DiAD – Distributed Acceleration for Datacenter FPGAs

DOI:
10.1109/FPL60245.2023.00031

Document Version
Accepted author manuscript

DiAD – Distributed Acceleration for Datacenter FPGAs

Joshua Lant, Emmanouil Skordalakis, Kyriakos Paraskevas, William B. Toms, Mikel Luján, John Goodacre
Department of Computer Science, University of Manchester, UK, M13 9PL.
Email: {firstname.lastname}@manchester.ac.uk

***Abstract*—The growing demands placed upon modern compute and network resources are far exceeding the capabilities of traditional computer architectures. It is now customary for accelerators to perform the bulk of compute, and this compute is being pushed ever closer to the network. FPGA vendors have brought powerful datacenter cards to the market, combining reprogrammable FPGA fabrics with high bandwidth networking capability. However, the supporting infrastructure has yet to reach maturity, so modern and diverse workloads are not yet able to fully leverage these architectural advances.**

In this paper we present DiAD; a novel framework providing FPGA firmware and driver support for fully unified, distributed compute and network acceleration across a commodity Ethernet network. We present a *far* richer feature set and greater flexibility than other existing solutions. We show comparable networking performance from the host when compared to Xilinx's OpenNIC solution. As well as allowing for host networking through the FPGA fabric, we demonstrate reliable data transfer directly between FPGA fabrics without host intervention, and show native memory transactions sent over the network supporting pointer-chasing workloads. We achieve line rate for dataflow type communication and approach line rate for larger native memory transfers (87G).

I. Introduction

FPGA vendors[1] have spent significant efforts on hardware design for the datacenter market, with advances also being made development tools [1], runtime technology [2], IP development [3], [4] and partial reconfiguration technology. Current trends around converging HPC and datacenter applications/infrastructure [5] require integration of FPGAs into large-scale commodity datacenter networks, with multiple FPGA devices per host, many possible configurations and with many complex use cases. For example; disaggregation [6] or bump-in-the-wire (BITW) [7] type architectures, use of tight memory coupling between the processing cores and FPGA [8], and datacenter application scaling [9].

Unfortunately, the majority of development effort is still designed to target single host/single accelerator configurations, with inter-FPGA communication being orchestrated through the hosts. This means there is now a large gap between the theoretic capability of the available hardware (e.g. multiple 100G ports connected to the FPGA fabric), and the realistic capabilities afforded by drivers, runtimes and surrounding FPGA infrastructure.

To address this gap, we have created a framework for Distributed Acceleration in Datacenter FPGAs (***DiAD***). DiAD is a hardware and software based framework, unifying the support for all major communication and computation paradigms for distributed FPGA accelerator applications, running over commodity Ethernet networks. Our framework consists of an RTL FPGA firmware platform, accompanying driver, and a custom transport layer supporting native memory transactions over IP networks. It targets Xilinx Alveo accelerator cards, and it is the first such solution to allow for all of the following features over the same 100G Ethernet network connected through the FPGA fabric: ***1)*** Traditional host-host networking through the FPGA fabric (for general networking capability). ***2)*** Traditional FPGA compute acceleration (for typical single host/single accelerator workloads). ***3)*** Bump-in-the-wire SmartNIC type architectures (for network or storage acceleration [10], [11], [12]). ***4)*** Direct network access for the FPGA fabric control and data plane (allowing disaggregation [13], zero-copy data transfer and avoiding host interaction [14]). ***5)*** Point-to-point data streaming over the commodity Ethernet network (for multi-FPGA dataflow applications [15], [16]). ***6)*** Remote Memory Access (RMA) one-sided communication/distributed shared memory (for pointer chasing applications, e.g. data-analytic or machine learning workloads [17]).

Whilst there are disparate frameworks which allow for any one or several of these paradigms to be implemented, a crucial issue regarding the general uptake of distributed FPGA acceleration is the lack of unified tools or standardised frameworks for development in such environments. Although it could be argued that the flexibility of the FPGA precludes these standardised development practices, within the context of datacenter networking this is simply not the case. Communication is extremely likely to be over PCIe or Ethernet depending on the scale (intra/inter-chassis), and no matter the configuration it will fall into one of the 6 computation/communication patterns discussed above. Our main motivation is thus to unify these patterns under a single framework to simplify the development of datacenter applications which exploit compute and/or network acceleration. After all, removing unused functionality from a well-structured design is vastly simpler than

[1]DiAD focuses on AMD/Xilinx devices as the use of the proprietary QDMA IP is central to the design.

	Corundum	EasyNet /VNx	NetFPGA+ /OpenNIC	FlexDriver	Catapult v2	DiAD
1. Host Networking	✓	✗	✓	✓	✓	✓
2. Compute Acceleration	✗	✓	✗	✓	✓	✓
3. Bump-in-the-Wire	✓	✗	✓	✓	✓	✓
4. Disaggregated FPGA Computing	✗	✓	✗	✓	✓	✓
5. Point-to-Point Dataflow	✗	✗	✗	✗	✗	✓
6. Distributed Memory Access/RMA	✗	✗	✗	✗	✗	✓

TABLE I: A comparison of the functionality between alternative solutions and DiAD.

attempting to add features to an existing framework. Removing functionality in DiAD takes advantage of synthesis tools (logic trimming) and using carefully segregated clock domains.

II. Related Work

The pressing need to support distributed FPGA acceleration within datacenters is evidenced by the number of solutions that have been developed in recent years to support FPGA datacenter networking. DiAD provides unified compute, inline network acceleration and direct inter-FPGA communication (disaggregation) through the same FPGA fabric, alongside hardware support for Remote Memory Acccess (RMA)/distributed shared memory. This section compares DiAD to the most recent developments in this area. Table I gives a breakdown of the capabilities of these solutions against DiAD for the introduced communication patterns. The following paragraphs describe each of these solutions; see table columns from left to right.

Corundum [18] is a framework for prototyping 100G NIC development on FPGAs. It consists of a custom PCIe DMA engine and Ethernet MAC, as well as a driver to support the Linux kernel networking stack. The design is able to achieve results approaching line rate communication from the host through the FPGA fabric. However, the PCIe IP is designed strictly for NIC prototyping, and thus cannot be used for compute acceleration or direct inter-FPGA fabric communication. The custom nature of the PCIe/DMA IP also means that Corundum cannot take advantage of the integrated DMA controllers available in the new Versal devices [3].

VNx [19] and EasyNet [20], provide UDP and TCP/IP support within the FPGA fabric respectively. The primary aim of both of these systems is to provide network offload for Vitis accelerator kernels, allowing the FPGA fabric to access the network directly. Since Xilinx Vitis imposes many constraints on the kernel interfaces, significant effort has been spent integrating EasyNet [20]. These include the creation of an API to abstract the TCP connection setup, and post-synthesis TCL scripting to integrate the CMAC network components. Since EasyNet employs custom interfaces to the network, host networking through the FPGA fabric is not currently supported, preventing bump-in-the-wire architectures from being deployed.

NetFPGA Plus [21] is a port of the NetFPGA project [22] to Alveo cards to provide 100G networking and PCIe gen 4 connectivity. NetFPGA Plus provides additions in the user logic portions of the existing Xilinx Open NIC Shell [23], and patch extensions to the associated Xilinx Open NIC driver [24]. The Open NIC Shell provides an RTL project with static portions for networking, and a region to add user logic for bump-in-the-wire type network acceleration. The Open NIC driver has several limitations which make it less suited to direct inter-accelerator communications. In simplifying the operation of their driver (making every PCIe Physical Function a Network Interface), they also prohibit traditional compute acceleration for the host [23].

FlexDriver [25] is a hardware module for FPGAs, developed by NVIDIA. It provides a data-plane driver designed to communicate with a dedicated NIC using peer-to-peer PCIe, giving RDMA capabilities to accelerators without a full FPGA NIC offload. This allows inter-FPGA communication without processor intervention, as well as inline style bump-in-the-wire packet processing. The solution is well adapted for SmartNIC type architectures which provide FPGA and NIC in the same card (e.g. NVIDIA Innova-2). However, use of the PCIe bus to send all traffic between the FPGA and NIC adds additional latency to inline network acceleration, and can lead to contention on the PCIe fabric. Since DiAD allows accelerators to communicate directly with with integrated QSFP modules it allows for multiple tenants within a single host without contention for network/PCIe resources.

The Catapult v2 infrastructure [10] developed by Microsoft is a proprietary solution which has some similarities with DiAD. For example, both use a dedicated, custom transport layer encapsulated in layer-3 IP packets implemented in the FPGA hardware for inter-FPGA communications. This transport layer holds unacknowledged frames in local buffers for retransmission, providing strong reliability. Similary, Catapult v2 can use traditional TCP/UDP software transports for networking and bump-in-the-wire applications. However, Catapult currently bypasses its hardware FPGA transport layer when using this traditional networking path among processing cores, directing this traffic to the cores through the NIC. Thus, Catapult would require additional hardware support to enable distributed memory accesses over the network using the FPGAs.

III. The DiAD Framework

DiAD consists of 3 major components:

1) Hardware Platform. A customised RTL hardware design platform for the Xilinx Alveo/Versal line of cards (implemented for this paper in the U200), built around the Xilinx QDMA IP for providing host-FPGA communication over PCIe[2]. The platform provides access to 100G QSFP ports,

[2]The use of either the XDMA or QDMA IP is integral for the targeted operation of these datacenter accelerator cards, so much so that they have been hardened as the CPM5 module on newer devices such as the Versal ACAP. The QDMA is the only viable solution for this work, as the ability to use four PCIe PFs is central to the rich feature set we offer.

memory interfaces from host or FPGA fabric, and interconnects supporting processing cores with integrated FPGA shared memory access.

2). QDMA-net An accompanying driver stack, comprising our extended version of Xilinx's QDMA driver [26], designed to enable one of the PCIe Physical Functions (PFs) of the QDMA hardware to appear and act as a standard Linux network device. The other PFs are left free for traditional host-accelerator data transfer, and Programmed-IO (PIO) or distributed/Remote Memory Access (RMA). Standard Xilinx user runtimes/applications such as XRT can be used in conjunction with our modified driver.

3) Network/Transport Layer. Customised IP to handle both IP traffic from the host (over the existing PCIe infrastructure), and encapsulate AXI traffic into IPv6 packets (on a dedicated subnet) to be passed over a commodity network. A transport layer is provided to enable reliability for inter-FPGA communication, for both native memory-mapped or streamed AXI transactions into the network. The native transactions support disaggregated FPGA computing applications, while the use of AXI stream interfacing directly to the integrated QSFP modules of the FPGA provides support for dataflow applications. Host and accelerator IP traffic are able to travel over the same network fabric by separating them out into different traffic types, using a specific IPv6 subnet for inter-FPGA networking communications.

A. Hardware Platform

Figure 1 shows an overview of the hardware platform. The DiAD system uses the Xilinx QDMA IP for host connectivity. The QDMA is capable of instantiating up to four PCIe Physical Functions (PFs). In this way we are able to segregate the functionality of the device into four distinct parts with distinct drivers. Each of the boxes inside the QDMA represents a physical interface between the FPGA and host PCIe. Each of these interfaces utilizes different driver functionality in order to support the various modes of communication.

The yellow portions show data paths for network packets, either through the host via the QDMA IP, or from the FPGA fabric directly via standard AXI4-MM memory-mapped or AXIS stream transactions. The blue portions show access to user defined accelerator kernel regions. In the base hardware platform these are simple placeholder modules to provide the connections to the interconnect of the FPGA. The host/FPGA can communicate using traditional memory transfers through the QDMA IP, and the accelerator itself can communicate with remote nodes using point-to-point stream connections, or using memory mapped global memory transfers. The red portions show memory-mapped bridging interfaces which can be used to provide distributed memory capabilities. These latter two (red and blue) use the same interconnect path since they access memory and network in the same way, utilising the offloaded transport layer of the FPGA hardware (see Section III-C). The grey AXI-Lite interface can be by any PF for configuration/status registers in the IPs. The framework can instantiate multiple memory controllers (MIGs) for accessing

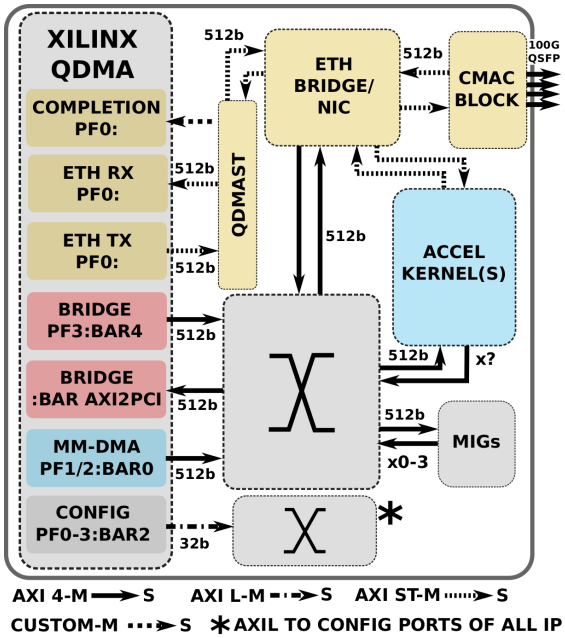

Fig. 1: Architecture of DiAD framework firmware, QDMA interface and corresponding Physical Functions (PF).

the local FPGA DRAM, providing higher memory bandwidth at a cost of additional area overhead.

QDMAST IP is a custom block that forms the interface between the QDMA AXIS and packet completion interfaces, and the Ethernet bridge and NIC. Traffic from the host is sent through this block unimpeded. On the RX side this handles the formation of *Completion Packets*. Every time an Ethernet packet arrives a corresponding completion packet is created, containing metadata such as IDs and packet length. This packet generates an interrupt internally and the metadata is used by our QDMA-Net driver to handle the inbound packet.

Eth Bridge/NIC IP performs several key functions. It primarily segregates IP traffic depending on its type. Inter-FPGA fabric communication is supported using a custom packet format encapsulated within IP packets targeting a private *FC00::/7* [27] IPv6 subnet to identify unique local addresses, with other traffic being directed toward the host. The IPv6/MAC address for the FPGA fabric communications are set statically in this IP before the NIC will accept any traffic (the host can still use ARP after this setup). This is required as broadcast IPv6 traffic may otherwise be incorrectly directed into the FPGA fabric's AXI interconnect. Since the FPGA fabric does not use a standard software stack to correctly drop the packet, it must be filtered out by the hardware. Otherwise it may be incorrectly decoded as a memory transaction destined for the FPGA. On the TX side the bridge will encapsulate stream or memory-mapped AXI transfers into IPv6 packets for transfer over the network. A source FPGA can reach a destination FPGA node ID (IPv6 address) beyond the limits of the physical address of the AXI-MM transaction using additional user bits within the

transaction itself (AW/AR/TUSER). On the RX side the bridge will identify the correct destination for packets (either host, FPGA fabric, or dropped), and generate the native memory transactions encapsulated in the FC00:: IPv6 subnet packets.

B. QDMA-net Software Stack

We extend the QDMA driver of Xilinx [28] for networking functionality. This driver was originally designed for userspace DMA applications, meaning no straightforward API is exposed to enable integration of networking operations from another kernel module. The QEP driver [29] of Xilinx (itself an extension of the QDMA) already has this networking functionality implemented using the QDMA, but has dependencies on proprietary hardware with unknown specifications and a custom shell released by Xilinx. Our QDMA-Net driver provides the core networking functionality of the QEP, while eliminating the need for a custom shell. Any Xilinx device capable of instantiating the QDMA and QDMAST IP (Figure 1) can run the QDMA-Net driver module.

In the original QDMA driver, software queues are implemented using character device files. A Xilinx userspace tool (*dma-ctl*) is provided that uses the netlink interface to allow users to create and start queues, associating them with a specific PCIe physical function. In our instance for the Ethernet PF0. We modify this functionality and incorporate it into the module so that TX and RX queues for the IP packets are instantiated automatically in the kernel along with the bring-up of the Linux network interface. Correspondingly, in the original driver, the function which adds DMA descriptors to the queues was called by read/write system calls. For our modified PF0, this is now called internally within the kernel by the transmit networking function.

C. Network and Transport Layer

As part of the DiAD framework we have developed a novel transport layer in order to enable disaggregated FPGA computing and host distributed/remote memory access using native AXI/AXIL/AXIS memory transactions. Thereby upgrading the status of the FPGA to a full network peer within the system. Traditional TCP offload techniques preclude the use of native memory transactions (which, e.g., have clear benefits for pointer-chasing workloads), as well as suffering other well known issues regarding scalability and memory bandwidth [30]. There are numerous other examples of custom transports being used over Ethernet/IP networks, such as Microsoft's Lightweight Transport Layer [10] (LTL, which was specifically designed for use among distributed FPGAs), and RoCE (RDMA over Converged Ethernet). There are also numerous examples of native memory transactions being encapsulated into Ethernet/IP packets [31], [32].

The DiAD transport layer is currently designed for use on systems with less than 10 simultaneously intercommunicating FPGA nodes, connected via a single switch. Targeting systems of this size means the packet buffers can be held within on-chip memory and also significantly simplifies the retransmission logic. The resulting reliability adds negligible overheads to the packet latency and is capable of line rate throughput. The retransmission protocol is loosely based on [33], extended to operate on IPv6 packets destined for a dedicated, private subnet (*FC00::/7* [27]), with multiple destinations.

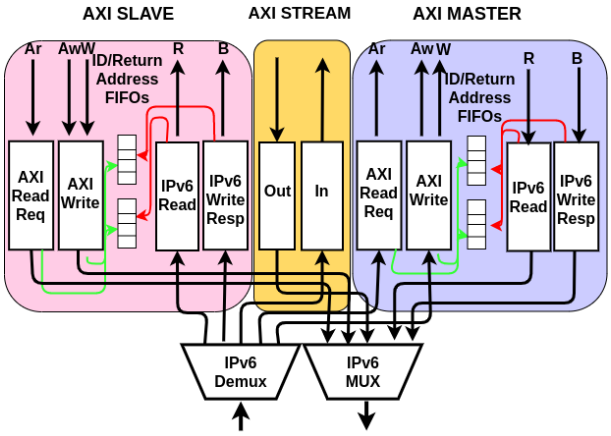

Fig. 2: IPv6 Bridge.

As well as servicing standard TCP/UDP packets from the host, the Eth Bridge/NIC (Figure 1) also encapsulates individual AXI transactions into IPv6 packets, allowing accelerator kernels or host memory transactions to communicate via simple AXI memory-mapped and AXI stream interfaces (see Figure 2). The bridge encodes transactions into 5 different types of IPv6 packets: ***Read Request***, ***Write Request***, ***Read Response***, ***Write Response***, and ***Stream***. Bidirectional AXI-MM transactions are controlled by IDs that are used to match requests to responses. The bridge has a fixed number of outstanding transactions and each transaction is given a unique ID. The reliability layer adds three additional packet types to the bridge types; acknowledgement ***ACK***, negative acknowledgement ***NACK***, and request for acknowledgement ***UNACK***.

The reliability layer uses a simple ordering model. No attempt is made to establish dependencies between packets. Instead, data packets to each destination are assigned a sequence number. Rather than attempt to reorder packets a receiver simply drops any packet received out of order from a destination, and they must be retransmitted. This significantly simplifies the logic required in the receiver, but incurs additional overheads when errors occur or packets are transmitted out of order.

IV. Implementation, System Installation and Network Performance

The current testbed installation consists of 6 host nodes connected to a 16x100G port Mellanox SN2100 switch. Each host node comprises an AMD EPYQ 7002 server (GIGABYTE G242-Z10 rev 100), with two Alveo U200 cards each attached via x16 lane gen3 PCIe, and connected via 100G QSFP to the switch. The host also has a separate 100G Mellanox ConnectX-5 NIC (MT27800) for testing and development

Name	LUT (%)	FF (%)	BRAM (%)	DSP (%)
Total Available	1182240	2364480	2160	6840
Full Design	271895 (23)	396590 (16.8)	450.5 (20.9)	16 (0.2)
QDMA	113055 (9.6)	99130 (4.2)	121 (5.6)	7 (0.1)
CMAC+FIFOs	8833 (0.7)	30924 (1.3)	97.5 (4.5)	0 (0)
Bridging IP	5131 (0.4)	5005 (0.2)	0 (0)	0 (0)
QDMAST	6113 (0.5)	12883 (0.5)	131 (6.1)	0 (0)
DDR4 CTL	59752 (5.1)	74220 (3.1)	76.5 (3.5)	9 (0.1)
Cross SLR Reg	21790 (1.8)	27325 (1.2)	0 (0)	0 (0)
Interconnects	57221 (4.8)	147103 (6.2)	24.5 (1.1)	0 (0)

TABLE II: IP Utilization on Alveo U200.

purposes. A vanilla kernel version 5.4.0 has been used for the collection of these results.

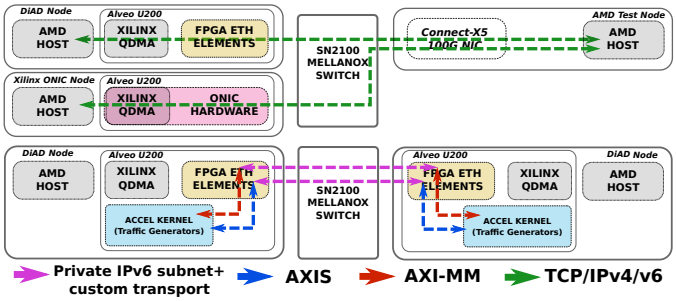

Fig. 3: Communication paths between nodes in the testbed, showing traditional Linux networking (top) and direct inter-FPGA (bottom) through the same fabric over a commodity Ethernet network switch.

The utilization for the DiAD framework is shown in Table II, using around 23% of the logic resources, with the QDMA IP alone using over 1/3 of this. The full design follows that of Figure 1, using 3-attached MIGs to interface the FPGA's DDR memory, and with placeholder modules in place of accelerator kernels (so that no interconnect logic is trimmed during synthesis). The design leaves ≈910K LUTs for accelerators. This is comparable to commercial hardware shell offerings such as that of AWS. The Amazon AWS F1.X.1.4 shell leaves ≈895K LUTs on a near-identical FPGA, and does not support the use of distributed accelerators [34].

A. *Linux Network Interface – Comparison to Xilinx OpenNIC*

Table III shows the results of running the ping and iperf3 benchmarks, in both instances with one side using the FPGA NIC (either DiAD or Xilinx's OpenNIC shell and driver [23]), communicating with a standard ConnectX-5 NIC on the other side of the Mellanox switch. This setup is shown in the upper portion of Figure 3. OpenNIC is built and set up with the default configuration for both shell (github commit #9a77192) and driver (github commit #c222cb4).

1) Achieving Line Rate: The achievable bandwidth for both DiAD and OpenNIC is far below that of the line rate of the 100G interface. Reasons for this are discussed in the OpenNIC FAQs [24] along with solutions to resolve the issues:

- The overhead caused by context switching between kernel and user-space can be reduced by using DPDK to deploy (most of) the network stack in user-space. This optimization is a considerable additional engineering task, requiring a completely separate API to be used for the networking component of the driver stack.
- Additional offloading techniques (e.g. TSO/GSO), or multi-queue NIC techniques (e.g. RSS) are required to reduce the processing burden on the processing cores/allow for multiple cores to be used. This optimization is simple to add within the driver itself, requiring only a set of flags to be enabled on device loading. However additional hardware is required to implement these features.

The DiAD results show similar performance to that of NetFPGA Plus, which uses the OpenNIC driver. They are able to achieve line rate communication within the FPGA fabric itself (as are we, see Section IV-B2). However, they are only able to achieve a fraction of this using the OpenNIC driver from the host. They process 297m 64B packets per second in the fabric, but only 5.5m (or 4m 1518B packets) when using the QDMA and driver stack [21]. We have not taken precise bandwidth measurements for NetFPGA Plus, but given the packets per second processing this gives ≈6Gb/s. Given the fact that the driver stack is clearly the bottleneck for both DiAD and NetFPGA Plus, we can see that these results are in line with our own OpenNIC experiments (Table III).

2) Ping: The results of the ping flood show that the latency of OpenNIC is on average 60% higher than for our DiAD NIC. Given the time taken for software to interrupt the system and context switch when compared with the raw hardware, it is highly unlikely that this discrepency is caused by the user logic itself. It is far more likely that because the interrupt handling is performed in a different manner from that of the OpenNIC driver, we are able to see lower average latency. The results of the iperf3 benchmark give us a greater clue as to why exactly this may be the case.

3) iperf3: The iperf3 results are interesting in that they show comparable levels of performance, with each of the FPGA NICs outperforming the other by a similar margin depending on whether being run in server or client mode (-s or -c respectively). In client mode, saturating the TX queue(s) of the driver, OpenNIC gives around 36% higher performance than DiAD, but in server mode, saturating the RX queue(s) of the driver, DiAD gives 39% higher performance.

One possible explanation of why DiAD outperforms OpenNIC in the RX path is that the number of retransmissions seen over the 10s run is around two orders of magnitude higher for OpenNIC than that of DiAD (101 against 35021 respectively). It is worth noting here that when acting as a client, neither of the FPGA NICs see any retransmissions. This suggests that rather than a faulty link causing packet corruption and subsequent drops, buffer overflow occurs on the receiver, and TCP's window scaling protocol is likely to have throttled the bandwidth to a greater degree in the case of OpenNIC.

This overflow and packet dropping could be due to either smaller buffers in the RX queue on the FPGA fabric in OpenNIC, or to do with differences in how interrupts are handled in the driver, which means that RX packet processing

	Open NIC	DiAD
ping flood		
minimum	0.019 ms	0.016 ms
average	0.056 ms	0.035 ms
iperf3		
Peak interval TX	10.1 Gb/s	8.09 Gb/s
Peak interval RX	4.85 Gb/s	4.97 Gb/s
Average TX	9.08 Gb/s	6.66 Gb/s
Average RX	3.51 Gb/s	4.88 Gb/s

TABLE III: DiAD vs OpenNIC.

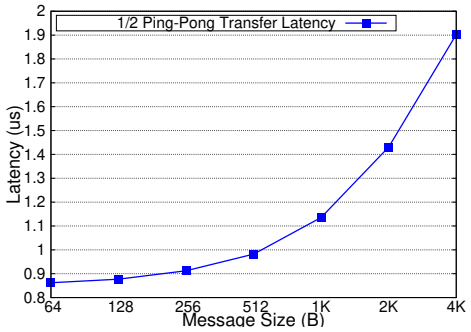

Fig. 4: Inter-FPGA latency.

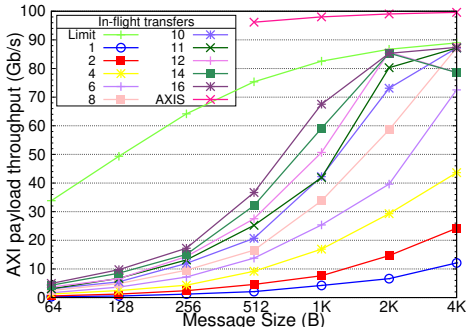

Fig. 5: Inter-FPGA throughput.

cannot happen as quickly in the case of OpenNIC. We see dramatic changes in the performance of DiAD when changing the interrupt mode of the driver from "direct" to "aggregation" modes [28]. Given the complexity and variety of the interrupt mechanisms in the QDMA hardware and it's driver, it is clear that optimizing the interrupt handling is key to properly exploiting the hardware.

This same complexity surrounding interrupt mechanisms is true for the TX side, where we see that performance is limited by the fact that we only employ a single TX queue currently, as opposed to OpenNIC which appears to employ 8 (according to the result of *netif_set_real_num_tx_queues()* when running OpenNIC on our system). Employing multiple queues effectively within the QDMA means handling distinct MSI-X interrupt vectors within the driver for each queue, so as to handle each individual TX queue on a separate core on the host. The use of multiple TX queues is a matter of ongoing development.

B. FPGA-FPGA Fabric Communication

In order to measure the latency and bandwidth of inter-FPGA communication, we set up a custom traffic generator IP instead of the placeholder accelerator block, and connected it into the bridge (Figure 1) via both the AXI Memory-Mapped (AXI-MM) and AXI Stream (AXIS) ports. Again each FPGA is connected through the Mellanox switch. The communication path for this is shown in the lower portion of Figure 3.

1) Latency: Figure 4 shows the latency to send AXI-MM packets of a given message size between FPGA fabrics over the switch. The Y-axis displays half the round-trip time for the message to be sent, and a corresponding message of the same size to be returned. This allows us to accurately measure the transfer latency as a number of cycles in a single clock domain. For small packets the round-trip latency seems relatively high ($\approx$1.8μs), suggesting a latency of just under 300ns for traversing each of the FPGA's fabric. This is given the 250MHz operating frequency of the main FPGA hardware, and the stated brochure port-to-port latency of the SN2100 switch (300ns [35]). However, it must be noted that we incur two store-and-forward penalties (necessary for the Xilinx CMAC IP), as well as having to use numerous register slices in the hardware block design and pipelining within the IP components in order to reach timing. VNx allows for FPGA-FPGA transfers, and provides a low-latency, unreliable UDP transport in the Alveo U280 hardware. They show a round-trip latency of packets sent over a switch between their two accelerator UDP components of 2.759μs [36]. Likewise, the reliable transmission used in EasyNet shows a RTT for 64B packets of 4.3μs [20]. Whilst these solutions use different switches from that which we use, these figures are sufficiently high for us to conclude that the latency we exhibit is normal for this type of solution.

2) Throughput: Figure 5 shows the achievable throughput between FPGA fabrics through a single Mellanox switch, for both native memory transfers (using AXI-MM interface), and for point-to-point streaming type applications (using AXIS interface). The results show that there are large overheads on small packets, as the scalability of the transport layer depends on the number of possible in-flight transactions. For this reason we vary the number of possible in-flight transactions in the memory-mapped case between 1 and 16. This is not the case for point-to-point AXIS connections, as the retransmission buffer will simply deassert the TREADY signal when it can accept no more data, so in this instance the performance is bound by the volume of unacknowledged data in the transactions, rather than the number of in-flight transactions.

The results show that the links are saturated using memory-mapped transfers at around 87Gb/s using only 8$\times$4K packets, or around 14$\times$2K packets over a single switch. We show that for an 8K message (2$\times$4K packets) we can achieve $\approx$25Gb/s bandwidth. These numbers can be roughly compared to the EasyNet solution, who's published results show saturation levels approaching line-rate for larger message sizes of 1MB, but can only achieve rates of ($\approx$15Gb/s) using smaller 10K messages [20]. This discrepancy may be due to the overheads associated with TCP's window scaling algorithm, and the fact that this is amortized during large transfers.

The results show our own stream interface is capable of approaching the effective line-rate, at 99.6Gb/s, but this is obviously only applicable for point-to-point dataflow applications. This primary difference is due to the overheads on the metadata associated with sending the memory-mapped

transfers (1/8*N+1/64*N, where N is bytes per message). This metadata includes all of the information required to form the full AXI4 bus protocol on the remote side. The theoretical limit of the AXI-MM transactions in Figure 5 shows that for small message sizes many more packets need to be issued concurrently to achieve saturation, and that the performance can never approach line rate due to the protocol overheads on the packets. The concurrent issuing of transactions is an optimization of the transport layer protocol, where a trade-off between achievable small-packet performance and area overheads must be considered.

C. Google Multi-Chase Benchmark

In order to highlight the properties of DiAD which are not afforded to any of the related works discussed in Section II we have run the Google Multichase benchmark [37] on the host cores, accessing resources other than its own DRAM. This demonstrates the use of Software Defined Networking to instantiate and enable access to distributed memory resources across the system. The development of a reconfigurable address translation hardware mechanism allows the dynamic instantiation and mapping of resources into a global address space (GAS). This GAS is then mapped to the process as part of the available PCIe address window, and is directly accessible by native load/store instructions generated by the processor, bypassing the software stack. Given the limited capability of processors to issue outstanding transactions, we select to measure the latency of accessing different types of memory resources across the system (remote reads), as shown in Figure 6. The average 8-byte read latency (round trip) for accessing remote FPGA DDR is 3.2 us, while the latency to access remote host DDR is 3.9 us, giving a one-way latency of $\approx$1.6 and 1.95 us respectively. Note the additional latency for accessing remote FPGA or host resources over the results for small packet sizes seen in Figure 4. This is attributed to the cost of traversing the PCIe interface (around 450ns per-hop [38]). This shows the benefits of our architecture in allowing direct control of the networking components from the FPGA fabric itself, as opposed to an architecture such as FlexDriver [25] in which NIC and FPGA fabric are separated by a PCIe hop, particularly for pointer-chasing workloads.

V. Further Optimizations

To achieve host networking performance that approaches the theoretical capability of a 100G QSFP module on the Alveo U200 would require additional engineering. The host-FPGA data path of the QDMA can saturate one of the 100G modules with raw DMA transfers. Data generated over x16 gen3 PCIe links are capable of roughly 128Gb/s throughput. However, constraints within the evaluated version of the QDMA hardware and networking driver stack require modification to saturate the host to fabric datapath.

Using the DPDK (Data Plane Development Kit) would enable DiAD to achieve line rate. DPDK moves significant portions of TCP packet processing into the userspace processes, rather than allowing kernel interrupts to handle this.

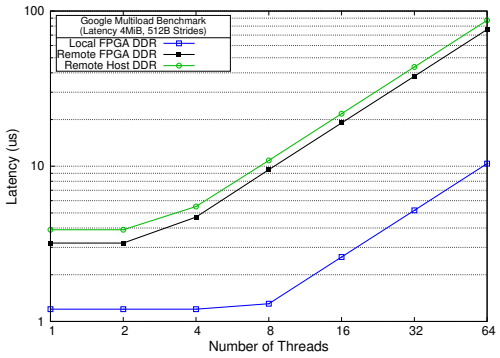

Fig. 6: Google Multichase (Multiload) benchmark results with multiple threads accessing local or remote memory resources.

By polling the hardware from userspace rather than handling hardware interrupts in the operating system, the overheads of context switching are reduced. Setting this up however requires reengineeing of the driver stack to support the DPDK APIs and to support polling of the hardware queues.

It is also possible that further performance increases could be gained for the implementation we provide for the standard Linux kernel networking stack, by the addition of hardware optimizations and protocol offloading techniques. An example of features which could be added are Receive-Side Scaling, TCP Segmentation Offload (TSO), or checksum offloading. However, the current driver would also require modifications aside from the hardware optimizations. The current version of the framework only utilises a single core for both TX and RX packet processing, meaning that packet processing for RX and TX cannot happen concurrently. This is due to the way that interrupts are handled through a single vector.

VI. Conclusions

This paper has presented DiAD, a framework enabling *Distributed Acceleration in Datacenter FPGAs*. This framework consists of a hardware design and software stack which allows for compute or network acceleration, disaggregation, and distributed shared memory all from the same base design. DiAD provides a richer feature set than existing solutions; supporting line-rate communication between networked accelerator fabrics using native memory transactions, and supporting traditional host-accelerator communications. We have showed how the hardware and driver enable simple, native, reliable, transactions between FPGA fabrics without host intervention, and further demonstrated the novel aspects using the Google multichase benchmark.

Acknowledgments

This work was funded partly by the EU Horizon 2020 programme under grant agreement No. 754337, and by the UK Industrial Strategy Challenge Fund (ISCF) under the Digital Security by Design (DSbD) Programme delivered by UKRI as part of the MoatE (10017512) and Soteria (75243) projects.

Mikel Luján is supported by a Royal Society Wolfson Fellowship and an Arm/RAEng Research Chair Award. AMD/Xilinx donated FPGAs to support this research.

REFERENCES

[1] Xilinx Inc. (2021) Vitis unified software platform. [Online]. Available: https://www.xilinx.com/products/design-tools/vitis/vitis-platform.html

[2] Xilinx Inc. (2021) Xilinx runtime (XRT) architecture. [Online]. Available: https://xilinx.github.io/XRT/master/html/index.html

[3] Xilinx Inc. (2021) DS950 (v1.14), Versal architecture and product data sheet: Overview. [Online]. Available: https://www.xilinx.com/support/documentation/data_sheets/ds950-versal-overview.pdf

[4] Xilinx Inc. (2021) AM016 (v1.1), Versal ACAP CPM CCIX architecture manual. [Online]. Available: https://www.xilinx.com/support/documentation/architecture-manuals/am016-versal-cpm-ccix.pdf

[5] T. Hoefler, A. Hendel, and D. Roweth, "The convergence of hyperscale data center and high-performance computing networks," *Computer*, vol. 55, no. 7, pp. 29–37, 2022.

[6] F. Abel, J. Weerasinghe, C. Hagleitner, B. Weiss, and S. Paredes, "An FPGA platform for hyperscalers," in *2017 IEEE 25th Annual Symposium on High-Performance Interconnects (HOTI)*. IEEE, 2017, pp. 29–32.

[7] J. Fowers, K. Ovtcharov, M. K. Papamichael, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, L. Adams, M. Ghandi *et al.*, "Inside project brainwave's cloud-scale, real-time ai processor," *IEEE Micro*, vol. 39, no. 3, pp. 20–28, 2019.

[8] P. Vogel, A. Marongiu, and L. Benini, "Exploring shared virtual memory for fpga accelerators with a configurable iommu," *IEEE Transactions on Computers*, vol. 68, no. 4, pp. 510–525, 2019.

[9] E. Chung, J. Fowers, K. Ovtcharov, M. Papamichael, A. Caulfield, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman *et al.*, "Serving dnns in real time at datacenter scale with project brainwave," *iEEE Micro*, vol. 38, no. 2, pp. 8–20, 2018.

[10] A. M. Caulfield, E. S. Chung, A. Putnam, H. Angepat, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur, J.-Y. Kim *et al.*, "A cloud-scale acceleration architecture," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2016, pp. 1–13.

[11] A. Stratikopoulos, C. Kotselidis, J. Goodacre, and M. Luján, "Fastpath: towards wire-speed NVMe SSDs," in *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2018, pp. 170–177.

[12] A. Stratikopoulos, C. Kotselidis, J. Goodacre, and M. Luján, "Fastpath_mp: Low overhead energy-efficient fpga-based storage multipaths," *ACM Trans. Archit. Code Optim.*, vol. 17, no. 4, nov 2020.

[13] J. Weerasinghe, F. Abel, C. Hagleitner, and A. Herkersdorf, "Disaggregated FPGAs: Network performance comparison against bare-metal servers, virtual machines and Linux containers," in *2016 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE, 2016, pp. 9–17.

[14] J. Lant, J. Navaridas, M. Luján, and J. Goodacre, "Toward FPGA-based HPC: Advancing interconnect technologies," *IEEE Micro*, vol. 40, no. 1, pp. 25–34, 2019.

[15] D. Oriato, S. Girdlestone, and O. Mencer, "Dataflow computing in extreme performance conditions," in *Dataflow Processing*, ser. Advances in Computers, A. R. Hurson and V. Milutinovic, Eds. Elsevier, 2015, vol. 96, pp. 105–137.

[16] Maxeler Technologies. Programming MPC systems white paper. [Online]. Available: https://www.maxeler.com/files/MPCwhitepaper.pdf

[17] G. Weisz, J. Melber, Y. Wang, K. Fleming, E. Nurvitadhi, and J. 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*, 2016, pp. 264–273.

[18] A. Forencich, A. C. Snoeren, G. Porter, and G. Papen, "Corundum: An open-source 100-gbps nic," in *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2020, pp. 38–46.

[19] M. Ruiz and N. Johnson. (2021) XUP vitis network example (VNx) repository. [Online]. Available: https://github.com/Xilinx/xup_vitis_network_example

[20] Z. He, D. Korolija, and G. Alonso, "EasyNet: 100 Gbps network for HLS," in *International Conference on Field-Programmable Logic and Applications (FPL 2021)*, 2021.

[21] Y. Tokusashi, "NetFPGA-Plus: the next generation of NetFPGA platforms."

[22] J. W. Lockwood, N. McKeown, G. Watson, G. Gibb, P. Hartke, J. Naous, R. Raghuraman, and J. Luo, "NetFPGA – an open platform for gigabit-rate network switching and routing," in *2007 IEEE International Conference on Microelectronic Systems Education (MSE'07)*. IEEE, 2007, pp. 160–161.

[23] Xilinx Inc. (2021) OpenNIC technical reference guide. [Online]. Available: https://github.com/Xilinx/open-nic/blob/main/OpenNIC_manual.pdf

[24] Xilinx Inc. (2021) OpenNIC project. [Online]. Available: https://github.com/Xilinx/open-nic

[25] H. Eran, M. Fudim, G. Malka, G. Shalom, N. Cohen, A. Hermony, D. Levi, L. Liss, and M. 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*, 2022, pp. 1115–1129.

[26] Xilinx Inc. (2021) Xilinx DMA IP reference drivers. [Online]. Available: https://github.com/Xilinx/dma_ip_drivers

[27] B. Haberman and B. Hinden, "Unique Local IPv6 Unicast Addresses," RFC 4193, Oct. 2005. [Online]. Available: https://rfc-editor.org/rfc/rfc4193.txt

[28] Xilinx Inc. (2020) Xilinx QDMA linux drivers. [Online]. Available: https://xilinx.github.io/dma_ip_drivers/2020.1/linux-kernel/html/index.html

[29] Xilinx Inc. (2021) Xilinx QEP linux kernel network driver. [Online]. Available: https://xilinx.github.io/qep-drivers/master/linux-kernel/html/index.html

[30] D. Sidler, G. Alonso, M. Blott, K. Karras, K. Vissers, and R. Carley, "Scalable 10gbps tcp/ip stack architecture for reconfigurable hardware," in *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*. IEEE, 2015, pp. 36–43.

[31] Chips Alliance. (2021) OmniXtend cache coherence protocol. [Online]. Available: https://github.com/chipsalliance/omnixtend

[32] Z. Bandic, D. Vucinic, R. Golla, and Western Digital, "Cpu project in western digital: From embedded cores for flash controllers to vision of datacenter processors with open interfaces," in *2018 Inaugural RISC-V Summit*, 2018.

[33] L. A. Plana, J. Garside, J. Heathcote, J. Pepper, S. Temple, S. Davidson, M. Luján, and S. Furber, "spinnlink: FPGA-based interconnect for the million-core SpiNNaker system," *IEEE Access*, vol. 8, pp. 84 918–84 928, 2020.

[34] Amazon. (2021) AWS FPGA small shell. [Online]. Available: https://github.com/aws/aws-fpga/tree/small_shell

[35] Mellanox Technologies. (2019) SN2700 open ethernet switch. [Online]. Available: https://www.mellanox.com/related-docs/prod_eth_switches/PB_SN2700.pdf

[36] M. Ruiz. (2021) XUP benchmark round trip time experiment (switch). [Online]. Available: https://github.com/Xilinx/xup_vitis_network_example/blob/master/Notebooks/vnx-benchmark-rtt-switch.ipynb

[37] Google. (2020) Multichase benchmark. [Online]. Available: https://github.com/google/multichase

[38] R. Neugebauer, G. Antichi, J. F. Zazo, Y. Audzevich, S. López-Buedo, and A. W. Moore, "Understanding PCIe performance for end host networking," in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, 2018, pp. 327–341.