Host Bypassing: Let your GPU speak Ethernet

Ralf Kundel*, Leonard Anderweit*, Jonas Markussen¶‡, Carsten Griwodz‡,
Osama Abboud§, Benjamin Becker*, Tobias Meuser*
*Multimedia Communications Lab, Technical University of Darmstadt, Germany
ralf.kundel@kom.tu-darmstadt.de
‡Department of Informatics, University of Oslo, Norway
¶Dolphin Interconnect Solutions AS, Oslo, Norway
§Huawei Technologies, Munich, Germany

2022 IEEE 8th International Conference on Network Softwarization (NetSoft) | 978-1-6654-0694-9/22/$31.00 ©2022 IEEE | DOI: 10.1109/NetSoft54395.2022.9844090

***Abstract*—Hardware acceleration of network functions is essential to meet the challenging Quality of Service requirements in nowadays computer networks. Graphical Processing Units (GPU) are a widely deployed technology that can also be used for computing tasks, including acceleration of network functions. In this work, we demonstrate how commodity GPUs, which do not provide any network interfaces, can be used to accelerate network functions. Our approach leverages PCIe peer-to-peer capabilities and allows the GPU to control the network interface card directly, without any assistance from the operating system or control application. The presented evaluation results demonstrate the feasibility of our approach and its performance of up to 10 Gbit/s, even for small packets.**

***Index Terms*—PCIe, GPU, NFV Offloading, Bypassing, DPDK**

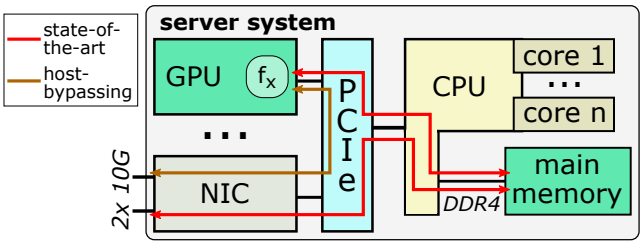

Fig. 1. Server system with PCIe-based periphery and accelerator integration. The red lines denote the data path in state-of-the-art systems, while the second path depicts the host bypassing approach.

I. Introduction

Computer networks have become the basis of many services in daily life over the last decades, and therefore the performance requirements are very high. For example, the total Internet utilization grows with a rate of around 25 % per year [1], implying constant increasing performance and flexibility demands on the underlying network.

Therefore, network functions in modern networks must have high performance while being flexible in terms of deployability and adaptability. Realizing network functions as a software component, named Network Functions Virtualisation (NFV), provides the aforementioned flexibility; however, standard servers do not provide the required performance [2]. Hardware accelerators, *e.g.*, Field Programmable Gate Arrays (FPGA) and Graphical Processing Units (GPU), can be used to increase the performance [3]. By this, an improved Quality of Service (QoS) can be achieved, especially a higher throughput, a lower latency, and low deterministic jitter.

A high and ensured end-to-end network QoS is of high importance for newly arisen applications, *e.g.*, softwarized radio access networks [4]. The network functions in 5G radio access networks must perform many compute-intensive tasks, including encryption and base-band channel encoding, at very high packet rates with low jitter and zero packet loss [5].

GPUs are one of the most common accelerator technology for these tasks as their internal architecture allows massive parallel packet processing [6], [7]. However, the main disadvantage of nowadays common GPUs is the lack of network connectivity as they provide only graphics outputs and at least one PCIe connector. PCIe is the current de-facto standard for integrating accelerator cards in computer systems [8]. Therefore, data is received by the Network Interface Card (NIC) in the main memory of a commodity server and subsequently copied into the GPU via PCIe using Direct Memory Access (DMA), as shown in Figure 1. Next, the network function running on the GPU is applied on the packet, denoted as f_x, in a highly parallelized manner. After completing the processing, the packet is transferred the same way back: The GPU writes the data into the system's main memory via DMA. After that, the NIC reads the packet from the main memory to transmit it on the network interface port. To summarize, the packet is copied four times within the system to be processed within the hardware accelerator, causing additional latency, jitter, and possibly limited bandwidth.

The main memory in data center servers is realized with DRAM memory, a technology building upon capacity-based bit storing. This technology, however, suffers from limited bandwidth and non-deterministic access times. Further, at least one software process running on the CPU is typically required to control the copy processes.

To overcome this, we presented the *host bypassing* approach in our previous work, allowing a direct interconnection between an FPGA and NIC in PCIe environments [9]. In this work, we aim to replace the FPGA with a commodity GPU. While FPGAs are special-purpose hardware allowing to realize almost any digital circuit at a high development effort compared to software processors, GPUs are much more

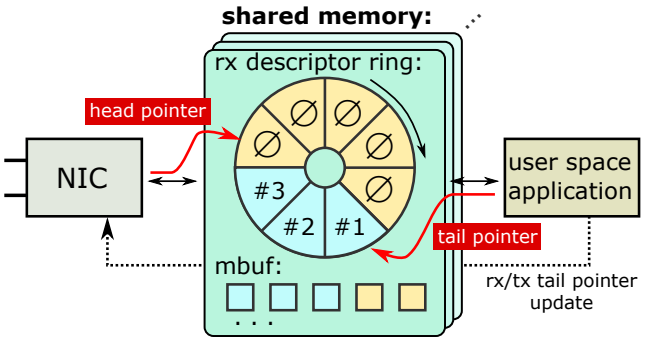

Fig. 2. Working principle of user space poll-mode NIC driver, communicating over a shared memory in the system's main memory with the application.

widely deployed and the development process is much simpler and very similar to conventional software engineering. In summary, we investigate the realization and performance of *host bypassing* with commodity off-the-shelf NICs and GPUs. This approach benefits from avoiding two memory copies per packet and bypassing the main memory.

The outline of this paper is as follows: First, we analyze the reception process of commodity NICs. Following this, we present our design for *host bypassing* with GPUs and present a detailed performance evaluation. Last, we present related work by other researchers and conclude the paper.

II. Background: User Space Poll-Mode Driver

Modern high throughput NICs are typically connected to a server system by a poll-mode driver instead of conventional interrupt-based packet handling, as the overhead of interrupt handling would be too high. The most prominent commercially maintained poll-mode driver collection is the Data Plane Development Kit (DPDK), which is open-source and the basis for this work [10]. In this section, we introduce poll-mode drivers in detail as a basis for the GPU-based *host bypassing*.

A. Poll-mode Drivers

Figure 2 depicts the working principle of NIC poll-mode drivers relying on a shared memory region accessible by the user space application and the NIC. Within this shared memory region, three different memories are realized:

Memory Buffer (mbuf): The largest memory area is used for storing packets to be sent or recently received. This memory region is aligned in constant-sized blocks, each storing a packet of the maximum allowed size, *e.g.*, $2048\ bytes$. The NIC and the user space application can read and write packets directly via DMA into this memory.

In addition to this, at least two ring buffers are used for handing over memory: The **rx descriptor ring** is a data structure allowing to hand over memory addresses between the NIC and user space application by a defined descriptor format. First, the application allocates at least one memory slot in the *mbuf* and writes these memory addresses into the according *rx descriptor ring entries*. Second, the NIC reads this address, and the next ingressing packet will be stored on the memory address in the *mbuf*. After storing the complete packet, the NIC overwrites the descriptor ring entry with multiple packet metadata, *e.g.*, receive timestamp and packet length, and the information that a packet is received. In parallel, the user space application continuously polls this descriptor entry until a packet is received. After the reception, the packet can be processed within the application, and the descriptor ring entry will be re-initialized with a new *mbuf* entry as described in the first step.

The descriptor ring typically has 64, 128, or 256 entries, allowing to receive and process packets asynchronously. For this, the *head pointer* indicates the position in the descriptor ring which contains the physical address for the next receiving packet. The *tail pointer* indicates the position in the ring describing the packet to be processed next by the application. By this, the NIC can receive many packets which do not have to be processed immediately by the application. In the given example of Figure 2, three packets are currently received but not yet processed. To avoid overruns of the ring, *i.e.*, the NIC receives packets faster than the user space application processes them, the *tail pointer* is written into the NIC periodically. The NIC is not allowed to increase the *head pointer* if this would imply a collision with the *tail pointer*.

Analogous to the receiving process, the **tx descriptor ring** is responsible for the transmission of packets (not shown in the figure). Packets are first written into the *mbuf*, and in a second step, this memory address is written into a descriptor entry of the *tx descriptor ring*. By updating the *tx tail pointer* of the NIC, the new packet is advertised and can be read via DMA by the NIC. Note that the tail pointer updates for receiving and sending can be performed for every packet but do not have to, *e.g.*, only for every 8_{th} packet to lower the PCIe bus utilization is sufficient.

B. Parallel Packet Processing

To improve the performance further, NICs allow parallel packet handling. For this, N descriptor rings for receiving and sending packets can be used parallel. The NIC computes a hash value for each incoming packet based on a few packet header fields, including source and destination IP addresses and L4-ports. This hash value is used to determine in which ring the packet belongs into. Each ring can be processed by a different software process of the user space application without any synchronization to the other processes. For the sending of packets, parallelization can also be achieved with multiple rings. In this case, the distribution of packets into the rings can be arbitrary as the packets of all rings are sent on the same egress port.

III. GPU-based Prototype

In the following, we present the design and prototypical implementation building upon the NVIDIA RTX 4000 GPU and the CUDA programming API. Further, we chose the Intel 82599 NIC, one of the most common interface cards in today's servers, together with the DPDK framework. To enable further research on our innovative host bypassing approach for GPUs, we provide the source code and detailed build instructions

open-source[1]. In general, the *host bypassing* emulates the behavior of the software driver (compare Figure 2) and works as follows: 1) A control process on the CPU, being part of the CUDA program, allocates memory in the GPU, accessible via physical addressing on the PCIe bus. For this, we build upon the *GPUDirect* technology. 2) The control process configures the GPU with the physical address of the NIC to enable tail pointer updates. 3) Last, a second control process configures the descriptor ring addresses within the NIC pointing on the GPU memory. This control process builds upon the DPDK framework, providing access to the low-level registers within the NIC. For this work, only minor modifications of the given DPDK API methods were required.

After these initial steps, the GPU can receive and send packets directly via the NIC without any active CPU process interaction. Consequently, the CPU resources can be used differently, or energy can be saved as no user space poll-mode driver processes are required anymore.

In the following, we present some key features in detail.

A. Memory Management

The main principle of *host bypassing* is relocating the shared memory, which is used for the NIC to application communication, from the system's main memory into the GPU. For this, we allocate a single memory block in the GPU, providing sufficient capacity for all descriptor rings and the packet *mbuf*. However, by default, such memory can not be accessed from another PCIe endpoint by physical addressing. For this, we must compile and load a kernel module, enabling memory pinning. This kernel module provides a *pin_memory* function, which is called by the initialization process, handing over the virtual address of the memory block as well as the PCIe bus and device ID of the GPU. As a result, this memory region is pinned to a fixed address within the physical address space of the GPU, and this address is returned. Further, within the CUDA application, the driver must modify the attributes of the memory pointer to enable synchronous memory operations on this memory region by the NIC and GPU simultaneously. Last, we must ensure that all PCIe permissions are set properly. The GPU and NIC must be allowed to write into the physical address range of each other. For this, PCIe bus mastering must be enabled and we recommend disabling the I/O Memory Management Unit (IOMMU). Furthermore, if there is a PCIe switch on the data path (compare Section IV-B), PCIe access control services must be disabled.

B. Parallel Packet Handling

One or multiple descriptor rings can be utilized for receiving and sending packets. For this, we implemented two CUDA kernels for incoming and outgoing transmissions. The behavior of the receiving and sending program is based on the behavior of the DPDK reference driver, as described in Section II. We utilized two *CUDA streams*, allowing the parallel execution of these two kernels. Further streams for packet processing can be easily added into this eco-system if required. The rx and tx kernels have a 1-dimensional block ID, indicating which ring they belong to. Within the allocated memory, all descriptor rings are arranged in ascending order, and the particular ring can be addressed by the base address, the ring size, and the process ID of the kernel. With our implementation, we tested one to eight descriptor rings; however, to serve even faster NICs, *e.g.*, 100 $Gbit/s$ link speeds, a higher degree of parallelism might be needed. Modern GPUs are providing more than sufficient resources for this. As the single-thread performance of GPUs is much lower than on commodity CPUs, more than a single descriptor ring is likely required to achieve a high throughput.

[1]https://github.com/ralfkundel/HostBypassing

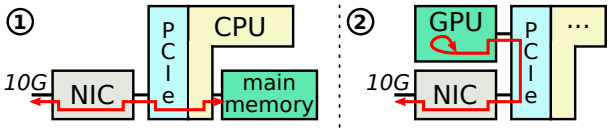

Fig. 3. Packet flow in the system under test for two evaluation scenarios.

Last, memory allocation must be performed within all rx kernel instances. Each rx kernel thread has its own memory allocation range in the global memory to circumvent any interference. The tx kernel is responsible for freeing memory after sending a packet; similarly, a network function that drops packets, *e.g.*, a firewall, must free the memory accordingly. Note that this allocation can not be realized with CUDA built-in functions. As stated above, the memory must be physically addressable from outside and set up before runtime. Therefore, we built an allocation mechanism, providing fixed-sized packet slots in the global memory, analogous to commodity drivers running on the host.

IV. Evaluation

In this section, we present the evaluation results of *host bypassing* with commodity NICs and GPUs, stressing the expected performance increase. Figure 3 depicts the two main evaluation scenarios we investigate. The first scenario describes state-of-the-art packet I/O from the NIC into the system's main memory and the same path back. In this scenario, neither *host bypassing* nor any GPU accelerator is involved. This scenario can be seen as an upper bound of theoretically achievable performance for state-of-the-art approaches with a GPU accelerator, as the packets must be copied first into the system's main memory before being transferred into the GPU. Note that this upper bound is not realistic, as the transfer from the main memory to the GPU and back in state-of-the-art GPU integration causes additional overhead. According to related work, in 2013 around $30\mu s$ for a single packet transfer between main memory and GPU can be assumed, and thus we still assume multiple microseconds for the latest GPU technologies [11].The second scenario transfers packets to the GPU using the *host bypassing* approach. As the goal of this work is packet I/O only, the GPU sends the packet back onto the same NIC port without any further packet processing.

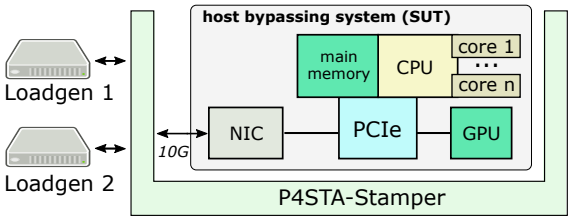

Fig. 4. Measurement setup for validating the *host bypassing* performance on GPUs utilizing the P4STA framework.

For evaluation, we measured latency, throughput, and packet loss of the *host bypassing* approach. To achieve a high measurement accuracy, we utilized the P4STA measurement framework, providing a nanosecond accuracy of up to 10 $Gbit/s$ link speed [12]. The general setup is shown in Figure 4 and consists of two load generation servers that create and validate test packets. The P4STA-Stamper is responsible for high-accurate latency measurement and packet loss detection. Further, as a single server can not generate a sufficiently high packet rate, the Stamper device can duplicate packets to achieve a high data rate even at small packet sizes. The System Under Test (SUT) is a commodity data center server with one Intel 82599 NIC and an NVIDIA RTX 4000 GPU. As mentioned in the respective sections, different CPU types from AMD and Intel are used.

A. General Performance

In the following, we investigate the general performance of the *host bypassing* approach and the DPDK baseline (Scenario 1 and 2). Figure 5 depicts the latency over time experienced by the packets passing through the test system. In all cases, the input rate was limited to 9.99 $Gbit/s$. The packet size, and by that, the packets per second rate, is varied between 300 $bytes$ and 1000 $bytes$. In addition to the DPDK baseline (Scenario 1) and the GPU *host bypassing* approach (Scenario 2), we added the equivalent evaluation results from our prior work on *host bypassing* with FPGAs (Scenario 2') [9]. In all tests, the GPU and NIC are connected to a PCIe switch (Broadcom PEX 8747), and no data is forwarded through the PCIe root complex of the CPU, as shown in Figure 6. We did not observe any packet loss in one of these four tests, each consisting of millions of packets.

First, we discuss the **DPDK baseline**, receiving and sending 300-$byte$ UDP packets. This packet size was chosen to stress the SUT and create more diverging results. We can observe that the latency is below 25 μs for most of the time; however, it periodically peaks up to $\geq 150\mu s$. We could not determine the reason for this, but it can be either caused by an interruption of the packet handling process, running on one of the CPU cores, or by accessing the main memory. A state-of-the-art system with GPU acceleration would have the same latency plus additional latency and jitter caused by transmitting the packet to and from the GPU accelerator.

The *host bypassing* results with FPGAs (named **FPGA bypassing** in the plot) show up a very low and constant latency,

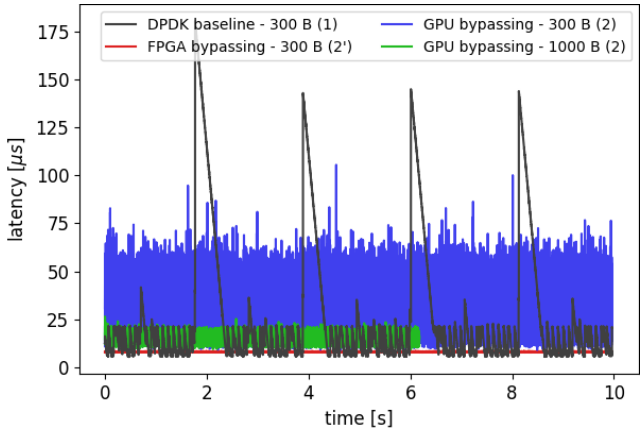

Fig. 5. Measured latency for different test scenarios at 9.99 $Gbit/s$ and 300/1000 $byte$ packet size. No packet loss occurred in all tests. GPU bypassing with 1000 $byte$ packets is stripped for readability reasons.

on average 7.99 μs. This is caused by neither involving the system's main memory nor any DRAM-based memory on the FPGA. Further, the FPGA processes incoming and outgoing packets with digital circuits built especially for this purpose. This allows the FPGA to detect and process new packets within a few nanoseconds only.

The *host bypassing* results with GPUs (named **GPU bypassing** in the plot) are generated with eight rings for parallel receiving and sending packets each. The latency for 300 $byte$ packets is significantly higher than for the DPDK baseline or the FPGA implementation. Nevertheless, zero packet loss occurred. As we can not observe the internals of the GPU at runtime, we could only speculate about the reasons, possibly caused by the global memory of the GPU, caching mechanisms, or the CUDA kernels running on them. Increasing the packet size to 1000 $bytes$, a more realistic average packet size in most computer networks, strongly decreases the jitter. However, the performance in terms of latency is still not comparable to the FPGA implementation. We assume that the internal architecture of GPUs, consisting of many processing cores and complex memory structures, is causing this non-deterministic processing behavior.

We can conclude that GPUs are currently no alternative to FPGAs for applications with very low jitter requirements, *e.g.*, less than 1 μs. Indeed, GPUs are very well suited for many compute intensive network functions and the general performance of GPU-based *host bypassing* is very good, *e.g.*, even for small packet sizes, zero packet loss occurs. This approach is superior by design to state-of-the-art techniques, performing network function offloading on GPUs.

B. PCIe Topology Influence

PCIe endpoints can be connected to the PCIe network in multiple ways. As shown in Figure 6, they can be either attached to the PCIe root complex of the CPU or to an external PCIe switch, increasing the total number of PCIe endpoints that can be attached to the system simultaneously. In the following, we consider two scenarios: Either the NIC and GPU are both connected to the CPU root complex (Intel Xeon Silver

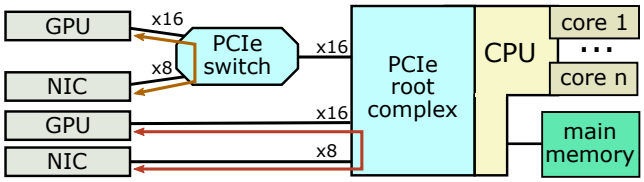

Fig. 6. Different PCIe topology scenarios evaluated in this work.

	avg. latency	latency std. dev.	packet loss
Broadcom PEX 8747	$544.18\mu s$	$99.68\mu s$	6.45%
AMD Epyc 7402	$564.67\mu s$	$94.75\mu s$	7.72%
Intel Xeon Silver 4110	$583.73\mu s$	$98.36\mu s$	9.82%

TABLE I
PERFORMANCE CHARACTERISTICS FOR 4 RX/TX RINGS AND 300 BYTE PACKETS AT 9.99 $Gbit/s$ DEPENDING ON THE PCIE INFRASTRUCTURE.

4110 and AMD Epyc 7402), or both are connected to a PCIe switch (Broadcom PEX 8747).

The results in Table I are created with 4 descriptor rings for receiving and 4 rings for sending packets. Compared to 8 rings, this causes packet loss in all scenarios and we can investigate the point-of-failure more precisely. In all cases, the input rate was a constant stream of packets with 300 $bytes$ each at 9.99 $Gbit/s$. The measured latency is a product of the maximum throughput and the receive buffer size of the NIC, which is constant. Consequently, a higher throughput causes a lower average latency. The packet loss is inverse to the throughput of the system, and the different PCIe architectures must cause the discrepancy between the runs. Consistent with our prior work [9], the PCIe switch has the best performance, followed by the AMD and Intel CPU. However, in contrast to the FPGA implementation, utilizing only a single ring for receiving and sending packets, the difference between the PCIe architectures is much lower. This might be either caused by an optimized PCIe interface of the GPU for common CPUs, or by the four parallel data transmissions which better utilize the PCIe root complex capabilities of the investigated CPUs.

C. Parallel Packet Handling

To improve the performance of *host bypassing* on GPUs, we enabled packet receiving and sending by up to 8 parallel descriptor rings each. Figure 7 shows the latency distribution for 1000 $byte$ packets at 7.4 $Gbit/s$, the maximum achievable rate without packet loss for all ring sizes. It is noteworthy that the latency and jitter are lowest for a single rx and tx descriptor ring. The higher jitter and latency might be caused by the multiple threads within the CUDA kernels accessing the same global memory interfering with each other.

The results in Table II show that for a single descriptor ring at maximum-sized packets, *i.e.*, 1500 $bytes$, and 9.99 $Gbit/s$ throughput, zero packet loss and no packet reordering occurs. Furthermore, the latency's standard deviation is very low (a reduction by $\sim$ 99%) compared to the aforementioned experiments with smaller packets sizes and multiple rings.

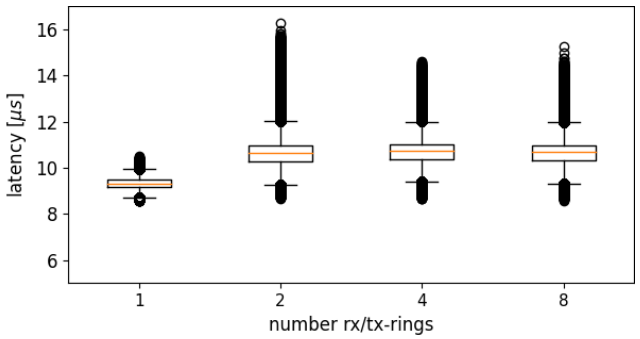

Fig. 7. Latency distribution in a non-overloaded scenario depending on the number of rx/tx rings. The 1000 $bytes$ test packets are shaped to 7.4 $Gbit/s$.

Rings	packet size	avg. latency	latency std. dev.	packet loss	reordered packets
1	1500	$9.76\mu s$	$87.7ns$	0	0
1	300	$1790\mu s$	$50.3\mu s$	74.78%	0.01%
2	300	$1000\mu s$	$87.7\mu s$	51.64%	49.49%
4	300	$544.2\mu s$	$99.7\mu s$	6.45%	42.28%
8	300	$27.17\mu s$	$7.6\mu s$	0	38.20%

TABLE II
GPU HOST BYPASSING PERFORMANCE CHARACTERISTICS DEPENDING ON THE NUMBER OF CONCURRENT THREADS (RX AND TX RINGS) FOR PACKET HANDLING BETWEEN NIC AND GPU.

To measure this, we captured each packet before and after the SUT and checked that the packet timestamps were strictly monotonously ascending. After decreasing the packet size to 300 $bytes$, massive packet loss occurs for a single ring. As the throughput in bits per second remained unchanged, we can determine the packet rate as the limiting factor, which is typical for virtual network functions. The number of reordered packets is close to zero; however, we observed a few reordered packets. This reordering might be caused by incorrect packet handling in this overloaded situation. When the number of descriptor rings is increased to two, the throughput is also almost doubled. This is a good indicator for the receive thread to be the bottleneck. Now, massive packet reordering can be observed, as two or more threads are sending packets asynchronously from each other. However, this can be the case for many other network functions that perform parallel packet processing as well, and is generally not a disadvantage. With 8 descriptor rings, we observed zero packet loss even for small packets but still high packet reordering.

To summarize, in order to achieve high throughput in GPUs, receiving and sending packets in parallel is mandatory. However, this causes massive packet reordering and is therefore improper for some networking applications. However, avoiding reordered packets may be possible at a high price of synchronization and is not required for most applications.

V. RELATED WORK

Utilizing PCIe-based hardware accelerators and their connectivity has been discussed in related work before.

Vasiliadis et al. presented a framework for GPU-based packet processing, providing basic network functionality, *e.g.*, TCP stream processing, to enable rapid and efficient applica-

tion development [13]. According to the authors, GPU processing can enormously improve the processing throughput; however, their measured latency of pure CPU-based processing is lower and therefore recommended for flows with low or strict latency requirements. We assume this problem can be solved by the *host bypassing* approach presented in this work.

The capabilities of PCIe peer-to-peer data transfers were investigated by *Bittner et al.* in 2014 [14]. The authors interconnected a GPU with an FPGA and demonstrated a significant reduction of the latency while transferring data.

Remote Direct Memory Access (RDMA) is a technology allowing to write directly into the main memory of a remote server by using special NICs and protocols on both sides. With this technology, it is also possible to write data directly from one server into the GPU main memory of a remote server over a network connection [15]. However, this technology is not applicable for network functions as it works only between two servers exchanging data on the application layer. *GPUDirect*, a technology also this work builds upon, allows the Direct Memory Access (DMA) transfers from one GPU to another without involving the system's main memory. Utilizing RDMA and GPUDirect technologies, data chains between multiple GPUs and CPUs in one or multiple servers can be built [16].

GPUDirect has also been used accelerating storage workloads [17]. In a fashion very similar to our own solution, *Markussen et al.* proposed a solution for leveraging PCIe peer-to-peer capabilities and using GPUDirect to operate a non-volatile memory device from a CUDA kernel [18].

VI. Conclusion and Future Work

Many network functions in current computer networks require very good performance in terms of latency, throughput, and jitter, along with high flexibility. By using general-purpose hardware accelerators, *i.e.*, GPUs, these goals can be achieved simultaneously. In this work, we investigated the direct connection between GPUs and Network Interface Cards (NICs) by utilizing PCIe peer-to-peer capabilities. We have shown that our approach, named *host bypassing*, is fully functional for receiving and sending packets over commodity NICs, still providing the same flexibility as existing state-of-the-art approaches. Our performance evaluation results have proven the potential of this approach to lower the end-to-end latency and jitter while providing very high throughput. The improvements can be explained by fewer memory copies within the system and bypassing the system's main memory, realized with non-deterministic DRAM memory technology. We observed no packet loss down to 300 $bytes$ packet size at 10 $Gbit/s$ throughput, corresponding to a packet rate of greater $4{\cdot}10^6$ pps, while measuring a deterministic low latency.

In future work, we will investigate the realization of 5G O-RAN functionality in GPUs together with *host bypassing* for packet I/O [19]. Especially the Central Unit (CU) in disaggregated O-RAN networks, responsible for the encryption of each packet sent to or received from the mobile subscriber, might benefit enormously from the combination of GPU acceleration and *host bypassing*.

Acknowledgment

This work has been funded by the Federal Ministry of Education and Research (BMBF, Germany) within the Software Campus Project "5G-PCI" and by the German Research Foundation (DFG) as part of the project B1 within the Collaborative Research Center (CRC) 1053 MAKI.

References

[1] Cisco Systems, Inc., "Cisco Annual Internet Report," 2020, [Online, last accessed: 28th February 2022].

[2] Y. Li and M. Chen, "Software-defined network function virtualization: A survey," *IEEE Access*, vol. 3, pp. 2542–2553, 2015.

[3] L. Nobach and D. Hausheer, "Open, elastic provisioning of hardware acceleration in nfv environments," in *2015 International Conference and Workshops on Networked Systems (NetSys)*. IEEE, 2015, pp. 1–5.

[4] C. Ranaweera, E. Wong, A. Nirmalathas, C. Jayasundara, and C. Lim, "5g c-ran with optical fronthaul: An analysis from a deployment perspective," *Journal of Lightwave Technology*, vol. 36, no. 11, pp. 2059–2068, 2017.

[5] M. Waqar, A. Kim, and P. K. Cho, "A transport scheme for reducing delays and jitter in ethernet-based 5g fronthaul networks," *IEEE Access*, vol. 6, pp. 46 110–46 121, 2018.

[6] W. Sun and R. Ricci, "Fast and flexible: Parallel packet processing with gpus and click," in *Architectures for Networking and Communications Systems*, 2013, pp. 25–35.

[7] A. Kalia, D. Zhou, M. Kaminsky, and D. G. Andersen, "Raising the bar for using gpus in software packet processing," in *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. Oakland, CA: USENIX Association, 2015, pp. 409–423.

[8] *PCI Express Base Specification 4.0*, Peripheral Component Interconnect Special Interest Group (PCI-SIG), 2017.

[9] R. Kundel, K. Eryigit, J. Markussen, C. Griwodz, O. Abboud, R. Hark, and R. Steinmetz, "Host bypassing: Direct data piping from the network to the hardware accelerator," in *2021 IEEE 14th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC)*. IEEE, 2021, pp. 23–30.

[10] The Linux Foundation, 2010, https://www.dpdk.org/.

[11] Y. Fujii, T. Azumi, N. Nishio, S. Kato, and M. Edahiro, "Data transfer matters for gpu computing," in *2013 International Conference on Parallel and Distributed Systems*. IEEE, 2013, pp. 275–282.

[12] R. Kundel, F. Siegmund, R. Hark, A. Rizk, and B. Koldehofe, "Network testing utilizing programmable network hardware," *IEEE Communications Magazine*, pp. 12–17, 2022.

[13] G. Vasiliadis, L. Koromilas, M. Polychronakis, and S. Ioannidis, "GASPP: A GPU-Accelerated Stateful Packet Processing Framework," in *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, 2014, pp. 321–332.

[14] R. Bittner, E. Ruf, and A. Forin, "Direct gpu/fpga communication via pci express," *Cluster Computing*, vol. 17, no. 2, pp. 339–348, 2014.

[15] B. Yi, J. Xia, L. Chen, and K. Chen, "Towards zero copy dataflows using rdma," in *Proceedings of the SIGCOMM Posters and Demos*, 2017, pp. 28–30.

[16] G. Shainer, A. Ayoub, P. Lui, T. Liu, M. Kagan, C. R. Trott, G. Scantlen, and P. S. Crozier, "The development of mellanox/nvidia gpudirect over infinibanda new model for gpu to gpu communications," *Computer Science-Research and Development*, vol. 26, no. 3, pp. 267–273, 2011.

[17] S. Bergman, T. Brokhman, T. Cohen, and M. Silberstein, "SPIN: Seamless operating system integration of peer-to-peer DMA between SSDs and GPUs," *ACM Transactions on Computer Systems*, vol. 36, no. 2, pp. 5:1–5:26, 2019.

[18] J. Markussen, L. B. Kristiansen, P. Halvorsen, H. Kielland-Gyrud, H. Stensland, and C. Griwodz, "Smartio: Zero-overhead device sharing through pcie networking," *ACM Transactions on Computer Systems*, vol. 38, no. 1–2, pp. 2:1–2:78, 2021.

[19] R. Kundel, T. Meuser, T. Koppe, R. Hark, and R. Steinmetz, "User plane hardware acceleration in access networks: Experiences in offloading network functions in real 5g deployments," in *Proceedings of the 55th Hawaii International Conference on System Sciences*. Computer Society Press, 2022, pp. 1–10.