FpgaNIC: An FPGA-based Versatile 100Gb SmartNIC for GPUs

Zeke Wang[1], Hongjing Huang[1], Jie Zhang[1], Fei Wu[1,2]
[1] *Collaborative Innovation Center of Artificial Intelligence, Zhejiang University, China*
[2] *Shanghai Institute for Advanced Study of Zhejiang University, China*

Gustavo Alonso
Systems Group, Dept. of Computer Science
ETH Zurich, Switzerland

Abstract

Network bandwidth is improving faster than the compute capacity of the host CPU, turning the CPU into a bottleneck. As a result, SmartNICs are often used to offload packet processing, even application logic, away from the CPU. However, today many applications such as Artificial Intelligence (AI) and High Performance Computing (HPC) rely on clusters of GPUs for computation. In such clusters, the majority of the network traffic is created by the GPUs. Unfortunately, commercially available multi-core SmartNICs, such as BlueFiled-2, fail to process 100Gb network traffic at line-rate with its embedded CPU, which is capable of doing control-plane management only. Commercially available FPGA-based SmartNICs are mainly optimized for network applications running on the host CPU. To address such scenarios, in this paper we present FpgaNIC, a GPU-oriented SmartNIC to accelerate applications running on distributed GPUs. FpgaNIC is an FPGA-based, GPU-centric, versatile SmartNIC that enables direct PCIe P2P communication with local GPUs using GPU virtual address, and that provides reliable 100Gb network access to remote GPUs. FpgaNIC allows to offload various complex compute tasks to a customized *data-path accelerator* for line-rate in-network computing on the FPGA, thereby complementing the processing at the GPU. The data-path accelerator can be programmed using C++-based HLS (High Level Synthesis), so as to make it easier to use for software programmers. FpgaNIC has been designed to explore the design space of SmartNICs, e.g., direct, on-path, and off-path models, benefiting different type of application. It opens up a wealth of research opportunities, e.g., accelerating a broad range of distributed applications by combining GPUs and FPGAs and exploring a larger design space of SmartNICs by making them easily accessible from local GPUs.

1 Introduction

While the computing capacity of CPUs is growing slowly and mostly either through parallelism (SIMD, multi-core) or specialization (GPGPU, security or virtualization support), network bandwidth is growing obviously faster. 100Gbps NICs are common and soon 400Gbps will be available [46]. This growing gap between network bandwidth and compute capability is being addressed through offloading of network functions to the Network Interface Card (NIC), so called SmartNIC [13, 14, 19, 36, 45], which frees up significant CPU cycles and provides better hardware to keep up with the growing network traffic and its often strict requirements in terms of bandwidth and latency.

Modern GPUs provide an order of magnitude higher memory bandwidth and higher compute capacity than modern CPUs. As a result, GPUs have become a key element in, e.g., Artificial Intelligence (AI) and High Performance Computing (HPC) applications that are both compute- and memory-bound [4]. Since a multi-GPU server is often not enough to cover the computing power needed in many AI, graph, and HPC applications, current solutions are typically based on a cluster of GPUs (e.g., [27, 58, 78]), with the GPUS generating the majority of the network traffic in such systems.

In this paper, we present the design of a 100Gb GPU-centric SmartNIC to serve distributed applications running on GPUs. From a GPU's perspective, such a SmartNIC should 1) enable the GPU directly triggering doorbell registers and polling on status registers on the SmartNIC without CPU intervention (**G1**); 2) use the GPU virtual address space to directly access GPU memory via Peer-to-Peer (P2P) communication without CPU intervention (**G2**); 3) implement in hardware the full network stack to ensure low latency and high throughput (**G3**); 4) support application logic offloading to a software-defined and hardware-accelerated data-path accelerator, i.e., *on-NIC* computing processing 100Gb network traffic at line-rate (**G4**)[1]; and 5) The data-path accelerator should be easily programmed by system programmers (**G5**). Commercially available SmartNICs are not able to satisfy all these goals as they are not optimized for GPUs. In the following, we analyze existing multicore and FPGA-augmented SmartNICs that motivate FpgaNIC.

Multicore SmartNIC. A multicore SmartNIC, such as

[1] In the paper, we use on-NIC computing module and data-path accelerator interchangeably.

Table 1: Comparison of FpgaNIC with existing SmartNIC types for GPUs. ✓ indicates full support, ✗ indicates no support, and ✓✗ indicates partial support.

	Multicore SmartNIC [44]	FPGA-aug. SmartNIC [45]	Ours
Control plane offload (G1)	✗	✗	✓
Access GPU with virtual address (G2)	✓	✗	✓
100Gb transport offload (G3)	✓✗	✓✗	✓
100Gb data-path accelerator (G4)	✗	✓✗	✓
High programmability (G5)	✓	✗	✓

BlueField-2 [44], combines a multicore CPU, e.g., ARM, with an ASIC network controller. It introduces an additional hop to implement the smart function using a multicore CPU, which features two DDR4 channels for staging. This allows to map a broad range of applications on multicore SmartNICs. Therefore, its high programmability **G5** is fully supported. However, it increases processing latency and multicore CPU's memory bandwidth can easily become a performance bottleneck. BlueField-2 has 27.3GB/s achievable memory bandwidth under a benchmarking tool sysbench [1], indicating that directly staging 100Gbps data stream at the NIC CPU already overwhelms BlueField-2, matching the findings in [40]. Therefore, it cannot act as a 100Gb data-path accelerator **G4**. To our knowledge, the multicore SmartNIC is controlled from the host CPU, so **G1** is not yet supported. The network transport is implemented with the packet processing engine with necessary control on the host (or ARM) CPU, so **G3** is partially supported. Finally, the ASIC network chip of multicore SmartNIC supports NVIDIA GPUDirect [52], which enables direct PCIe P2P data communication to a GPU, so **G2** is fully supported.

FPGA-augmented SmartNIC. An FPGA-augmented SmartNIC combines a hardware-programmable FPGA with an ASIC network controller. For example, Mellanox Innova-2 [45] is an FPGA-augmented SmartNIC featuring a network adapter ConnectX-5 and an Xilinx FPGA. ConnectX-5 consists of a 100Gbps InfiniBand/Ethernet interface for networking and a PCIe Gen4x8 interface for communicating with the host CPU. The FPGA communicates with ConnectX-5 via a PCIe x8 Gen4 link, so processing packets on the FPGA adds considerable latency to the packets and processing cannot happen at line rate because Innova-2 has limited PCIe link bandwidth between the FPGA and ConnectX-5. Therefore, Innova-2 can only acts as a partial 100Gb data-path accelerator **G4**. **G1** is not yet supported, **G2** is not supported, **G3** is partially supported, and the high programmability **G5** is not supported.

Given the limitations of existing NICs, in this paper we present FpgaNIC, a full-stack FPGA-based GPU-centric versatile SmartNIC that opens up the opportunity to explore a *large design space around SmartNICs* due to the FPGA's reconfiguable nature and efficient *FPGA-GPU co-processing* while achieving all the five goals mentioned above in a single system. We have implemented FpgaNIC as a composable architecture that consists of a *GPU communication stack*, a *100Gb hardware network transport*, and an *On-NIC computing* (ONC), i.e., data-path accelerator.[2] The GPU communication stack enables offloading of control plane onto GPUs (**G1**) and thus *for the first time* enables local GPUs directly to manipulate SmartNIC without CPU intervention. and enabling the FPGA-based SmartNIC *for the first time* to use GPU virtual address to directly access GPU memory via PCIe P2P communication (**G2**). The 100Gb hardware network transport enables efficient and reliable 100Gb network communication with remote GPUs (**G3**). Moreover, FpgaNIC adopts a layered design to allow developers to easily explore the design space of SmartNIC models (i.e., direct, off-path, and on-path) to benefit their application, where different applications favor a different SmartNIC model. FpgaNIC allows to prototype applications that can eventually be migrated to hardened SmartNICs. Implementing a data-path accelerator on an FPGA can easily satisfy line-rate processing requirement (**G4**) due to its hardware implementation, while FpgaNIC allows to use C++-based High Level Synthesis (HLS) so as to provide high programmability (**G5**). As such, in the context of FPGA-GPU co-processing, the GPU provides to applications expressiveness and computing flexibility, while the FPGA provides a flexible network infrastructure and the necessary ONC. FpgaNIC results in significant end-to-end performance improvements as data can be processed as it flows from/to the GPU in a streaming manner and without involving the CPU.

We have prototyped FpgaNIC on a PCIe-based Xilinx FPGA board Alveo U50 [74], whose UltraScale+ FPGA features a 100Gbps networking port, a X16 PCIe Gen3, and 8GB HBM. Its form factor is half-length, half-height and its Maximum Total Power (MTP) is 75W, allowing it to be easily deployed in any CPU server. [3]In addition to comprehensive benchmarking, we validate the **versatility** and **potential** of FpgaNIC by implementing use cases for all three models: GPU-centric networking (in a direct model), a collective primitive AllReduce (in an off-path model), and cardinality estimation on incoming streaming data (in an on-path model). The experimental results show that FpgaNIC is able to efficiently support all three SmartNIC models at the full line rate of 100 Gbps Ethernet. Particularly, FpgaNIC-enhanced AllReduce almost reaches the maximum theoretical throughput when performing on a distributed pool of eight RTX 8000 GPUs, while requiring fewer than 20% of the FPGA resources on the U50 board. It indicates that, even when considering the full network stack offloading, it has sufficient FPGA resources to allow more aggressive offloading, e.g., the Adam

[2]In the paper, we use on-NIC computing module and data-path accelerator interchangeably.

[3]We have also migrated FpgaNIC onto the Alveo U280 FPGA board [73] with minor modifications affecting the FPGA pin mapping. Though we have not ported FpgaNIC to Intel FPGA boards yet, we believe that it requires only a small amount of effort to do so. We leave the porting to future work.

optimizer [31].[4] As such, FpgaNIC enables efficient FPGA-GPU co-training on Deep Learning models. We leave this exploration to future work.

2 Design and Implementation of FpgaNIC

2.1 Design Challenges

We highlight four concrete research challenges we faced in designing FpgaNIC.

C1: How to Enable the FPGA to Access the GPU Virtual Address? Enabled by NVIDIA GPUDirect [52], the DMA engine in the PCIe IP core allows the FPGA to efficiently transfer data from and to GPU memory via issuing a DMA read/write command that consists of a starting physical address and length (no larger than a GPU page size). However, doing so in the context of SmartNIC raises two challenges. First, a GPU program manipulates GPU virtual address rather than physical address, so the FPGA should work on GPU virtual address to be consistent with the view of GPUs. Second, a single contiguous virtual address space needs not to be physically contiguous on GPU memory, and the typical memory page size is 64KB on modern GPUs as they do not yet support huge pages, making TLB management really challenging, especially when the required number of TLB entires is large.

C2: How to Enable Efficient Reliable Network Transport between Distributed GPUs? Modern GPUs have become a key compute engine to power AI and HPC applications due to its massive parallel compute capacity and huge memory bandwidth. AI and HPC applications typically need reliable network communication between distributed GPUs to realize GPU-accelerated cluster computing. However, GPUs are not originally designed for reliable network transport [28, 43] since reliable networking reduces the degree of parallelism and requires a complex flow control, e.g., retransmission.

The straightforward approach to realize network transport is to implement it on the CPU. Such a CPU-based approach consumes several CPU cores to implement a 100Gbps network transport layer. Furthermore, the network operations are initiated from the GPU, incurring longer network latencies. Instead, we offload the implementation of the reliable network transport to the FPGA to make the data plane fully bypass the host CPU. Fortunately, there is a growing amount of open-source FPGA-based 100Gb network transports [3] such as the TCP/IP stack of [57, 60] and the RoCEv2 stack used in [62]. However, how to enable the GPU to efficiently manipulate the reliable hardware transport on the FPGA becomes a new challenge.

C3: How to Enable High-level Programming Interface for FpgaNIC? The traditional programming interface on FPGAs use tedious, low-level, cycle-sensitive hardware description language (HDL), which hinders FPGAs from wide adoption by software programmers. Therefore, the programming interface of FpgaNIC cannot be HDL so as to attract more system programmers.

[4]The total offloading tasks (communicator and optimizer) in FpgaNIC do not need any GPU computing cycles that can be used for compute-intensive and memory-intensive forward and backward propagation.

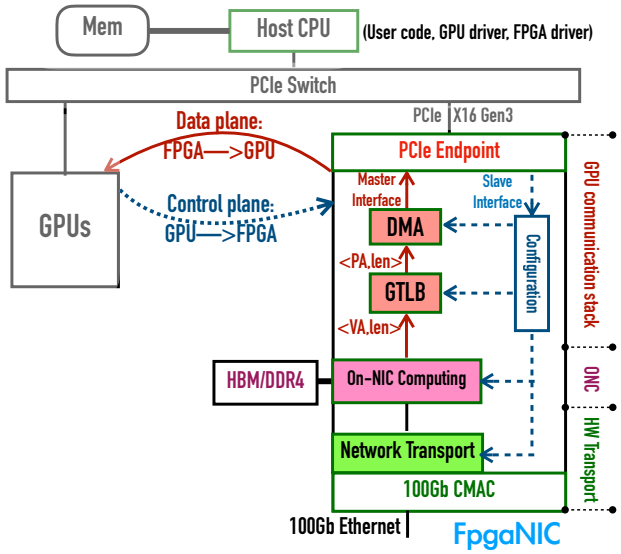


Figure 1: System architecture of FpgaNIC, which enables control/data plane offloading and reliable network transport offloading, and enables on-NIC computing module to process data from network at line rate. Moreover, FpgaNIC enables a large SmartNIC design space exploration.

C4: How to Enable Various SmartNIC Models? Based on the location of the smart function, SmartNICs can be categorized into three models: direct, on-path and off-path. A direct SmartNIC allows local GPUs to directly manipulate the network transport to realize, e.g., GPU-centric networking.

An on-path SmartNIC directly works on each network packet according to the corresponding smart function so that packets do not need to be staged, avoiding unnecessary additional latency in calling the smart function. However, its application scope is limited since it cannot handle complex functions as they are directly on the critical path of the network packets.

An off-path SmartNIC introduces an additional hop to implement the smart function using, e.g., a multicore CPU, which features two DDR4 channels for staging. This allows to map a broad range of applications on off-path SmartNICs. However, multicore CPU's memory bandwidth can easily become a performance bottleneck when processing 100Gb network traffic using the multicore CPU [40].

Different smart functions favor different SmartNIC models. For example, an on-path approach is preferred when offloading database's filter operator [61] while AllReduce [4] is better mapped to off-path SmartNICs. Instead of using specialized SmartNICs, we argue for a flexible architecture that enables all this models.

2.2 Main Architecture of FpgaNIC

To address the above four challenges, FpgaNIC adopts a layered design to enable easy design space exploration for SmartNIC architectures dedicated for various distributed applications that run on distributed GPUs, while minimizing the development effort and increasing the overall system efficiency. FpgaNIC consists of three main components: *GPU communication stack*, *reliable network transport in hardware*, and *on-NIC computing* (ONC), as shown in Figure 1. The goal of the GPU communication stack is 1) to allow the FPGA to use GPU virtual address (**C1**) to directly access GPU memory via direct PCIe P2P data communication at low-latency and line-rate, and 2) to allow GPUs to initiate data transfers by using *doorbell registers* on the FPGA to avoid having to involve the host CPU in the invocation. The goal of reliable network transport in hardware is to provide a reliable, low-latency, and high-throughput network access to the local GPUs (**C2**). The goal of on-NIC computing is 1) to enable high-level programming interface, and 2) to enable three NIC models: direct, on-path, and off-path, such that FpgaNIC is able to benefit a broad range of GPU-powered distributed applications (**C4**).

2.3 GPU Communication Stack

Built on a PCIe IP core, e.g., Xilinx's UltraScale+ Gen3 x16, the GPU communication stack of FpgaNIC aims at enabling offloading the control plane onto GPUs (via a slave interface) and offloading the data plane onto the FPGA (via a master interface), such that the host CPU is bypassed.

2.3.1 Offloading Control Plane onto GPUs

In order to allow GPUs to directly access the FPGA's control and status registers, FpgaNIC needs to offload the control plane onto the GPUs.

How to Enable Control Plane Offloading? Enabling control plane offloading requires a hardware-software codesign approach. On the hardware side, we enable a PCIe BAR exposing a configurable FPGA address space at the PCIe IP core on the FPGA. On the software side, our implementation consists of a *GPU driver*, an *FPGA driver*, and *user code* that interacts with both drivers. The process consists of three steps. First, the FPGA driver uses the function *misc_register* to register the PCIe BAR with the Linux kernel as an IO device */dev/fpga_control*. Second, the user code uses the function *mmap* to map the device into the host address. Third, the user code adopts a CUDA (Compute Unified Device Architecture) memory management function to register the host address for use within a CUDA kernel [52]. With this, the GPU can directly trigger doorbell registers and poll status registers on the FPGA without CPU intervention.

What Control Plane Offloading can Do? After enabling control plane offloading, the doorbell/status registers that are instantiated by all the components (GPU communications stack, ONC, and network transport) have to be mapped into GPU virtual address space so that the GPU program is able to access these registers without CPU intervention. Moreover, it enables us to populate GPU TLB (GTLB) entries on the FPGA such that the FPGA can translate GPU virtual address to physical address before issuing a DMA read/write operation to GPU memory (§2.3.2).

2.3.2 Offloading Data Plane onto the FPGA

FpgaNIC needs to offload the data plane onto the FPGA to allow the FPGA to directly access the GPU memory. However, NVIDIA GPUDirect [52] allows direct PCIe P2P data communication using physical address. For the sake of easy programming, FpgaNIC needs to work on GPU virtual address, rather than physical address (**C1**). Via a GPU BAR window, Tesla GPUs expose all of their device memory space, e.g., 40GB, while Quadro GPUs typically expose 256MB memory space with 36MB reserved for internal use [52]. In order to allow the FPGA to access more GPU memory space, FpgaNIC needs to store all the related virtual to physical address translation entries. To minimize the overhead of translation, we intend to keep all the entries on on-chip memory. However, the 64KB GPU page size becomes the main challenge, because storing a great number of translation entries on the FPGA needs a large on-chip memory. For example, 32GB GPU memory needs 512K entries, far beyond the number the FPGA implementation can accommodate without hurting timing.

How to Enable FPGA to Efficiently Work on Virtual Address? To this end, we propose a GPU Translation Lookaside Buffer (GTLB) to perform address translation on the FPGA, while keeping the on-chip memory consumption reasonably low. The key motivation behind the design of GTLB is that even though a single contiguous virtual address space needs not be physically contiguous on GPU memory, it has high probability to be physically contiguous, especially at the granularity of 2MB. Therefore, we manually coalesce 32 consecutive 64KB GPU memory pages into a 2MB page if these 64KB pages are allocated to a contiguous portion of physical memory and aligned within the 2MB page. The GTLB consists of a *main TLB* and a *complementary TLB*. The process of populating the GTLB on the FPGA involves four steps, as shown in Algorithm 1. First, we pre-malloc GPU memory space using *gpuMemAlloc* for staging the GPU memory that will be accessed by the DMA engines on the FPGA (Line 1). Second, we pass the initial virtual address and length of this GPU memory to the GPU kernel function *nvidia_p2p_put_pages* to get all the <*VA, PA*> pairs for all the 64KB pages (Line 2), where VA refers to virtual address and PA refers to physical address. Third, we try to coalesce 64KB pages into 2M pages as aggressive as possible (Line 3). Fourth, we populate main and complementary TLBs (Lines 4-18) via the control registers exposed by the control plane offloading (§2.3.1). The main TLB provides the virtual to physical address translations for 2MB pages (Lines 7-10). If any 2MB page is not physi-

Algorithm 1: POPULATING GTLB

```
   Input  : init_addr: initial GPU virtual address
            len: length of GPU memory
   Output : TLB^main: main TLB
            TLB^comp: complementary TLB
   /* Step 1: Malloc GPU memory space.                                  */
1  init_addr = gpuMemAlloc(len);
   /* Step 2: Get <VA, PA> pairs of all the 64KB pages.                 */
2  <VA_64KB, PA_64KB> pairs = nvidia_p2p_put_pages (init_addr, len);
   /* Step 3: Coalescing 64KB pages to 2MB pages if possible            */
3  <VA_2MB, PA_2MB>  pairs <- <VA_64KB, PA_64KB> pairs;
   /* Step 4: Populating TLB^main and TLB^comp                          */
4  index = 0; /* Large page index                                       */
5  comp = 0; /* Base page index                                        */
6  for (pair in <VA_2MB, PA_2MB> pairs) do
7      if (pair is physically contiguous) then
           /* Update the TLB^main                                       */
8          TLB^main[index].pair = pair;
9          TLB^main[index].valid = 1;
10     end
11     else
           /* Update the TLB^comp                                       */
12         TLB^comp[32*comp+31 : 32*comp] = pair's 32 64KB pages;
13         TLB^main[index].valid = 0;
14         TLB^main[index].comp_offset = comp*32;
15         comp++;
16     end
17     index++;
18 end
```

cally contiguous, we store the corresponding 32 translations of 32 64KB pages in the complementary TLB, which provides 2048 entires for accommodating 64 such 2M pages (Lines 12-15).[5] As such, the total number of required entires for 32GB memory becomes 16K+2K=18K, significantly smaller than the previous 512K entries.

Fully-pipelined Translation Lookup. After the population, FpgaNIC is able to directly access GPU memory using on-line virtual to physical address translation. Given a virtual address, FpgaNIC first checks the corresponding entry in the main TLB to see whether it is continuous or not ($TLB^{main}.valid == 1$). If yes, FpgaNIC fetches the PA and feeds it into the DMA engine. If no, FpgaNIC will read the corresponding entry in the complementary TLB using the offset $TLB^{main}.comp\_offset$. We can observe that the proposed GTLB can easily achieve fully-pipelined translation lookup on the FPGA.

GTLB Miss/Eviction. Currently, we pre-populate TLB entries for each application, assuming that the FPGA only accesses certain range of GPU memory. When GTLB miss or eviction happens, we need to re-populate GTLB entries for successful GPU memory references. However, we suggest to instantiate multiple GTLBs to provide sufficient number of GTLB entries to eliminate potential GTLB misses and evictions on the FPGA, because each GTLB entry only occupies a row of a BRAM, indicating relatively low cost of storing GTLB entries on the FPGA.

[5] In our experiment, 2048 entires are far beyond enough.

2.4 100Gbps Hardware Network Transport

In order to address the second challenge (**C2**), FpgaNIC offloads the transport-layer network to the FPGA to provide a reliable and high-performance hardware network transport to the local GPUs. Fortunately, there is a growing amount of open-source FPGA-based 100Gb network stacks such as the TCP/IP stack of [57, 60] and the RoCEv2 stack used in [62]. Without loss of generality, FpgaNIC is built on the 100Gb TCP/IP stack [57, 60], which is able to support thousands of connections with external FPGA memory for buffering.[6] We have modified this stack to adapt it to the requirements of FpgaNIC's by modifying its interface to improve bandwidth utilization and allow local GPUs to directly control the network transport.

2.4.1 Efficient Decoupled Application Interface

The original application interface [25, 57, 60] requires a control handshake between the TCP stack and the application code before sending or receiving a network packet to or from the TCP stack. A control handshake takes from 10 to 30 cycles while the payload of a packet (up to 1460B) only takes up to 23 cycles, leading to low network bandwidth utilization. To reduce the overhead of the handshake, we introduce an efficient decoupled application interface that does not need the handshake and further overlaps the control handshake and the packet transfer, maximizing the network bandwidth utilization and easing programming.

Decoupled Sending Application Interface. The original sending interface only allows to send a data chunk at a time after a control handshake, where the size of a data chunk is up to 1460B. A data chunk and a TCP header constitute a TCP segment, which can be encapsulated into an IP packet before sending over Ethernet. The proposed decoupled interface gets rid of the handshake, overlapping the control handshakes with the data transfer. And it further allows to send data streams of up to 4GB in size by automatically splitting the data stream into the right size chunks without programmer's involvement in packetization.

Decoupled Receiving Application Interface. The original receiving interface informs the user logic through a valid notification when a TCP segment is available to be consumed, which then sends out the read request to the receiving interface. After 10 to 30 cycles, the TCP segment's payload will be available at the 64B-wide AXI (Advanced eXtensible Interface)-Stream interface and consumed by the user logic. Similar to the sending interface, the proposed decoupled interface gets rid of the handshake, and further overlaps handshake

[6] The TCP stack needs two 64KB fixed-sized buffers per connection, one buffer for incoming packets and the other for outgoing packets. Therefore, external FPGA memory is needed to support thousands of concurrent connections. However, if fewer than 10 concurrent connections are needed, the TCP stack of [57, 60] can implement the buffers using on-chip memory such that external FPGA memory could be saved for offloaded smart functionality. In this paper, FpgaNIC uses both versions.

Table 2: Resource Usage breakdown of FpgaNIC on U50.

	LUTs	REGs	RAMs	DSPs
Available	871K	1743K	232.4Mb	9024
GPU Commu. Stack	79K	103K	5.2Mb	0
100G HW Transport	101.3K	166.5K	23.4Mb	0
ONC: GPU-centric networking	14.5K	20K	24.6Mb	0
ONC: AllReduce	7.3K	10K	12.8Mb	0
ONC: Hyperloglog	19.5K	26K	7.1Mb	1104

and data transfer and assembles the complete data stream for each TCP connection without programmer's involvement in depacketization.

2.5 On-NIC Computing (ONC)

The on-NIC computing module sits between the GPU communication stack module and the 100Gbps network hardware transport module, so ONC can directly manipulate the other two modules to enable flexible design space exploration around GPU-centric SmartNICs. The key goal of on-NIC computing module is to 1) expose high-level programming interface for system programmer, and 2) enable three SmartNIC models for various GPU-powered distributed applications. In the following, we discuss the programming interface of ONC and how to enable three three models.

2.5.1 High-level Manipulation Interfaces of ONC

In order to address the third challenge (**C3**), FpgaNIC intends to raises the programming abstraction from HDL to high-level synthesis (HLS), i.e., C/C++, such that systems programmers are able to use C/C++ to manipulate FpgaNIC, rather than cycle-sensitive HDL.[7] In the following, we present the concrete manipulation interfaces for the GPU communication stack and hardware network transport modules.

Manipulation Interfaces of GPU Communication Stack. The GPU communication stack exposes two manipulation interfaces: a slave interface that allows GPUs to access FPGA's registers and a master interface that allows the FPGA to directly access GPU memory, as shown in Table 3.

The slave interface is a 4B-wide AXI-Lite interface (*axilite_control*), through which local GPUs directly access doorbell and status registers within FpgaNIC without CPU intervention. In FpgaNIC, we instantiate 512 doorbell registers and 512 status registers, each of which has its own PCIe address to allow individual access. We correspond a few doorbell and status registers to each *engine* from any of three components within FpgaNIC. The doorbell registers can be triggered by GPUs to manipulate the engine, and the status registers can be polled by GPUs to check the status of the engine.

The master interface consists of two command streams and two data streams. The two command streams are 96-bit-wide AXI-Stream interfaces (*dma_read_cmd* and *dma_write_cmd*) that provide the GPU virtual address and length to directly access GPU memory, where the length is up to 4G. The data from and to GPU memory is sent over the *dma_read_data* and *dma_write_data* data streams, which are 64B-wide AXI-Stream interfaces. For either GPU memory read or write operation, we need to configure the command stream and then work on the corresponding data stream, allowing programmers to easily access GPU memory.

[7] Nevertheless, ONC can be also programmed in HDL if necessary.

Table 3: Two interfaces of GPU communication stack

Type	Interface	Content
Slave interface	axilite_control	AXI-Lite interface for configuration
Master interface	dma_read_cmd	Dest. GPU virtual address, length
	dma_read_data	AXI data stream from GPU memory
	dma_write_cmd	Source GPU virtual address, length
	dma_write_data	AXI data stream to GPU memory

Table 4: Manipulation interface for the network transport

Type	Interface	Meaning
Data interface	tcp_tx_meta	Session ID, length
	tcp_tx_data	AXI data stream to remote node
	tcp_rx_meta	Session ID, length, IP, port, etc.
	tcp_rx_data	AXI data stream from remote node
Control interface	server_listen_port	A TCP listening port
	server_listen_start	Staring to listen
	client_conn_port	Destination port to connect
	client_conn_ip	Destination ip to connect
	client_conn_start	Start to connect to server
	conn_close_session	Destination session to connect
	conn_close_start	Start to close connection

Interfaces of Hardware Network Transport. The hardware network transport exposes two interfaces: *data interface* and *control interface*.

The data interface consists of a *sending interface* and a *receiving interface*. The sending interface consists of a metadata stream and a data stream. The metadata stream (*tcp_tx_meta*) is a 48-bit-wide AXI-Stream that provides a 4B-wide data length and a 2B-wide session ID that corresponds to a remote node. The data stream (*tcp_tx_data*) is a 64B-wide AXI-Stream to send payload stream. The receiving interface also consists of a metadata stream and a data stream. The metadata stream (*tcp_rx_meta*) is an 44B-wide AXI-Stream that provides session ID, length, IP address, port and close session flag. The data stream (*tcp_rx_data*) is a 64B-wide AXI-Stream to receive payload stream from remote node.

The control interface of the hardware transport is similar to that of the well-understood socket interface, which allows GPU programmers to easily leverage the network transport, with their meanings as shown in Table 4. We instantiate the corresponding doorbell and status registers, exposed through the PCIe's slave interface (§2.3), to allow local GPUs to directly manipulate or poll the 100Gb hardware network transport.[8] In summary, the network transport serves as a network proxy, through which the ONC module and local GPUs can access the network transport directly without CPU intervention, so as to address the second challenge **C2**.

[8] Inside the FPGA, the ONC module (§2.5) can also directly manipulate the hardware transport via these registers.

Table 5: Lines of code for each component of FpgaNIC

	Hardware	Software
GPU Commu. Stack	2.9K (Verilog/HLS)	0.7K (C++, CUDA)
100G HW Transport	15.3K (HLS)	
ONC: GPU-centric networking	1.0K (HLS)	0.5K (C++, CUDA)
ONC: AllReduce	2.7K (HLS/Verilog)	1.5K (C++)
ONC: Hyperloglog	1.6K (HLS)	0.3K (C++, CUDA)

2.5.2 How to Support Three SmartNIC Models?

In order to address the fourth challenge **C4**, FpgaNIC's on-NIC computing component allows system programmers to customize data-path engines between the GPU communication stack and the hardware network transport to accelerate various distributed applications. Table 2 shows that the previous GPU communication stack and network transport consume less than 20% FPGA resources on a mid-sized FPGA U50, so the on-NIC computing component has plenty of resources to realize complex data-path engines to accelerate various distributed applications. Moreover, the commercial FPGA board that features DDR4 (even HBM) is able to stage data from network or GPUs, and perform on-NIC computing on the data before feeding into GPUs or sending out to network.

Due to the reconfigurable nature of the FPGA, FpgaNIC can easily support various SmartNIC models: direct, on-path, and off-path, to benefit a broad range of distributed applications. Table 5 shows the lines of code for each component.

The direct model directly exposes the hardware network transport module to local GPUs via the GPU communication stack module, such that local GPUs can directly manipulate the network transport to do reliable network communication. An an example, we develop a GPU-centric networking to demonstrate the potentials of the direct model. Due to space limitation, we describe the detailed design and implementation of GPU-centric networking to the Appendix §A.1.

The on-path model is similar to the direct model that local GPUs directly manipulate the hardware network transport, except that the on-path model allows the network stream also to enter an on-path engine in the ONC component for the offloaded computation, where the on-path engine needs to consume the network stream at line-rate such that the on-path engine would not impede line-rate network traffic. We use the HyperLogLog (HLL) application [18, 33] as an example to demonstrate the power of the on-path model. The detailed design and implementation of HLL with FpgaNIC can be found in the Appendix §A.3.

The off-path model enables an off-path engine in the ONC component to directly manipulate the GPU communication stack and the hardware network transport such that FpgaNIC is able to orchestrate the data flow between all the three components. Typically, the off-path needs to stage data in on-board memory. We use the collective communication primitive AllReduce [4, 11, 50] as an example to demonstrate the power of the off-path model (§A.2). The detailed design and implementation of AllReduce with FpgaNIC is in the Appendix §A.2.[9]

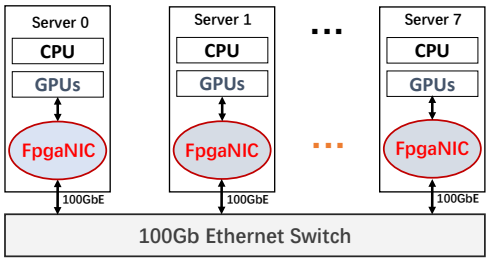


Figure 2: Experimental Setup

How to Support Multiple Tenants? To support multiple tenants, we can adopt Coyote [32] to wire the GPU communication stack and hardware network stack into the static region of FpgaNIC while exposing the same programming interface to offloaded tasks, for which we pre-synthesize the FPGA bitstreams ahead of time. Furthermore, FpgaNIC adopts the notion of *vFPGAs* (virtual FPGAs or separate application regions that are individually reconfigurable) as implemented in Coyote [32] to smoothly support secure, temporal and spatial multiplexing of GPU communication stack and hardware network transport between tenants (without pre-emption and context switching). For each tenant, FpgaNIC provides sufficient FPGA resources in a partial reconfiguration region to implement an independent ONC engine to guarantee performance isolation, and thus we no longer need to reboot the FPGA to change the functionality of FpgaNIC. We leave this as future work.

3 Experimental Evaluation

3.1 Experimental Setup

System Architecture. The experiments are run on a cluster consisting of eight 4U AMAX servers, connected with a Mellanox 100Gbps Ethernet SN2700 switch (Figure 2). Each server is equipped with two Intel Xeon Silver 4214 CPUs @2.20GHz, 128GB memory, FpgaNIC (i.e., a Xilinx UltraScale+ FPGA [72]), and a Nvidia RTX 8000 GPU, where the FPGA and the GPU have direct PCIe P2P communication, as shown in Figure 1. Two servers have an additional two A100 GPUs. FpgaNIC is implemented on Xilinx Alveo cards U50 or U280 with Vivado 2020.1.

Methodology. We first benchmark the GPU communication stack and hardware network transport to demonstrate that FpgaNIC allows easy PCIe P2P communication with local GPUs and reliable network communication with remote GPUs. We then evaluate the three FpgaNIC models: direct (§3.3), off-path (§3.4), and on-path (§3.5), to demonstrate FpgaNIC's performance and ability to enable the exploration of a large SmartNIC design space.

[9] The off-path model is generic enough such that it would also work well in other applications that follow a partition/aggregate pattern and require multiple rounds of communication [38].

3.2 Benchmarking Shared Infrastructure

We benchmark the shared GPU communication stack and hardware network transport.

3.2.1 GPU Communication Stack

To analyze the effect of control plane and data plane offloading, we measure the latency and throughput of the PCIe P2P link (§2.3). We use two classes of GPU: Quadro RTX8000 (labelled “R8K") and Tesla A100 (labelled “A100"), since a different GPU class leads to different latency and throughput.

Effect of Control Plane Offloading. We examine the effect of control plane offloading by comparing the latency of commands issued from the GPU to the FPGA. Figure 3 shows the read latency when using various end points: “X_Y" means that device “X" reads from device “Y". A first important result is that the latency of interactions between the GPU and the FPGA is comparable to that of the CPU calling the FPGA and it is under 1 microsecond. Moreover, the GPU-FPGA's latency fluctuation is smaller than that of CPU-FPGA, demonstrating one of the advantages of FpgaNIC in terms of offering deterministic latency. The results also show that performance improves slightly with a better GPU, indicating that the overall system will improve with future versions of the GPU. Finally, the latency of “GPU_FPGA" is significantly lower than that of “GPU_CPU" plus “CPU_FPGA", proving the efficiency of control plane offloading proposed in FpgaNIC.

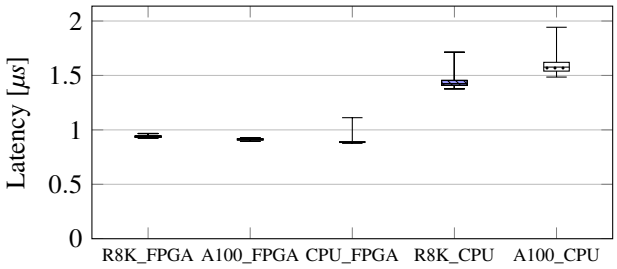


Figure 3: Control plane latency comparison. X_Y refers to the device “X" accesses the device “Y". R8K refers to RTX 8000 GPU, and A100 refers to A100 GPU. Whiskers show the 1st and 99th percentile.

Effect of Data Plane Offloading. We examine the effect of data plane offloading by measuring the throughput when the FPGA issues a DMA read/write operation to GPUs. Each operation transfers 4GB of data between the FPGA and the GPU memory. Figure 4 illustrates the achievable throughput with varying burst size, which is associated with the length of a DMA operation. It is interesting to observe that DMA read and write operations reach peak throughput at different burst sizes: 512B for read and 8K for write, indicating that we need to carefully choose the right DMA size to saturate the PCIe P2P bandwidth between FPGA and GPU. As with latency, a newer GPU class leads to much higher PCIe throughput. For example, a DMA read operation to an A100 GPU yields 12.6GB/s, close to the maximum possible PCIe bandwidth.

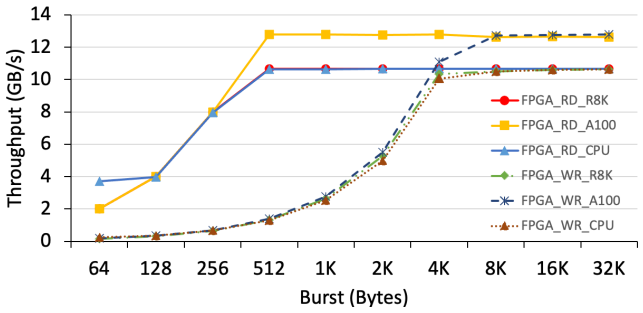


Figure 4: PCIe P2P throughput between FPGA and GPU

3.2.2 Hardware Network Transport

Next, we measure the throughput and latency of the hardware network transport (§2.4).

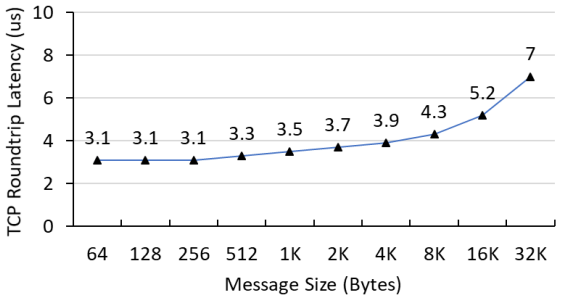
Latency. We measure the the round-trip time (RTT) between two FPGAs connected via the network switch. Figure 5a shows the RTT with varying message size. The most striking result is that the TCP latency is in microseconds, instead of milliseconds, demonstrating the advantages of offloading to a SmartNIC instead of using the CPU for communication. For messages smaller than 1 KB, the RTT latency (roughly 3.1us) is dominated by the physical communication path (the Ethernet switch introduces an additional hop with roughly 1us latency.

Throughput. We measure as well the throughput between two network transports with varying packet size and varying number of connections. Figure 5b shows the observed throughput by sending out a total of 1GB from one transport to the other with varying packet size and number of connections. We observe that the number of connections does not affect the achievable throughput under the same packet size, indicating that FpgaNIC is able to efficiently support multi-connection communication. For small packets, the throughput is low due to the fixed overhead, i.e., the 40B header, per packet and the turnaround cycles to process each packet. However, for larger packets, the achievable throughput is close to the 100Gbps channel capacity, demonstrating that FpgaNIC efficiently uses the available network bandwidth.

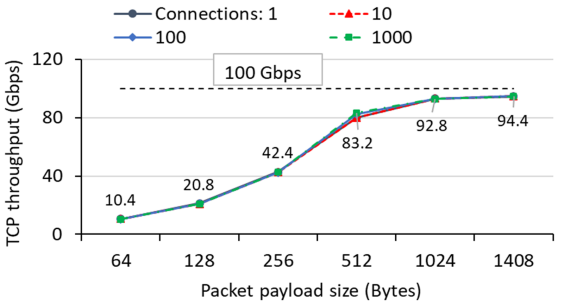
3.3 Evaluation of the Direct Model

We evaluate the throughput of FpgaNIC used in direct mode. The experiment involves sending data from one GPU to a remote GPU through the corresponding FPGAs using the direct model path: GPU-PCIe-FPGA-network-FPGA-PCIe-GPU.

Effect of Slot Size. We examine the effect of the slot size (W) of the circular buffer for each connection (§A.1.2). The slot size determines the size of the DMA operation between an FPGA and a GPU. Figure 6a illustrates the throughput with varying slot size. We have two observations. First, a sufficiently large slot size leads to saturated throughput. A



(a) Round-trip latency with varying message size



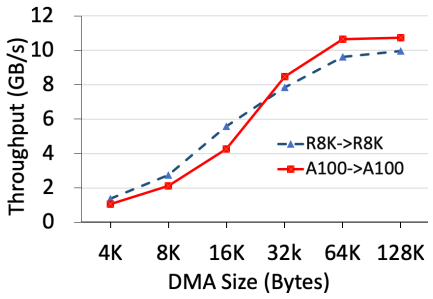
(b) Throughput with varying packet size and connections

Figure 5: 100Gb TCP stack: latency and throughput

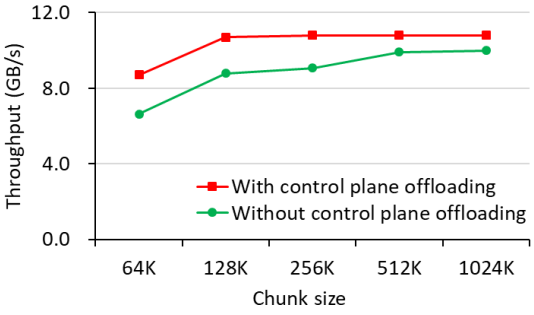
small slot size (<64KB) leads to lower throughput since it leads to low DMA engine utilization (Figure 4). Second, the network bandwidth between A100 GPUs is higher than that between RTX 8000 GPUs, as the slow PCIe speed between a RTX 8000 GPU and the FPGA becomes the bottleneck of network bandwidth (Figure 4).

Effect of Control Plane Offloading on Slot Size. We examine the effect of control plane offloading on different slot sizes. Without control plane offloading, we need to use CPU to trigger the DMA operation after executing a CUDA kernel that copies a chunk in the "GPU user" layer into the *send buffer* in the "GPU kernel" layer, leading to one kernel invocation per chunk. Intuitively, such frequent kernel invocations lead to significant overhead when the chunk size or the transfer size is not large. Figure 6b illustrates the throughput comparison with and without control plane offloading under different chunk size, when the data transfer size is 1GB. We observe that control plane offloading can leads to obviously higher throughput than the implementation without control plane offloading. Moreover, a smaller chunk size leads to higher throughput improvement, because control plane offloading eliminates more CUDA kernel invocations.

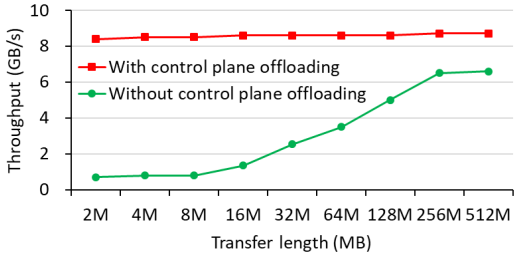
Effect of Control Plane Offloading on Transfer Size. We examine the effect of control plane offloading on different transfer sizes. Figure 6c illustrates the throughput comparison with and without control plane offloading under different transfer size, when the chunk size is 64KB. We observe that when the transfer length is smaller, control plane offloading leads to significant throughput improvement over the case without control plane offloading, whose performance is dominated by the kernel invocation overhead and context switch. In contrast, control plane offloading can remove these overheads by triggering doorbell registers from within a CUDA kernel, rather other from the host CPU.



(a) Effect of DMA size with control plane offloading



(b) Effect of control plane offloading on chunk size



(c) Effect of control plane offloading on transfer size

Figure 6: Throughput of GPU-centric networking

3.4 Evaluation of the Off-path Model

In this subsection, we evaluate the performance of FpgaNIC-enhanced AllReduce on a distributed pool of eight GPUs, as shown in Figure 2. When accelerating AllReduce, we configure FpgaNIC in an off-path model and offload the AllReduce engine to the FPGA. In the following, we present the baseline and the corresponding performance comparison.

Baseline. The experimental platform used as a baseline is similar to Figure 2, except that FpgaNIC in each server is replaced with a Mellanox ConnectX-5 100Gbps MT27800 NIC with RoCE and GPUDirect enabled. We use NVIDIA Collective Communication Library (NCCL) [50] which provides state-of-the-art collective communication primitives, e.g., AllReduce, over distributed Nvidia GPUs.

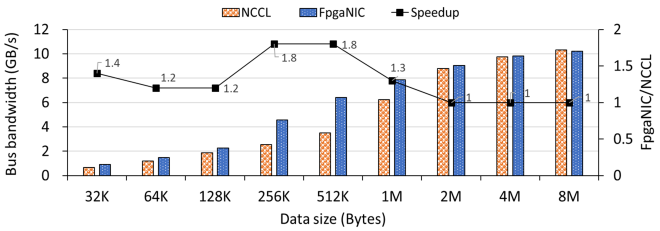


Figure 7: Effect of data size under eight nodes

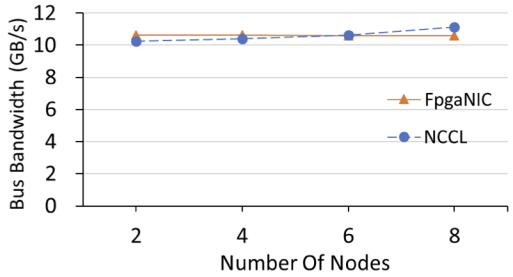


Figure 8: Effect of node number with 64MB data size

Comparison Metric. To demonstrate the performance of AllReduce, we introduce the metric *bus bandwidth* [51], which is calculated to be *algorithm bandwidth* times $2*(N-1)/N$, where algorithm bandwidth is calculated to be the data size divided by the elapsed time and N is the number of nodes. The elapsed time is estimated to be the average time of all the involved nodes in five rounds [9].

Effect of Data Size. We examine the effect of data size when performing AllReduce. Intuitively, a large data size easily leads to a saturated throughput because we hit the bandwidth limits of the underlying channels. Figure 7 illustrates the bus bandwidth comparison between FpgaNIC and NCCL in a cluster with 8 nodes. FpgaNIC leads to up to 2.5x speedup over NCCL, because the AllReduce engine in FpgaNIC efficiently overlaps the operations of the PCIe DMA, network transport, and FPGA memory. Moreover, FpgaNIC does not consume any GPU/CPU cycles, freeing up these precious computing resources for other important tasks. FpgaNIC reaches the theoretical bus bandwidth when the data size is larger than 8MB, indicating that FpgaNIC's AllReduce implementation is using all resources efficiently. Finally, when data size is small (<1MB), the speedup is up to 2.5x, due to the faster transition between states in FpgaNIC (Table 8) when compared to the same operation being implemented on GPUs.

Impact of Systems Size. We examine the effect of number of nodes on the AllReduce performance under 64MB data size. Figure 8 shows how both FpgaNIC and NCCL reach the theoretical bus bandwidth with an increasing number of nodes. However, NCCL needs a quite amount of CPU/GPU computing cycles to realize, while FpgaNIC does not.

Discussion. In the context of distributed AI model training, these results indicate that only offloading the AllReduce engine will not able to fully harvest FpgaNIC's potential. We can offload not only the communication functions (i.e., the AllReduce engine) but also part of the learning engine such as the compressor (e.g., compression engine) and optimizer (e.g., Adam engine) to FpgaNIC, such that the entire communication part of training is offloaded to minimize the communication overhead for GPUs. Since these engines can easily achieve line-rate throughput, plenty of interesting trade-offs in the design of distributed learning, e.g., sync vs. async, can be revisited by using FpgaNIC. We leave this idea to future work.

3.5 Evaluation of the On-path Model

We finally evaluate the performance of FpgaNIC-enhanced HLL, when FpgaNIC is configured in an on-path model. The cardinality is calculated when the data stream has been transferred. The goal of this experiment is to verify whether the HLL module within an FPGA can act as a bump in the wire.

The baseline, labelled "write", is to feed data to a GPU without processing the data in the FPGA. The GPU receives the data from the FPGA and stores it in the current *block* in GPU memory, while at the same time performing HLL on the previous block, overlapping data transfer with cardinality calculation at the block granularity. Table 6 illustrates that at least 8 SMs, in terms of 8 thread blocks and 512 threads per a thread block, are required to consume 100Gbps data stream (packet payload size: 1408 bytes) on an A100 GPU, when the block size is no smaller than 256K. Moreover, when the block size is smaller than 128K, an A100 GPU is not able to consume the data stream, as such a small block size cannot fully utilize GPU's processing parallelism.

Table 6: Number of required GPU SMs w.r.t block size

Block size	<=128K	256K	512K	1024K
Number of A100 SMs	>256	8	8	8

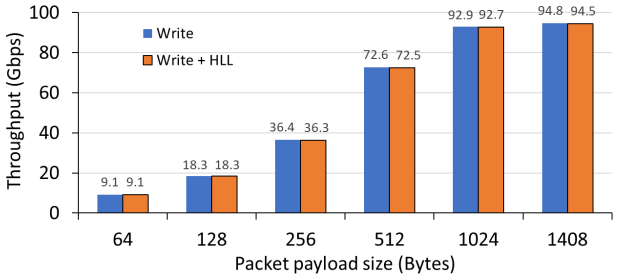


Figure 9: Performance of HLL with and without offloading. HLL with offloading does not affect the overall throughput, but saves at least 8 A100 GPU SMs, required by HLL without offloading, to consume 100 Gbps HLL data stream.

Figure 9 illustrates that FpgaNIC-enhanced HLL is able to achieve similar throughput as the baseline under various packet payload size, where the block size is 256K. It indicates that offloading HLL does not block the incoming data stream and introduces negligible latency. More important, FpgaNIC-enhanced HLL does not require any GPU compute power,

Table 7: Comparison of FpgaNIC with existing SmartNICs from industry. ✓ indicates full support, ✗ indicates no support.

	Programmable flow processing	Targeted applications	CPU-centric	GPU-centric
Broadcom [7]	✓	Virtualization, storage, NFV	✓	✗
Pensando [53]	✓	Storage, security	✓	✗
Netronome [49]	✓	SDN-controlled server-based networking	✓	✗
Intel IPU [23,24]	✓	Cloud, storage, security	✓	✗
FpgaNIC	✓	AI model training	✗	✓

e.g., at least 8 A100 SMs, which can be used in other computing task. Moreover, Table 2 shows that FpgaNIC-enhanced HLL takes a small amount of FPGA resources. Therefore, in the context of FPGA+GPU co-processing, it is clearly more efficient to offload HLL onto FpgaNIC, rather than processing HLL on the GPU.

4 Related Work

To our knowledge, FpgaNIC is the first FPGA-based GPU-centric 100Gbps SmartNIC that addresses the bottleneck limitations introduced by the use of conventional CPUs or small cores (ARM) in SmartNICs.

FPGA-augumented SmartNICs. Several commercial systems [10, 21, 22, 45, 55, 79] feature an FPGA within a SmartNIC. The closest work is from Mellanox Innova [45] that features an FPGA in its SmartNIC to accelerate offloaded compute-intensive applications, while PCIe and network interfaces are handled by a NIC ASIC ConnectX-5. The FPGA is connected with the NIC ASIC via a PCIe interface and therefore acts as an additional PCIe endpoint. The FPGA is entirely dedicated to the user's application logic. In contrast, FpgaNIC implements all functionalities, including networking and PCIe, within a powerful FPGA, enabling a large design space exploration of SmartNIC architecture, while Innova provides limited architectural flexibility due to how the FPGA is connected.

GPU-FPGA Communication. Previous work [6, 64] has implemented GPUDirect RDMA on an FPGA to directly access GPU memory, but not allowing the GPU to trigger doorbell registers within an FPGA. In contrast, FpgaNIC allows GPUDirect RDMA and the GPU to trigger registers within an FPGA, and is an FPGA-based SmartNIC that allows large design space exploration of SmartNIC architecture.

Acceleration using FPGA-based SmartNICs. Most previous work [2, 3, 8, 10, 13, 14, 14, 17, 26, 35, 36, 36, 37, 59, 62, 65] features an FPGA on SmartNICs to offload data processing to the network from the host CPU. In contrast, FpgaNIC is an FPGA-based full-stack SmartNIC that mainly targets compute task offloading from local GPUs which require a more complex system than offloading for CPUs. For example, we can offload partial G-TADOC [76, 77] that is a novel optimization to perform compressed data direct processing onto FpgaNIC to maximize the performance of distributed system under efficient FPGA-GPU co-processing.

Multicore-based SmartNICs. There is also a lot of work done [12, 16, 29, 41, 42, 44, 47, 48, 54, 56, 63, 66, 70] on SmartNIC built upon a wimpy RISC cores plus hardware engines to accelerate dedicated functionality such as compression. These RISC cores are used to both process packets as well as to implement "smart" functions instead of using the host CPUs. Such an approach inevitably suffers from load interference since packet processing and the smart functions have to compete for the shared resources, e.g., the last level cache and memory bandwidth. In contrast, FpgaNIC implements an GPU-centric SmartNIC on an FPGA.

On-going SmartNICs in Industry. Besides NVIDIA's DPU, we compare FpgaNIC with other SmartNICs from industry, as shown in Table 7. Broadcom offers the Stingray SmartNIC [7], which features a ARM 8-core CPU for control-plane management and P4-like TruFlow packet processing engine for data-plane processing, targeting various applications such as virtualization, storage, and NFV. Pensando has a DPU architecture [53] that features an ARM CPU and a P4 processor for data-plane packet processing, targeting various applications such as security and storage. Netronome provides the NFP4000 Flow Processor architecture [49] that features a ARM CPU, 48 packet processing cores, and 60 P4-programmable flow processing cores for data-plane processing, targeting the SDN-controlled server-based networking application. Intel presents the FPGA-based IPU (Infrastructure Processing Unit) that consists of a MAX 10 FPGA for control-plane management and an Arria 10 FPGA for data-plane processing [23], and uses an ASIC IPU whose architecture is not publicly documented [24]. Fungible has a DPU [15] featuring multiple PCIe endpoints, TrueFabric for networking, and specialized engines such as compression and EC/RAID to address inefficient data-centric computation within a node and inefficient interchange between nodes, targeting various applications such as virtualization, cloud storage, and data analytics. All these systems are CPU-centric in that they are designed to complement the CPU. In several cases, they suffer from the bottleneck problem pointed out above that prevents them from being above to operate at line rate. In contrast, FpgaNIC is an FPGA-based GPU-centric SmartNIC that targets various applications such as AI model training and security and specifically designed to operate at line rate which also means that it might not be suitable for operations that would significantly impair the flow of network packets (such as blocking operations or computations generating large amounts of intermediate state).

5 Insights and Implications of FpgaNIC

In this section, we discuss three interesting properties regarding FpgaNIC.

High Performance of On-NIC Computing Module. An increasing amount of SmartNIC solutions intend to remove the conventional CPU from the data-path (e.g., Microsoft Catapult). However, they either do not exploit the possibilities of direct communication between the FPGA and the GPU, or use small CPUs (ARM cores) that cannot process at line rate to impose additional hops within the NIC to implement the smart functionality (e.g., Bluefield-2). Furthermore, commercially available multi-core SmartNICs, such as BlueFiled-2, fail to process 100Gbps network traffic at line rate with its embedded CPU, which is capable of doing control-plane management only. The embedded CPU in Bluefield-2 is overwhelmed by trying to stage a 100Gbps data stream coming from the network. In contrast, FpgaNIC provides a 100Gbps data-path accelerator for distributed computing over GPUs, and thus enables a large design space exploration around SmartNIC for GPU-based applications. The key aspect of FpgaNIC is that it can process data at line rate as it comes from the network, something that other systems cannot do, because this requires to insert the accelerator in the data path, which cannot be done with conventional hardware (running conventional software) but can be done with FPGAs. To do so, FpgaNIC only consumes roughly 20% of the FPGA resources (marked in blue) to implement the NIC architecture (100Gbs hardware network transport and GPU communication stack) in a half-length, half-height FPGA board (Alveo U50), as shown in Table 5. It implies that the majority of the FPGA resources can be dedicated to on-NIC computing for SmartNIC functionality. Moreover, U50 has High Bandwidth Memory (HBM) which can be used to implement functionality with more intermediate states as memory access does not become the bottleneck. Therefore, FpgaNIC allows the offloading of compute-bound and memory-intensive tasks from multiple tenants (e.g., like in [39]) onto a mid-size FPGA.

Performance Guarantee and Isolation. Many multicore-based SmartNICs use small CPU cores for in-network computing. On these CPUs is really hard to provide performance guarantee and isolation due to insufficient CPU processing abilities and interference across tasks. We have shown that FpgaNIC is able to guarantee performance and isolation from two perspectives. From a compute's perspective, FpgaNIC provides dedicated hardware resources for each offloaded compute task, leading to a strict performance guarantee and perfect performance isolation. From a memory's perspective, U50 features 2-channel HBMs [71, 75] with 32 independent memory channels, each of which provides up to 13.6GB/s of memory throughput [20, 68]. This guarantees that each offloaded compute task is able to gain exclusive control over the assigned memory channels, without interfering with other offloaded compute tasks and the NIC infrastructure, which operates on dedicated hardware resources to guarantee line-rate network throughput.

Medium Programmability. Programming FPGAs using a Hardware Description Language (HDL), is error-prone and difficult to debug, limiting the adoption of FPGAs by system programmers. When using FpgaNIC, we intentionally ensure that it can be programmed using C++-based HLS (High Level Synthesis), to make it easier to use for software programmers, where HLS is the highest level of abstraction commercially available for programming FPGAs. To let FpgaNIC support both HDL and HLS, FpgaNIC's interface mainly leverages the stream type in HLS, i.e., AXI stream in HDL, for better compatibility. In future work, we intend to raise the level of abstraction further by developing a comprehensive framework such that users without hardware design experience can easily leverage FpgaNIC to accelerate distributed GPU-powered applications by automatically identifying offloaded functionalities via an FPGA-aware performance analysis framework [69] for maximum performance and high programmability.

6 Conclusion

Inspired by the fact that there is no SmartNIC designed for GPUs, we present FpgaNIC, a full-stack FPGA-based GPU-centric 100Gbps SmartNIC that allows a large design space exploration around SmartNICs for accelerating applications running on distributed GPUs. FpgaNIC enables direct data communication to local GPUs via PCIe P2P communication, enables local GPUs to directly manipulate the FPGA, provides reliable network communication with remote nodes, and enables on-NIC computing module to process the data from network at line rate. FpgaNIC can be efficiently used in three SmartNIC modes: direct, off-path, and on-path, to accelerate a broad range of GPU-powered distributed applications, such as Deep Learning model training. FpgaNIC is open-source to encourage further development and research in GPU-centric applications (Github: https://github.com/RC4ML/FpgaNIC).

Acknowledgement. We thank our shepherd and anonymous reviewers for their constructive suggestions. We are grateful to the AMD-Xilinx University Program for the donation of some of the AMD-Xilinx FPGAs used in the experiments. The work is supported by the following grants: the National Key R&D Program of China (Grant No. 2020AAA0103800), the Fundamental Research Funds for the Central Universities 226-2022-00151 and 226-2022-00051, Starry Night Science Fund of Zhejiang University Shanghai Institute for Advanced Study (SN-ZJU-SIAS-0010).

References

[1] Alexey Kopytov. sysbench. https://github.com/akopytov/sysbench, 2020.

[2] Catalina Alvarez, Zhenhao He, Gustavo Alonso, and Ankit Singla. Specializing the Network for Scatter-Gather Workloads. In *SOCC*, 2020.

[3] Mina Tahmasbi Arashloo, Alexey Lavrov, Manya Ghobadi, Jennifer Rexford, David Walker, and David Wentzlaff. Enabling Programmable Transport Protocols in High-Speed NICs. In *NSDI*, 2020.

[4] Baidu. baidu-allreduce. https://github.com/baidu-research/baidu-allreduce, 2016.

[5] M. Banikazemi, V. Moorthy, and D. K. Panda. Efficient collective communication on heterogeneous networks of workstations. In *ICPP*, 1998.

[6] R. Bittner and E. Ruf. Direct GPU/FPGA Communication via PCI Express. In *ICPP Workshops*, 2012.

[7] Broadcom. Stingray PS250 2x50-Gb High-Performance Data Center SmartNIC. https://docs.broadcom.com/doc/PS250-PB, 2019.

[8] Marco Spaziani Brunella, Giacomo Belocchi, Marco Bonola, Salvatore Pontarelli, Giuseppe Siracusano, Giuseppe Bianchi, Aniello Cammarano, Alessandro Palumbo, Luca Petrucci, and Roberto Bifulco. hXDP: Efficient Software Packet Processing on FPGA NICs. In *OSDI*, 2020.

[9] D. Bureddy, H. Wang, A. Venkatesh, S. Potluri, and D. K. Panda. Omb-gpu: A micro-benchmark suite for evaluating mpi libraries on gpu clusters. In Jesper Larsson Träff, Siegfried Benkner, and Jack J. Dongarra, editors, *Recent Advances in the Message Passing Interface*, 2012.

[10] A. M. Caulfield, E. S. Chung, A. Putnam, et al. A Cloud-scale Acceleration Architecture. In *MICRO*, 2016.

[11] M. Cho, U. Finkler, M. Serrano, D. Kung, and H. Hunter. Blueconnect: Decomposing all-reduce for deep learning on heterogeneous network hierarchy. *IBM Journal of Research and Development*, 2019.

[12] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. FaRM: Fast remote memory. In *NSDI*, 2014.

[13] Haggai Eran, Lior Zeno, Maroun Tork, Gabi Malka, and Mark Silberstein. NICA: An Infrastructure for Inline Acceleration of Network Applications. In *ATC*, 2019.

[14] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, et al. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *NSDI*, 2018.

[15] Fungible. FUNGIBLE F1 DATA PROCESSING UNIT. https://www.fungible.com/wp-content/uploads/2020/08/PB0028.01.02020820-Fungible-F1-Data-Processing-Unit.pdf, 2020.

[16] Stewart Grant, Anil Yelam, Maxwell Bland, and Alex C. Snoeren. SmartNIC Performance Isolation with FairNIC: Programmable Networking for the Cloud. In *SIGCOMM*, 2020.

[17] Zhenhao He, Dario Korolija, and Gustavo Alonso. EasyNet: 100 Gbps Network for HLS. In *FPL*, 2021.

[18] Stefan Heule, Marc Nunkesser, and Alexander Hall. HyperLogLog in Practice: Algorithmic Engineering of a State of the Art Cardinality Estimation Algorithm. In *CEDT*, 2013.

[19] T. Hoefler, S. Di Girolamo, K. Taranov, R. E. Grant, and R. Brightwell. sPIN: High-performance Streaming Processing in the Network. In *SC*, 2017.

[20] Hongjing Huang, Zeke Wang, Jie Zhang, Zhenhao He, Chao Wu, Jun Xiao, and Gustavo Alonso. Shuhai: A Tool for Benchmarking High Bandwidth Memory on FPGAs. *TC*, 2022.

[21] Intel. Intel SmartNICs for Telecommunications. https://www.intel.com/content/www/us/en/products/programmable/smart-nics-fpga-for-broadband-edge.html, 2020.

[22] Intel. Infrastructure Processing Units (IPUs) and SmartNICs. https://www.intel.com/content/www/us/en/products/network-io/smartnic.html, 2021.

[23] Intel. Intel FPGA Programmable Acceleration Card N3000 for Networking. https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/po/intel-fpga-programmable-acceleration-card/-n3000-for-networking.pdf, 2021.

[24] Intel. Intel Unveils Infrastructure Processing Unit. https://www.intel.com/content/www/us/en/newsroom/news/infrastructure-processing-unit-data-center.html#gs.bdzkbc, 2021.

[25] Zsolt István, David Sidler, Gustavo Alonso, and Marko Vukolic. Consensus in a Box: Inexpensive Coordination in Hardware. In *NSDI*, 2016.

[26] Wenqi Jiang, Zhenhao He, Shuai Zhang, Kai Zeng, Liang Feng, Jiansong Zhang, Tongxuan Liu, Yong Li, Jingren Zhou, Ce Zhang, and Gustavo Alonso. FleetRec: Large-Scale Recommendation Inference on Hybrid GPU-FPGA Clusters. In *KDD*, 2021.

[27] Yimin Jiang, Yibo Zhu, Chang Lan, Bairen Yi, Yong Cui, and Chuanxiong Guo. A Unified Architecture for Accelerating Distributed DNN Training in Heterogeneous GPU/CPU Clusters. In *OSDI*, 2020.

[28] Anuj Kalia, Dong Zhou, Michael Kaminsky, and David G. Andersen. Raising the Bar for Using GPUs in Software Packet Processing. In *NSDI*, 2015.

[29] Antoine Kaufmann, Simon Peter, Naveen Kr. Sharma, Thomas Anderson, and Arvind Krishnamurthy. High Performance Packet Processing with FlexNIC. In *ASPLOS*, 2016.

[30] Sangman Kim, Seonggu Huh, Xinya Zhang, Yige Hu, Amir Wated, Emmett Witchel, and Mark Silberstein. GPUnet: Networking Abstractions for GPU Programs. In *OSDI*, 2014.

[31] Diederik P Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[32] Dario Korolija, Timothy Roscoe, and Gustavo Alonso. Do OS abstractions make sense on FPGAs? In *OSDI*, 2020.

[33] A. Kulkarni, M. Chiosa, T. B. Preußer, K. Kara, D. Sidler, and G. Alonso. HyperLogLog Sketch Acceleration on FPGA. In *FPL*, 2020.

[34] N. T. Kung and R. Morris. Credit-based Flow Control for ATM Networks. *IEEE Network*, 1995.

[35] N. Lazarev, N. Adit, S. Xiang, Z. Zhang, and C. Delimitrou. Dagger: Towards Efficient RPCs in Cloud Microservices With Near-Memory Reconfigurable NICs. *IEEE Computer Architecture Letters*, 2020.

[36] 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 *SOSP*, 2017.

[37] Bojie Li, Kun Tan, Layong (Larry) Luo, Yanqing Peng, Renqian Luo, Ningyi Xu, Yongqiang Xiong, Peng Cheng, and Enhong Chen. ClickNP: Highly Flexible and High Performance Network Processing with Reconfigurable Hardware. In *SIGCOMM*, 2016.

[38] Y. Li, J. Park, M. Alian, Y. Yuan, Z. Qu, P. Pan, R. Wang, A. Schwing, H. Esmaeilzadeh, and N. S. Kim. A Network-Centric Hardware/Algorithm Co-Design to Accelerate Distributed Training of Deep Neural Networks. In *MICRO*, 2018.

[39] Jiaxin Lin, Kiran Patel, Brent E. Stephens, Anirudh Sivaraman, and Aditya Akella. PANIC: A High-Performance Programmable NIC for Multi-tenant Networks. In *OSDI*, 2020.

[40] Jianshen Liu, Carlos Maltzahn, Craig Ulmer, and Matthew Leon Curry. Performance Characteristics of the BlueField-2 SmartNIC. *arXiv preprint arXiv:2105.06619*, 2021.

[41] Ming Liu, Tianyi Cui, Henry Schuh, Arvind Krishnamurthy, Simon Peter, and Karan Gupta. Offloading Distributed Applications onto SmartNICs Using IPipe. In *SIGCOMM*, 2019.

[42] Ming Liu, Simon Peter, Arvind Krishnamurthy, and Phitchaya Mangpo Phothilimthana. E3: Energy-efficient microservices on smartnic-accelerated servers. In *ATC*, 2019.

[43] Yuanwei Lu, Guo Chen, Bojie Li, Kun Tan, Yongqiang Xiong, Peng Cheng, Jiansong Zhang, Enhong Chen, and Thomas Moscibroda. Multi-path transport for RDMA in datacenters. In *NSDI*, 2018.

[44] Mellanox. Mellanox BlueField SmartNIC. http://www.mellanox.com/related-docs/prod_adapter_cards/PB_BlueField_Smart_NIC.pdf, 2019.

[45] Mellanox. Mellanox innova-2flex. http://www.mellanox.com/related-docs/prod_adapter_cards/PB_Innova-2_Flex.pdf, 2020.

[46] Michael Cooney. Speed Race: Just as 400Gb Ethernet Gear Rolls Out, an 800GbE SPEC is Revealed. https://www.prnewswire.com/news-releases/400gbe-to-drive-the-majority-of-data-center-/ethernet-switch-bandwidth-within-five-years-/forecasts-crehan-research-300587873.html, 2020.

[47] YoungGyoun Moon, SeungEon Lee, Muhammad Asim Jamshed, and KyoungSoo Park. Acceltcp: Accelerating network applications with stateful TCP offloading. In *NSDI*, 2020.

[48] Craig Mustard, Fabian Ruffy, Anny Gakhokidze, Ivan Beschastnikh, and Alexandra Fedorova. Jumpgate: In-Network Processing as a Service for Data Analytics. In *HotCloud*, 2019.

[49] Netronome. NFP-4000 Theory of Operation. https://www.netronome.com/static/app/img/products/silicon-solutions/WP_NFP4000_TOO.pdf, 2016.

[50] Nvidia. NVIDIA NCCL. https://developer.nvidia.com/nccl, 2016.

[51] Nvidia. Performance reported by NCCL tests. https://github.com/NVIDIA/nccl-tests/blob/master/doc/PERFORMANCE.md, 2018.

[52] NVIDIA. Developing a Linux Kernel Module using GPUDirect RDMA. https://docs.nvidia.com/cuda/gpudirect-rdma/index.html, 2020.

[53] Pensando. Pensando DSC-25 Distributed Services Card. https://pensando.io/wp-content/uploads/2020/03/Pensando-DSC-25-Product-Brief.pdf, 2020.

[54] Phitchaya Mangpo Phothilimthana, Ming Liu, Antoine Kaufmann, Simon Peter, Rastislav Bodik, and Thomas Anderson. Floem: A Programming System for NIC-Accelerated Network Applications. In *OSDI*, 2018.

[55] Andrew Putnam, Adrian M Caulfield, Eric S Chung, et al. A Reconfigurable Fabric for Accelerating Large-scale Datacenter Services. In *ISCA*, 2014.

[56] Yiming Qiu, Qiao Kang, Ming Liu, and Ang Chen. Clara: Performance Clarity for SmartNIC Offloading. In *HotNets*, 2020.

[57] M. Ruiz, D. Sidler, G. Sutter, G. Alonso, and S. López-Buedo. Limago: An FPGA-Based Open-Source 100 GbE TCP/IP Stack. In *FPL*, 2019.

[58] Haichen Shen, Lequn Chen, Yuchen Jin, Liangyu Zhao, Bingyu Kong, Matthai Philipose, Arvind Krishnamurthy, and Ravi Sundaram. Nexus: a GPU cluster engine for accelerating DNN-based video analysis. In *SOSP*, 2019.

[59] Ran Shu, Peng Cheng, Guo Chen, Zhiyuan Guo, Lei Qu, Yongqiang Xiong, Derek Chiou, and Thomas Moscibroda. Direct Universal Access: Making Data Center Resources Available to FPGA. In *NSDI*, 2019.

[60] D. Sidler, G. Alonso, M. Blott, K. Karras, K. Vissers, and R. Carley. Scalable 10Gbps TCP/IP Stack Architecture for Reconfigurable Hardware. In *FCCM*, 2015.

[61] David Sidler, Zsolt István, Muhsen Owaida, and Gustavo Alonso. Accelerating Pattern Matching Queries in Hybrid CPU-FPGA Architectures. In *SIGMOD*, 2017.

[62] David Sidler, Zeke Wang, Monica Chiosa, Amit Kulkarni, and Gustavo Alonso. StRoM: Smart Remote Memory. EuroSys, 2020.

[63] Brent Stephens, Aditya Akella, and Michael M. Swift. Your Programmable NIC Should Be a Programmable Switch. In *HotNets*, 2018.

[64] Y. Thoma, A. Dassatti, and D. Molla. FPGA2: An Open Source Framework for FPGA-GPU PCIe Communication. In *ReConFig*, 2013.

[65] Yuta Tokusashi, Huynh Tu Dang, Fernando Pedone, Robert Soulé, and Noa Zilberman. The Case For In-Network Computing On Demand. In *EuroSys*, 2019.

[66] Maroun Tork, Lina Maudlej, and Mark Silberstein. Lynx: A SmartNIC-Driven Accelerator-Centric Architecture for Network Servers. In *ASPLOS*, 2020.

[67] Guanhua Wang, Shivaram Venkataraman, Amar Phanishayee, Nikhil Devanur, Jorgen Thelin, and Ion Stoica. Blink: Fast and Generic Collectives for Distributed ML. In *MLSys*, 2020.

[68] Z. Wang, H. Huang, J. Zhang, and G. Alonso. Shuhai: Benchmarking High Bandwidth Memory On FPGAs. In *FCCM*, 2020.

[69] Zeke Wang, Bingsheng He, Wei Zhang, and Shunning Jiang. A performance analysis framework for optimizing OpenCL applications on FPGAs. In *HPCA*, 2016.

[70] Xingda Wei, Rong Chen, and Haibo Chen. Fast RDMA-based Ordered Key-Value Store using Remote Learned Cache. In *OSDI*, 2020.

[71] Mike Wissolik, Darren Zacher, Anthony Torza, and Brandon Day. Virtex UltraScale+ HBM FPGA: A Revolutionary Increase in Memory Performance, 2019.

[72] Xilinx. XILINX ALVEO Adaptable Accelerator Cards for Data Center Workloads. https://www.xilinx.com/products/boards-and-kits/alveo.html, 2010.

[73] Xilinx. Alveo U280 Data Center Accelerator Card Data Sheet. https://www.xilinx.com/support/documentation/data_sheets/ds963-u280.pdf, 2019.

[74] Xilinx. Alveo U50 Data Center Accelerator Card Data Sheet. https://www.xilinx.com/support/documentation/data_sheets/ds965-u50.pdf, 2019.

[75] Xilinx. AXI High Bandwidth Memory Controller v1.0, 2019.

[76] Feng Zhang, Zaifeng Pan, Yanliang Zhou, Jidong Zhai, Xipeng Shen, Onur Mutlu, and Xiaoyong Du. G-TADOC: Enabling Efficient GPU-based Text analytics without Decompression. In *ICDE*, 2021.

[77] Feng Zhang, Jidong Zhai, Xipeng Shen, Dalin Wang, Zheng Chen, Onur Mutlu, Wenguang Chen, and Xiaoyong Du. TADOC: Text Analytics Directly on Compression. *VLDBJ*, 2021.

[78] Hanyu Zhao, Zhenhua Han, Zhi Yang, Quanlu Zhang, Fan Yang, Lidong Zhou, Mao Yang, Francis C.M. Lau, Yuqi Wang, Yifan Xiong, and Bin Wang. HiveD: Sharing a GPU Cluster for Deep Learning with Guarantees. In *OSDI*, 2020.

[79] N. Zilberman, Y. Audzevich, et al. NetFPGA SUME: Toward 100 Gbps as Research Commodity. *IEEE Micro*, 2014.

Appendices

A Implementation of FpgaNIC: Three Models

Due to the reconfigurable nature of FpgaNIC, the on-NIC computing component has been designed to easily support three SmartNIC modes for various types of application. We use an application to map to each of three SmartNIC models to demonstrate the versatility and efficiency of FpgaNIC.

A.1 Direct Model: GPU-centric Networking

We allow the direct model to support GPU-centric networking (GCN), which enables direct network communication between GPUs via a socket-like interface similar to that of GPUnet [30]. The design is based on four goals.

D1: Reliable Communication. FpgaNIC intends to serve AI and HPC applications which typically rely on reliable data communication between computing nodes. With the FPGA acting as a network proxy, this implies extending the functionality of FpgaNIC on the GPU side to avoid losing messages due to processing rate mismatches.

D2: Easy to Program. We strive to spare the developer the need to deal with tedious GPU-related optimization methods such as the number of thread blocks and the number of threads within a thread block. However, existing systems like GPUnet [30] invokes its send/recv calls within a thread block and requires all threads in a thread block to work in a coalesced manner [30]. As a result, a programmer needs to be quite familiar with GPU programming to leverage GPUnet. Moreover, RDMA programming is much more complex than traditional socket programming because RDMA exposes the underlying functions and data structures of NIC to allow programmers to manipulate. Therefore, FpgaNIC implements instead a simple socket API, making the development of distributed GPU applications easier.[10]

D3: Light-weight. Our GCN implementation has to be light-weight, in terms of low GPU memory footprint and low GPU core usage, on the GPU side, since the targeted HPC and AI applications running on GPUs are both compute-intensive and memory-bound. Thus, the overall design needs to free up GPU resources to maximize the application's performance.

[10]Nevertheless, FpgaNIC can also implement RDMA APIs with reasonably small amount of modifications.

```
//setup context for socket programming
context = create_socket_context();
int sockfd = socket(context); //create an socket

connect(); //connect to the server
send(sockfd, send_data, len); //send out an array
```

(a) Client

```
//setup context for socket programming
context = create_socket_context();
sockfd = socket(context);
listen(sockfd);
newsock = accept(sockfd, ...);
recv(newsock, recv_data, len)//recv an array
```

(b) Server

Figure 10: A typical example between a sender and a receiver

D4: Generalization to GPU Classes. Different GPU generations have different specifications for GPUDirect [52]: maximum BAR size and PCIe P2P bandwidth. The maximum BAR size available for GPUDirect is 256MB on Kepler-class GPUs, while at least 16GB on Tesla-class GPUs. Besides, the PCIe P2P bandwidth is 10.6GB/s on Quadro GPUs while 12.6GB/s on Tesla GPUs (§3.2). Our goal is to provide a solution that works on all GPUs regardless of their specifications.

In the following, we describe an example GPU-centric networking (§A.1.1), followed by the overall architecture (§A.1.2) and the implementation details (§A.1.3 and §A.1.4).

A.1.1 GCN Example

Figure 10 illustrates a typical example, written with the proposed socket-like APIs, between a client and a server running on distributed GPUs. After performing a typical socket handshake, the client GPU sends the data of length *len*, starting from the GPU memory address *send_data*, to the server GPU, starting from the GPU memory address *recv_data*. This example is similar to a typical socket program running on CPUs, except the need to set up a context for the inter-GPU socket programming. The resulting code is concise and easy to understand, satisfying one of the stated goals (**D2**: Easy of Use).

A.1.2 Overall Architecture of GCN

Figure 11 shows the software/hardware co-design approach chosen to implement the GPU-centric network model. The overall architecture consists of three layers: ONC, GPU kernel, and GPU user. The key idea is to aggressively overlap the operations performed on these three layers such that the overall performance is maximized.

ONC Layer. The "ONC" layer of FpgaNIC consists of the *control*, *DMA read*, and *DMA write* modules. The *control* module directly accepts control-plane commands from the GPUs and calls the other modules, e.g., *DMA read*. The *DMA read* module accepts a DMA read command from the *control* module and then issues a DMA read operation to the GPU. Next, the *DMA read* module forwards the data stream from PCIe to the 100Gb TCP stack. The *DMA write* module polls on the incoming stream interface from the 100Gb TCP stack and then forwards the received data to the GPU by issuing a DMA write operation.

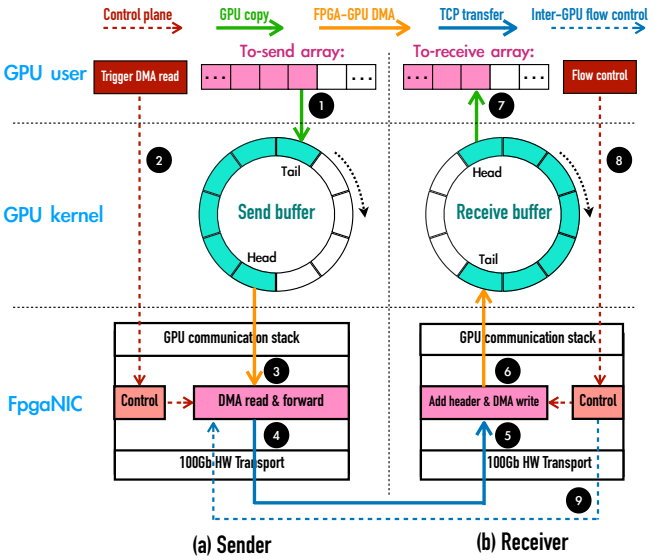


Figure 11: GCN between a sender and a receiver

GPU Kernel Layer. To manage incoming and outgoing traffic, we implement a *send buffer* and a *receive buffer*, conceptually two circular buffers, for each established connection. The key role of either buffer is to provide a staging option at the GPU memory exposed to other PCIe devices, as not all the memory space is visible to other PCIe devices on Quadro GPUs. Since the total exposed memory size is 220MB, we allocate 100MB to the send buffers and 100MB to the receive buffers, while the remaining 20MB is reserved for internal use. The number of supported connections is N, so each connection has a circular buffer of size $M = 100/N$ MB. We also split a circular buffer into F slots, each of which containing $W = \frac{M}{F}$ MB GPU memory space.[11]

GPU User Layer. In the GPU user layer, calling a send() or a receive() function will launch a *data-mover* kernel that leverages Streaming Multiprocessing (SM)[12] to move data between the GPU user and kernel layers such that the speed of the data mover matches the DMA read/write speed.

A.1.3 Handshake Protocol of GCN

We demonstrate how the handshake protocol works in GCN. The key idea is to directly leverage the TCP stack on the FPGA (§2.4) via the control plane offloaded to GPUs. The handshake process consists of the following three steps.

First, each side creates a GPU-aware context by calling *create_socket_context*, which specifies the number N of supported TCP connections (e.g., 8), the GPU send/recv buffer size of each connection (e.g., 12.5MB), the initial address of control plane.

[11] Nevertheless, FpgaNIC can be easily extended to support dynamic buffer size with slight modifications in the GPU kernel layer.

[12] In our experiment, a Streaming Multiprocessing (SM) provides more than enough throughput on both Quadro and Tesla GPUs. Each SM consists of 64 GPU cores. RTX8000 has 72 SMs while A100 has 108.

Second, each side creates a socket (*sockfd*) using the function *socket*, which launches a GPU kernel with only one thread that will apply for one free TCP connection slot in the FPGA 100Gb TCP stack.

Third, the server will listen to the socket *sockfd* by setting the listen-port register *listen_port* and then triggering the doorbell register *listen_start* (Table 4). Then, the server initiates the function *accept* to wait for an incoming connection from a client. The client calls the function *connect* with two parameters *conn_port* and *conn_ip* to specify the destination IP address and port. Once a connection is established, a client and a server can proceed to exchange data.

A.1.4 Send/Recv Functions of GCN

We now describe the implementation details of the two-sided communication between distributed GPUs by explaining the overall data and control flow shown in Figure 11.

Data Flow. The sender splits the "to-send array" into chunks, each of which has the size of W MB. For each chunk, we perform the following five steps. First, we employ a data-mover kernel that occupies a GPU SM, in terms of a thread block with 1024 threads, to copy a chunk in the "GPU user" layer into the "tail" slot in the *send buffer* in the "GPU kernel" layer (❶). Second, the sender kernel triggers a doorbell register within an FPGA to start a DMA read operation (❷). Third, the *DMA read* module reads the data stream from the "head" slot in the send buffer (❸), and then forwards to the 100Gb TCP stack (❹). Fourth, the receiver accepts the data stream from its 100Gb TCP stack (❺), and then adds a header and a trailer to the data stream and forwards it to the "tail" slot in the receive buffer in the "GPU kernel" layer (❻). Fifth, the receiving GPU kernel monitors the "head" slot and leverages a data-mover kernel that also occupies a GPU SM to copy the data in the "tail" slot to the destination chunk in the to-receive array in the "GPU user" layer (❼).

Flow Control. Reliable communication (goal **D1**) is achieved through a simple credit-based flow control [34] over each TCP connection, so as to avoid potential congestion at a slow receiving GPU receiving a heavy traffic load. At the beginning, the sender has a full credit of M MB to leverage. If we send data from the to-send array to the tail slot in the send buffer (❶), the corresponding credits are consumed, and the data in the send buffer will be safely delivered to the receive buffer on the other side. When the receiver copies the data from the head slot to the to-receive array in the "GPU user" layer, and then accumulates the amount M_r of correctly received data, where M_r is initialized to be 0. Once the ratio of M_r to M is over a threshold (Th), the receiver sends back a credit with $Th \times M$ bytes to the sender, indicating $Th \times M$ bytes of data have been correctly received. To do so, the receiver triggers a doorbell register (*consumed_bytes*) specified in the "control" module (❽), and then forms a flow-control packet to the sender (❾). After the sender receives the credits, the sender can proceed to send $Th \times M$ additional bytes.

Table 8: State transition of AllReduce within FpgaNIC. "x/y" means that x is input and y is output, where G refers to the communication with a GPU, E refers to the communication with the 100Gb TCP stack. "$(G+E)$" means that we perform the reduction on the data from GPUs (G) and the data from the 100Gb TCP stack (E), and store the reduced result in on-board memory. "(E,G)" means that the data is read from on-board memory and forwarded to the next GPU via 100Gb TCP stack (E) and GPUs (G). E_i^j indicates the *subarray*[i] has already been accumulated j times, where $1 \leq j \leq 4$. When j is 4, E^4 and G^4 are the final reduced result sent to the next GPU via 100Gb TCP stack and to local GPU, respectively.

	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7
FPGA 0	G_0^1/-	$(G_3^1+E_3^1)/E_0^1$	$(G_2^1+E_2^2)/E_3^2$	$(G_1^1+E_1^3)/E_2^3$	$E_0^4/(E_1^4,G_1^4)$	$E_3^4/(E_0^4,G_0^4)$	$E_2^4/(E_3^4,G_3^4)$	-/G_2^4
FPGA 1	G_1^1/-	$(G_0^1+E_0^1)/E_1^1$	$(G_3^1+E_3^2)/E_0^2$	$(G_2^1+E_2^3)/E_3^3$	$E_1^4/(E_2^4,G_2^4)$	$E_0^4/(E_1^4,G_1^4)$	$E_3^4/(E_0^4,G_0^4)$	-/G_3^4
FPGA 2	G_2^1/-	$(G_1^1+E_1^1)/E_2^1$	$(G_0^1+E_0^2)/E_1^2$	$(G_3^1+E_3^3)/E_0^3$	$E_2^4/(E_3^4,G_3^4)$	$E_1^4/(E_2^4,G_2^4)$	$E_0^4/(E_1^4,G_1^4)$	-/G_0^4
FPGA 3	G_3^1/-	$(G_2^1+E_2^1)/E_3^1$	$(G_1^1+E_1^2)/E_2^2$	$(G_0^1+E_0^3)/E_1^3$	$E_3^4/(E_0^4,G_0^4)$	$E_2^4/(E_3^4,G_3^4)$	$E_1^4/(E_2^4,G_2^4)$	-/G_1^4

Discussion. The design matches the goals we established the beginning, which dictate many of the architectural decisions. To ensure reliable communication (goal **D1**), we implement a credit-based flow control (§A.1.4) on the GPU side. To simplify programming (goal **D2**), the GPU-aware socket-like functions are executed sequentially by leveraging the default CUDA stream, which is transparent to programmers. Nevertheless, our APIs also allow programmers to explicitly specify CUDA streams to maximize execution overlap between kernels. To minimize overhead (goal **D3**), GCN uses at most 220MB of the GPU memory for the communication, and a GPU SM only when a send() or a receive() function is active. Moreover, each handshake function launches a GPU kernel with only one active thread. Finally, to support a wide range of GPU classes (goal **D4**), GCN only exposes 220MB GPU memory, which is allowed in all supported Nvidia GPUs.

A.2 Off-path SmartNIC: AllReduce

To illustrate the the off-path model of FpgaNIC, we implement a use case from HPC and AI applications: a collective communication primitive AllReduce [4, 11, 50] operating on the data residing in a distributed pool of GPUs. In particular, we implement a ring-based AllReduce algorithm [4, 50] as it provides high performance while having a simple communication flow that fits well within an FPGA. The communication pattern is as follows. Assume there are P GPUs and each GPU divides its own array for AllReduce into P subarrays. The p-th GPU receives *subarray*[i] from the $(p-1)$-th GPU, performs a reduction operation on the received *subarray*[i] and its local *subarray*[i], and then sends the reduced result to the $(p+1)$-th GPU, where $0 \leq i, p < P$.

A.2.1 Overall Architecture of AllReduce

The AllReduce [5, 67] engine implements the entire logic in the ONC component, which is configured in an off-path model, as show in Figure 12. The engine concurrently operates on three components on the FPGA: the PCIe DMA operation (§2.3), the network stack (§2.4), and the on-board memory. The overall execution under FpgaNIC allows to overlap the access to these components to maximize throughput.

PCIe DMA Operation. The PCIe DMA operation is used transfer data between the FPGA and its local GPU by issuing a DMA operation within an FPGA directly to the GPU and without CPU intervention.

Network Stack. The network stack is used to communicate with remote GPUs through their corresponding FPGAs. In our ring implementation, data arrives the previous GPU and it is sent to the next GPU in the ring.

On-board Memory. The FPGA on-board memory is used to store intermediate results. The current reduced result is accumulated in on-board memory before being forwarded to the next GPU. While it is being forwarded, the next result is being calculated. Thus, the memory needs to provide sufficient bandwidth for simultaneously writing partial results and reading the previous result as it is being sent.

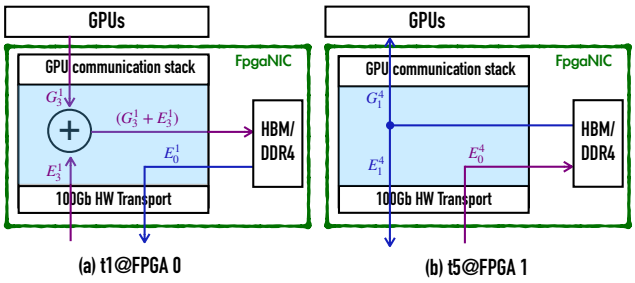


Figure 12: Architecture of AllReduce on the off-path model

A.2.2 Execution Flow of AllReduce on FpgaNIC

Table 8 illustrates the detailed execution flow of FpgaNIC-enhanced AllReduce with a concrete example over 4 distributed nodes, labelled FPGA i, where i is from 1 to 4. The execution is divided into eight steps ($t_0 \sim t_7$).

At step t_0, FPGA i issues a DMA read operation to transfer its local *subarray*[i] in GPU memory to the FPGA's memory (labelled G_i^1). At step t_1, FPGA i issues a DMA read operation to read from its local *subarray*[i-1] (G_{i-1}^1) in GPU memory, receives E_{i-1}^1 from the previous GPU in the ring (i.e., arriving via the network), and then accumulates these two on-the-fly and finally stores the accumulated result in the FPGA memory (labelled $(G_{i-1}^1+E_{i-1}^1)$). At the same time, FPGA i

forwards its local *subarray*[i] in FPGA memory to the next GPU, labelled E_i^1. Figure 12(a) illustrates the data flow of FPGA 0. At step t_2, FPGA i performs the reduction operation on its local G_{i-2}^1 in GPU memory and E_{i-2}^2 from the previous GPU, and then stores the accumulated result in FPGA memory (labelled $(G_{i-2}^1 + E_{i-2}^2)$). At the same time, FPGA i forwards its local E_{i-1}^2 from the FPGA memory to the next GPU. At step t_3, FPGA i performs the reduction operation on G_{i-3}^1 from its local GPU memory and E_{i-3}^3 from the previous GPU, and then stores the accumulated result in FPGA memory, labelled $(G_{i-3}^1 + E_{i-3}^3)$. At the same time, FPGA i sends its local E_{i-2}^3 from FPGA memory to the next GPU in the ring. At step t_4, FPGA i receives E_i^4 from the previous GPU and copies it to FPGA memory. At the same time, FPGA i sends $(G_{i-3}^1 + E_{i-3}^3)$ from the FPGA memory to the next GPU E_{i-3}^4 and writes it to its local GPU memory G_{i-3}^4. At step t_5, FPGA i receives E_{i-3}^4 from the previous GPU and copies into the FPGA memory. At the same time, FPGA i sends E_i^4 in the FPGA memory to both the next GPU E_i^4 and writes it to its local GPU memory G_i^4. Figure 12(b) illustrates the data flow of FPGA 1. At step t_6, FPGA i receives E_{i-2}^4 from the previous GPU and copies it to on-board memory. At the same time, FPGA i sends E_{i-3}^4 to the next GPU E_{i-3}^4 and writes it to its GPU memory G_{i-3}^4. At the final step t_7, FPGA i writes E_{i-2}^4 from the FPGA memory into its GPU memory G_{i-2}^4 which now has the final aggregated result.

Comparison with AllReduce on Innova. To illustrate the advantages of FpgaNIC's design over existing commercial solutions, consider how the same AllReduce operation would work on Mellanox Innova. In Innova, the PCIe link connecting the FPGA to the rest of the system limits the overall throughput because the AllReduce engine on the FPGA is forced to interact with both the local GPU and the network through its PCIe X8 Gen4 endpoint. In such a design, the FPGA can consume data either from the GPU or from the network but not from both at the same time. We estimate that the overall throughput would be halved. Both Innova and FpgaNIC approaches do not involve any GPU cores during execution, freeing up GPU cores for other computing tasks.

A.3 On-path SmartNIC: HyperLoglog

To illustrate the on-path model of FpgaNIC, we use HyperLogLog (HLL) [18, 33] as an example application. HLL is widely used in data analytic applications to estimate the cardinality of data streams or of large data sets. In our case, HLL will work on the data as it flows from the network transport towards the GPU. The basic scenario is transferring data to be processed to the GPU and using the FPGA to compute the cardinality on the fly without adding any overhead.

The on-path model is similar to the direct model (labelled "direct"), except that the incoming data stream is forwarded to both the GPU and to the on-path module (HLL in this case) via the "op_in" port. We use an open source implementation of HLL [33] that is embedded into the ONC component (Figure 13). After the data stream has been consumed, the cardinality of the data set can be forwarded to the local GPU via the "op_out" port. The on-path model also allows the result to be sent back to the network through the "op_return" port. In such a case, FpgaNIC can be used as an independent, network attached accelerator that uses the on-path module to process data on behalf of a remote client.

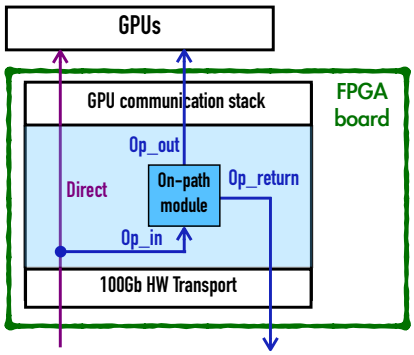


Figure 13: On-path model of FpgaNIC

7 Artifact

7.1 Abstract

This artifact provides the source code of FpgaNIC and scripts to reproduce the main experimental results. The experiments are run on a cluster consisting of eight 4U AMAX servers, connected with a Mellanox 100Gbps Ethernet SN2700 switch. Each server is equipped with two Intel Xeon Silver 4214 CPUs@2.20GHz, 128GB memory, FpgaNIC (i.e., a Xilinx Ultra-Scale+ FPGA), and a Nvidia RTX 8000 GPU, where the FPGA and the GPU have direct PCIe P2P communication. Two servers have an additional two A100 GPUs. FpgaNIC is implemented on Xilinx Alveo cards U50 or U280 with Vivado 2020.1.

7.2 Check-list

1. At least two nodes, each has a GPU that supports NVIDIA GPUDirect and the Xilinx U280 or U50 card.
2. Each FPGA card is connected to a 100Gbps Ethernet switch.
3. FPGA card and GPU are connected to the same PCIe switch.
4. Host OS: Linux 4.15.0-20-generic
5. Nvidia Driver Version: 450.51.05
6. CUDA Version: 11.0

Hugepages Setting. Make sure that each server has enabled Hugepages. If not, run the following commands.

1. $ sudo apt install libboost-program-options-dev cmake
2. $ sudo groupadd hugetlbfs
3. $ sudo getent group hugetlbfs
4. $ sudo adduser xxx hugetlbfs
   xxx is the user name you are using
5. Edit “/etc/sysctl.conf" and specify the number of pages you want to reserve.
6. $ mkdir /media/huge
7. Add this line “hugetlbfs /media/huge hugetlbfs mode=1770,gid=1001 0 0" to “/etc/fstab".
8. $ reboot

7.3 Thee Steps to Run Experiments

There are three steps to run each experiment. Before running FpgaNIC, please clone the source code:
$ git clone https://github.com/RC4ML/FpgaNIC

7.3.1 Hardware: FPGA Bitstream

1. $ mkdir build && cd build
2. $ cmake ..
3. Make HLS IP core
   $ make installip
4. Create vivado project, add the hardware project option after *make*, as shown in Table 9.
   $ make pcie_benchmark
5. Now the hardware project is produced, generate bitstream using vivado and flush it to every FPGA card.
6. Every time you download the bitstream to the FPGA, you have to reboot the machine, do not forget to reinstall xdma driver and GDR driver (See Subsection 7.3.2).

Table 9: The options of hardware project

Project	Description
direct	To create direct model project
pcie_benchmark	To create PCIe benchmark project
tcp_latency	To create TCP latency benchmark project
tcp_benchmark	To create TCP throughput benchmark project
allreduce	To create off-path model project
hyperloglog	To create on-path model project

7.3.2 Software: Driver Installation

1. $ cd FpgaNIC/driver
2. $ make && sudo insmod xdma_driver.ko
3. $ cd FpgaNIC/gdrcopy
4. $ sudo ./insmod.sh
5. Note that you need to reinstall xdma driver and gdr driver every time you reboot your machine.

7.3.3 Software: Running Application Code

1. $ cd FpgaNIC/sw && mkdir build && cd build
2. $ cmake ../src
3. $ make
4. $ sudo ./dma-example -b 0
5. $ Above command would report GPU read CPU memory latency, for more details, please refer to sw/README.md