for handling web-scale application and data sizes. The model trained from the learning process is deployed in Bing

Table 4: ALMs cost (%) of different dense part

	Base	D1	D2	D3	D4	All
Parall = 32	26.0	11.0	9.4	11.0	9.8	67.2
Parall = 64		20.8	19.1	20.8	19.6	**106.3**

Table 5: Latency (μs) of the dense part in FPGA.

	D1	D2	D3	D4	Comm.	E2E
Single FPGA	7.27	6.58	13.30	12.96	N/A	40.12
Two FPGAs	4.17	3.48	7.22	7.09	1.419 (FC)	**23.38**
					3.354 (LTL)	**25.32**

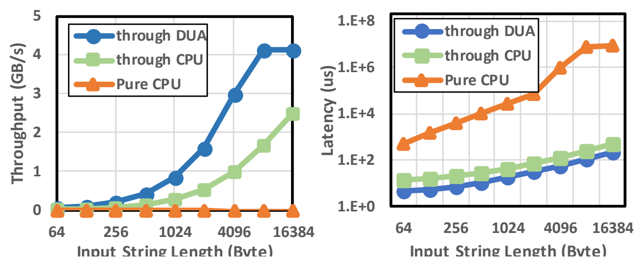
to provide web services. The critical metric for the deep crossing model is latency since it impacts service response time.

Fig. 12 shows the structure of the deep crossing model in our experiments. There are three sparse matrix-vector (SPMV) multiplications and four dense matrix-vector (DMV) multiplications. Each input data to the whole model is a 49,292 dimensional vector, as in [27]. The sparse part is a memory-intensive task while the dense part is computation-intensive, and the vector between each dense part is small. Therefore, we offload the dense parts to FPGA to reduce latency.

We implement all dense parts inside FPGA using OpenCL. In our implementation, for each matrix multiplication, there is an adjustable parameter called parallel degree (Parall), which determines the number of concurrent multiplications being done in one cycle. The larger Parall, the fewer cycles are needed to complete this matrix multiplication; meanwhile, the larger parall, the more FPGA logic resources are consumed. As shown in Tab. 4, if we implement the whole four DMVs in a single FPGA board, we can only offload the model with Parall = 32 because of FPGA resource limitations.

To achieve better latency, we use DUA to build a two-FPGA deep crossing accelerator. Specifically, we implement all the DMVs in the model with Parall = 64, and download the first two DMVs on one FPGA, and the other two DMVs into another FPGA. The two FPGAs are physically connected through both the intra-server PCIe network and Ethernet, with underlying stack FPGA Connect / LTL enabled. We use DUA interface to connect the DMVs logic on the two FPGA boards.

It only incurs 26 extra lines of OpenCL code to call the DUA interface to connect the two FPGA boards. Moreover, changing the communication method only requires changing the routing table, without any change to OpenCL code and thus eliminates hours of hardware recompilation time. Table 5 shows the latency results of the FPGA offloading (counting only the FPGA-related latency). The results show that the two-FPGA version built with DUA reduces the latency by ∼42% (through FPGAConnect, FC for short in table) or ∼37% (through LTL) compared to the single-FPGA version.



(a) Throughput (b) Latency

Figure 13: Performance of multi-regular-expression matching system.

8.2.1 Fast Multi-Regular-Expression Matching

A network intrusion detection system (IDS) [28–31] lies between the network and the host, intercepting and scanning the traffic to identify patterns of malicious behaviour. Typically, these patterns in IDS are expressed as regular expressions. Each pattern is denoted as a *rule*, and all patterns in the different stages together are called the *rule set*.

We have built a fast multi-regular-expression matching prototype which consists of three FPGAs over DUA. The boards are physically connected through PCIe within the same server. Each regular expression is translated into one independent NFA, and the NFA is translated into matching circuit logic using methods in [32]. We use DUA with underlying FPGA Connect stack to transfer data between these three FPGAs. Connecting the DUA interface only costs less than 30 lines of OpenCL code on each FPGA. We implement the whole core rule set [33] of ModSecurity in our system. The rule set contains 219 different regular expression rules in total. We randomly divide these 219 rules into three pattern stages with each stage containing 73 rules, and implement each stage in a single FPGA.[2] For each stage in each FPGA, we implement 32 parallel matching engines, with each engine matching one 8-bit character in one cycle. An input string goes to the next FPGA only after it finishes the matching in this stage.

For comparison, we also let these three FPGAs exchange data through CPU, without the direct communication method provided by DUA FPGA Connect. Also, we compare with the baseline performance, which uses a single 2.3GHz Xeon CPU core and the widely used PCRE [34] regular expression engine.

We generate different length of input strings for matching, to evaluate the throughput and latency of the whole regular expression matching system. String lengths vary from 64 to 16K byte, with contents randomly generated. In our experiment, we use a single CPU core to DMA the input string into the matching system instead from the network. We get the matching results back using DMA and count the performance in the same CPU core.

[2]Note that the number of rules in each FPGA does not affect the matching speed, since all rules are matched in parallel.

Fig. 13 shows the result. Enabled by direct communication through DUA, our regular expression matching system (denoted as "through DUA") achieves about three times higher throughput and lower latency compared to FPGAs exchanging data through CPU (denoted as "through CPU"). Benefitting thus from the direct communication through pure hardware, our system almost reaches the maximum possible throughput of input string DMA when the string length exceeds 8KB. On the contrary, when exchanging data through CPU, the CPU needs to frequently read matching results from former-stage FPGAs and send strings to next-stage FPGAs, which becomes the performance bottleneck. Also, we can see that for such a complex rule set, pure CPU can only achieve very low performance. Note that our throughput is slightly lower than the maximum FPGA Connect speed (§7.2.2) due to the following reasons: 1) software libraries for DMA incur overhead compared with pure physical interface; 2) although the data paths through PCIe are different for DMA input/output data to CPU and exchanging data between FPGAs, they all use the same PCIe HIP to issue operations which incurs some contention.

9 Related Work and Conclusion

While prior work has aimed to provide abstractions and simplified FPGA communications, to the best of our knowledge, DUA is the first unified communication architecture for FPGAs to access all available data center resources, regardless of their location and type. Catapult shell [10, 11], Amazon F1 [2] and its derivatives provide an abstract interface between FPGA logic, software and physical interfaces, but it remains far from being the unified communication architecture provided by DUA. The Altera/Intel Avalon bus [35] and the AXI bus [36] used by Xilinx provide a unified interface for FPGA applications, but they are designed solely for on-chip buses, not the scale of data center network. TMD-MPI [37] provides a general MPI-like communication model for multi-FPGA systems, but it only targets communication between FPGAs rather than general resource access. Also, it is implemented in software and requires the CPU. The recent LTL [11] work targets the communication between FPGAs in data center through Ethernet. DUA can leverage and support all these works as communication stacks to improve connectivity.

Providing a communication architecture with unified naming and common interface has proven widely successful in IP networks. In this paper, DUA takes a first step to bring this communication architecture into the FPGA world. Our experiments show that DUA has negligible impact on performance and area, and greatly eases the programming of distributed FPGA applications that access data center resources. Note that though this work is targeted to FPGAs, there is no reason why it cannot be applied to other devices as well.

References

[1] Microsoft Goes All in for FPGAs to Build Out AI Cloud. https://www.top500.org/news/microsoft-goes-all-in-for-fpgas-to-build-out-cloud-based-ai/.

[2] Amazon EC2 F1 Instances. https://aws.amazon.com/ec2/instance-types/f1/.

[3] Intel, Facebook Accelerate Datacenters With FPGAs. https://www.enterprisetech.com/2016/03/23/intel-facebook-accelerate-datacenters-fpgas/.

[4] Data Engine for NoSQL - IBM Power Systems Edition White Paper. https://www-01.ibm.com/common/ssi/cgi-bin/ssialias?htmlfid=POW03130USEN.

[5] Baidu Takes FPGA Approach to Accelerating SQL at Scale. https://www.nextplatform.com/2016/08/24/baidu-takes-fpga-approach-accelerating-big-sql/.

[6] Jian Ouyang, Shiding Lin, Song Jiang, Zhenyu Hou, Yong Wang, and Yuanzheng Wang. Sdf: Software-defined flash for web-scale internet storage systems. *ACM SIGPLAN Notices*, 49(4):471–484, 2014.

[7] Jian Ouyang, Shiding Lin, Wei Qi, Yong Wang, Bo Yu, and Song Jiang. Sda: Software-defined accelerator for large-scale dnn systems. In *Hot Chips 26 Symposium (HCS), 2014 IEEE*, pages 1–23. IEEE, 2014.

[8] Alibaba, Intel introduce FPGA to the Cloud. https://luxeelectronicscomblog.wordpress.com/2017/03/13/alibaba-intel-introduce-fpga-to-the-cloud/.

[9] Tencent FPGA Cloud Computing. https://www.qcloud.com/product/fpga.

[10] Andrew Putnam, Adrian M Caulfield, Eric S Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, et al. A reconfigurable fabric for accelerating large-scale datacenter services. In *Computer Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium on*, pages 13–24. IEEE, 2014.

[11] Adrian M Caulfield, Eric S Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, et al. A Cloud-Scale Acceleration Architecture. In *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*, pages 1–13. IEEE, 2016.

[12] Microsoft demonstrates the world's 'first AI supercomputer,' using programmable hardware in the cloud. https://www.geekwire.com/2016/microsoft-touts-first-ai-supercomputer-using-programmable-hardware-cloud/.

[13] Ray Bittner, Erik Ruf, and Alessandro Forin. Direct gpu/fpga communication via pci express. *Cluster Computing*, 17(2):339–348, 2014.

[14] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008.

[15] Nick McKeown. Software-defined networking. *INFOCOM keynote talk*, 17(2):30–32, 2009.

[16] Jian Gong, Tao Wang, Jiahua Chen, Haoyang Wu, Fan Ye, Songwu Lu, and Jason Cong. An efficient and flexible host-fpga pcie communication library. In *Field Programmable Logic and Applications (FPL), 2014 24th International Conference on*, pages 1–6. IEEE, 2014.

[17] Eric Chung, Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Adrian Caulfield, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, et al. Serving dnns in real time at datacenter scale with project brainwave. *IEEE Micro*, 38(2):8–20, 2018.

[18] Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Logan Adams, Mahdi Ghandi, et al. A configurable cloud-scale dnn processor for real-time ai. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*, pages 1–14. IEEE Press, 2018.

[19] Michael Johannes Jaspers. Acceleration of read alignment with coherent attached fpga coprocessors. 2015.

[20] 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*. ACM, 2017.

[21] Katayoun Neshatpour, Maria Malik, Mohammad Ali Ghodrat, and Houman Homayoun. Accelerating big data analytics using fpgas. In *Field-Programmable Custom Computing Machines (FCCM), 2015 IEEE 23rd Annual International Symposium on*, pages 164–164. IEEE, 2015.

[22] Jagath Weerasinghe, Francois Abel, Christoph Hagleitner, and Andreas Herkersdorf. Enabling fpgas in hyperscale data centers. In *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 IEEE 12th Intl Conf on*, pages 1078–1086. IEEE, 2015.

[23] Malte Vesper, Dirk Koch, Kizheppatt Vipin, and Suhaib A Fahmy. JetStream: an open-source high-performance PCI express 3 streaming library for FPGA-to-host and FPGA-to-FPGA communication. In *Field Programmable Logic and Applications (FPL), 2016 26th International Conference on*, pages 1–9. IEEE, 2016.

[24] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy M Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, et al. Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network. *SIGCOMM*, 45(4):183–197, 2015.

[25] Jiao Zhang, Fengyuan Ren, Ran Shu, and Peng Cheng. Tfc: token flow control in data center networks. In *Proceedings of the Eleventh European Conference on Computer Systems*, page 23. ACM, 2016.

[26] Stratix 10 - Overview. https://www.altera.com/products/fpga/stratix-series/stratix-10/overview.html.

[27] Ying Shan, T Ryan Hoens, Jian Jiao, Haijing Wang, Dong Yu, and JC Mao. Deep crossing: Web-scale modeling without manually crafted combinatorial features. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 255–262. ACM, 2016.

[28] Snort - Official Site. https://www.snort.org/.

[29] The Bro Network Security Monitor. https://www.bro.org/.

[30] ModSecurity: Open Source Web Application Firewall. https://www.modsecurity.org/.

[31] Application Layer Packet Classifier for Linux. http://l7-filter.sourceforge.net/.

[32] Reetinder P S Sidhu and Viktor K Prasanna. Fast regular expression matching using fpgas. pages 227–238, 2001.

[33] OWASP ModSecurity Core Rule Set (CRS). https://modsecurity.org/crs/.

[34] PCRE - Perl Compatible Regular Expressions. http://www.pcre.org/.

[35] Avalon Interface Specifications. https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/manual/mnl_avalon_spec.pdf.

[36] AXI Reference Guide. https://www.xilinx.com/support/documentation/ip_documentation/axi_ref_guide/v13_4/ug761_axi_reference_guide.pdf.

[37] Manuel Saldana and Paul Chow. TMD-MPI: An MPI implementation for multiple processors across multiple FPGAs. In *Field Programmable Logic and Applications, 2006. FPL'06. International Conference on*, pages 1–6. IEEE, 2006.

Appendices

A OpenCL sample code to use DUA API

Figure 15 shows an example Verilog code for application using DUA interface to initiate a connection and send data. It first generates a CONNECT primitive and then waits for the response. If the connection is successfully established, it records the source UID and port number, then enters the sending data state. If the connection setup fails, it will generate another CONNECT primitive. Note that the source address and port is not available before the connecting setup, thus this field is reserved when issuing a CONNECT command. And the response of CONNECT command will contain the corresponding fields.

B OpenCL sample code to use DUA API

Fig, 14 shows an OpenCL sample code pieces to use DUA API. *DUA_Msg* is a union storing DUA messages to be sent in current clock (see Fig. 7). *dua_tx* is a reserved channel that automatically connects to the request interface of our DUA I/O interface (Fig. 4). *simple_write ()* function writes 32B data to address *DST_ADDR* of the resource whose UID is *DST_UID* through DUA interface.

```
void simple_write () {
  DUA_Msg msg;
  bool is_header = true;
  while (1) {
    if (is_header) {
      msg.header.length = 32;
      msg.header.type = WRITE;
      msg.header.src_uid = SRC_UID;
      msg.header.dst_uid = DST_UID;
      msg.header.dst_addr = DST_ADDR;
      is_header = false;
    }
    else {
      msg.raw = data;
    }
    write_channel_altera(dua_tx, msg.raw);
  }
}
```

Figure 14: OpenCL sample code to use DUA API

```
assign rx_header = rx_data_in;

assign connect_header.type = CONNECT;
assign connect_header.dst_uid = dst_uid;
assign connect_header.dst_port = dst_port;

assign send_header.dst_UID = dst_UID;
assign send_header.src_UID = src_UID;
assign send_header.type = SEND;
assign send_header.length = 48;
assign send_header.src_port = src_port;
assign send_header.dst_port = dst_port;

always @(posedge clk) begin
  tx_valid_out <= 1'b0;
  tx_first_out <= 1'b0;
  tx_last_out <= 1'b0;
  case (state)
  SETUP_CONNECTION: begin
    if (tx_ready_in) begin
      tx_data_out <= connect_header;
      tx_valid_out <= 1'b1;
      state <= WAITING_RESPONSE;
    end
  end
  WAITING_RESPONSE: begin
    if (rx_valid_in
        && rx_data.type == CONNECT) begin
      if (rx_data_in.status == SUCCESS) begin
        src_UID <= rx_data_in.src_UID;
        state <= SENDING_HEADER;
      end
      else begin
        state <= SETUP_CONNECTION;
      end
    end
  end
  SENDING_HEADER: begin
    if (tx_ready) begin
      tx_data_out <= send_header;
      tx_valid_out <= 1'b1;
      tx_first_out <= 1'b1;
      state <= SENDING_DATA_0;
    end
  end
  SENDING_DATA_0: begin
    if (tx_ready) begin
      tx_valid_out <= 1'b1;
      tx_data_out <= data_0;
      state <= SENDING_DATA_1;
    end
  end
  SENDING_DATA_1: begin
    if (tx_ready) begin
      tx_valid_out <= 1'b1;
      tx_data_out <= {128'h0, data_1};  // 128bits
      tx_last_out <= 1'b1;
      state <= CLOSE_CONNECTION;
    end
  end
  CLOSE_CONNECTION: begin
    // close connection logic
  end
  endcase
end
```

Figure 15: DUA API usage example