GPU-Ether: GPU-native Packet I/O for GPU Applications on Commodity Ethernet

Changue Jung[†], Suhwan Kim[‡], Ikjun Yeom[‡], Honguk Woo[‡] and Younghoon Kim[‡*]
[†]Department of Electrical and Computer Engineering, College of Information and Communication Engineering
[‡]Department of Computer Science and Engineering, College of Computing and Informatics
Sungkyunkwan University, Suwon, Republic of Korea
Email: {jcgpk, tnghks9735, ikjun, hwoo and yhoon}@skku.edu

Abstract—**Despite the advent of various network enhancement technologies, it is yet a challenge to provide high-performance networking for GPU-accelerated applications on commodity Ethernet. Kernel-bypass I/O, such as DPDK or netmap, which is normally optimized for host memory-based CPU applications, has limitations on improving the performance of GPU-accelerated applications due to the data transfer overhead between host and GPU. In this paper, we propose GPU-Ether, GPU-native packet I/O on commodity Ethernet, which enables direct network access from GPU via dedicated persistent kernel threads. We implement GPU-Ether prototype on a commodity Ethernet NIC and perform extensive testing to evaluate it. The results show that GPU-Ether can provide high throughput and low latency for GPU applications.**

I. Introduction

Graphics processing unit (GPU) becomes one of the most popular accelerators in recent years. GPUs with massively parallel processing capability and large memory bandwidth are well suited to compute/memory-intensive applications ranging from accelerated network functions (NFs) [1]–[5] to distributed deep learning [6], [7] and various scientific computing, including research for COVID-19 [8]. Accordingly, most computing servers in modern data centers [9], [10] and high-performance computing (HPC) [11] environments are equipped with GPUs.

Many researches have been proposed to enhance GPU-based acceleration performance, and they reveal that, as GPU-based acceleration becomes intensified, networking overhead accounts for a more significant portion of the performance. Various network aspects have been examined to reduce the networking burden of accelerators (e.g., GPUs), including the use of proprietary network protocols and specialized NICs [12], offloading network I/O to SmartNICs [13] or Programmable NICs [14], optimization of network protocols such as MPI [15], and providing efficient network I/O on top of RDMA [16], [17].

Recent studies on network I/O optimization of GPUs often require specialized hardware such as RDMA HCA, NetFPGA, or SmartNIC. However, many data centers and cloud areas are still equipped with commodity Ethernet devices, and it is unlikely to immediately replace them with specialized hardware to improve performance. Instead, kernel-bypass I/O [18]–[20] can be a promising candidate, but using it for GPU requires complicated pipe-lining implementation and tuning efforts [1], [5].

In this paper, we propose *GPU-Ether*, a GPU-native packet I/O framework that enables direct network access from the GPU on top of the commodity Ethernet device. The proposed framework simplifies the GPU programming model by eliminating complex multi-staged pipe-lining. Moreover, it replaces heavy memory copy operations with P2P-DMA, thereby dramatically reducing network latency and completely excludes CPU intervention, which allows other host applications to fully utilize CPU resources. Benefits of GPU-Ether are as follows.

Low latency with direct network access: GPU-Ether shows significantly lower latency. In GPU-Ether, incoming network traffic is transferred directly to preallocated packet buffers in GPU memory through P2P-DMA, so a GPU application can immediately access data from the packet buffers without waiting for memory copy from its host memory. In an experiment comparing round trip time (RTT) of various networking methods for the GPU, GPU-Ether proves that it can achieve a minimum-level of latency achievable on the commodity Ethernet link (Section II).

No CPU intervention: GPU-Ether eliminates CPU intervention by letting dedicated GPU persistent kernel threads manage network traffic destined for the GPU itself. Thereby, CPU cores no longer need to perform boilerplate logic, such as packet I/O and data transfer between the host and the GPU. Another benefit of eliminating CPU intervention is that it minimizes the CPU cache pollution. Using the typical Linux network stack or kernel-bypass I/O, the cache pollution is inevitable because all network traffic to the GPU passes through host memory. In GPU-Ether, network traffic is delivered to the GPU directly and CPU cache pollution is reduced significantly (Section V).

A simpler model of GPU network programming: GPU-Ether's direct packet I/O scheme allows developers to implement high-performance GPU-accelerated network applications easily. Developers no longer have to spend time on tedious tuning and configuring complex pipe-lining to hide data transfer delays.

To provide the benefits mentioned above, GPU-Ether has to address the following major challenges in implementing packet I/O operating on the GPU.

[*]Younghoon Kim is the corresponding author.

Sequential processing within parallel computing architecture: The packet I/O requires sequential processing, but GPUs are devices designed for parallel processing. GPU maximizes its performance by executing a large number of GPU threads in parallel. GPU threads are scheduled in a 32 threads unit called *warp* (for Nvidia GPUs), and sequentiality is not guaranteed in warp scheduling. Meanwhile, commodity network interface cards hire descriptor ring structures for managing packet I/O, and their drivers assume sequential accesses to them. Careful consideration is needed when designing direct packet I/O running on the GPU to overcome this disparity.

Lack of thread-level locking: GPU-Ether uses a preallocated memory pool in GPU global memory, from which packet buffers can be allocated to the GPU threads. In the process of allocating the packet buffer, lock operations are usually required because packet buffer allocation may occur concurrently from multiple threads. However, GPUs only provide warp-level or thread-block-level locking rather than thread-level, which is required on our GPU-Ether. Moreover, according to [21], there is a high likelihood of causing livelocks by implementing thread-level synchronization to GPUs using the CPU synchronization mechanism.

Expenses due to frequent communication between CPU and GPU: The packet I/O operation requires a constant ring descriptor information update and doorbell register access. If these tasks are done on the host-side (e.g., NIC driver), the periodic exchange of information between CPU and GPU is required, which complicates implementation and increases overhead and latency.

Our GPU-Ether prototype is implemented on a system with an Nvidia Quadro P4000 GPU and an Intel 82599 X520-DA2 10GbE NIC. The GPU-Ether prototype demonstrates low round trip latency, and high throughput.

The contributions of this paper are as follows: (a) we present a packet I/O scheme operating on GPU which supports the direct network access from a GPU on the commodity Ethernet environment; (b) we develop several optimization techniques for enabling GPU persistent kernel threads to process network packets in line-rate; and (c) we prototype GPU-Ether for a 10GbE NIC and evaluate its I/O performance and applicability with several network applications, including software router, IPSec gateway, and NIDS.

II. Motivation

An earlier work [22] has investigated the efficacy of P2P-DMA between an Ethernet NIC and a GPU. However, they did not make the entire packet I/O using P2P-DMA and any performance comparison with other networking methods. To confirm the availability of P2P-DMA in the packet I/O, we partially implemented GPU-Ether based on P2P-DMA and then conducted simple ping-pong message experiments as follows.

Networking methods: In this experiment, we employ three different networking methods available in the commodity Ethernet as follows: (a) POSIX UDP socket + data transfer to/from a GPU; (b) DPDK (kernel-bypass I/O) + data transfer to/from a GPU; and (c) direct data transfer between a NIC and a GPU via P2P-DMA.

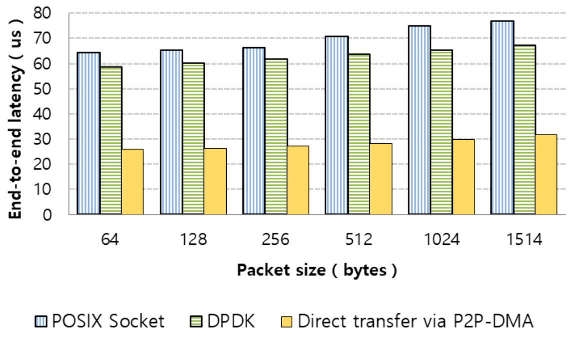


Fig. 1. Latency comparison of various networking methods for the GPU.

Client: A simple UDP client repeatedly sends 100 packets five times to collect the average RTT of each trial. Sending additional five packets to warm up and excludes their RTTs from the average. Also, to avoid the effect of *interrupt moderation* [23], packets are transmitted back-to-back without a time gap. We vary the packet size from 64 bytes to 1514 bytes with a random payload.

Servers: Servers for (a) and (b) operate as follows: receive a packet, conduct two data transfers with the GPU (H-to-D and D-to-H), and then send the packet back to the client. For (c), a server is running on a GPU. Two P2P-DMAs occur between the NIC and the GPU, and the GPU directly manages ring descriptors and doorbell registers of the NIC to send and receive packets.

Fig. 1 shows the average RTT of the three different networking methods. Among them, UDP socket (blue bar) shows the longest latency due to system calls and memory copies included in the network stack processing. DPDK (green bar) shows lower latency by avoiding the kernel network stack but includes two data transfers between the host and the GPU because network traffic must pass through the host memory. Direct transfer via P2P-DMA (yellow bar) shows the shortest latency in all packet sizes, which is consistent with the results in [5] claiming that data transfer between the host memory and the GPU memory is a major deterioration of GPU networking.

Typically, a pipe-lining technique that reduces the number of memory copies by batching a large number of packets is used to hide the GPU's data transfer delay. However, an optimal batch size depends on the packet size, the network bandwidth, and the GPU's processing power and it is not trivial to find one. During the experiment, we observe that DPDK uses 100% of two CPU cores for polling (master and slave lcore each), while P2P-DMA has no CPU core usage. We measure the time spent on the link and it is almost the same as the round trip time of P2P-DMA method (about 24 us). It implies that only little latency is added to the link delay for the P2P-DMA method while reflecting packets.

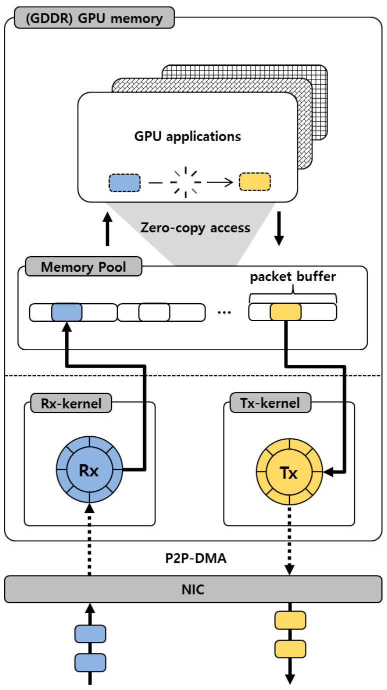


Fig. 2. Overall architecture of GPU-Ether. Rx,Tx: Ring descriptors.

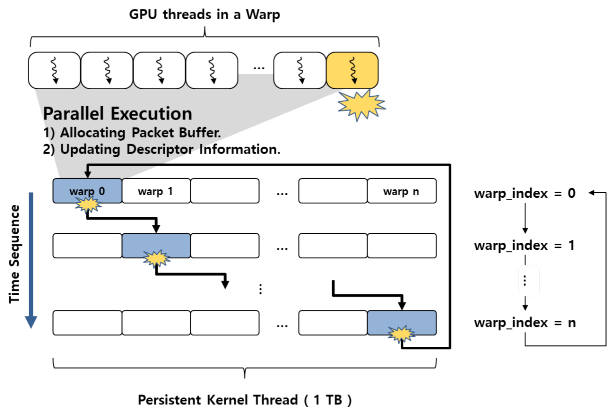


Fig. 3. Sequential Packet I/O processing in persistent kernel threads.

III. Design

In this section, we present an overview of the high-level design of GPU-Ether and then describe major components in detail.

A. Overall system design

Fig. 2 shows the overall architecture of GPU-Ether. Two persistent kernel threads in GPU are responsible for Rx and Tx, respectively. Received packets are transferred to packet buffers in the GPU memory via P2P-DMA and Rx-kernel updates the information of associated descriptors. By default, GPU applications can directly access the packet data in a zero-copy manner.[1] When a packet is ready to transmit, Tx-kernel updates the descriptor information related to packet transmission, sends out the packet with a doorbell register access and returns the packet buffer to the memory pool.

B. Main components

Memory pool and packet buffer: The memory pool in GPU-Ether is a set of available packet buffers. It is preallocated by GPU-Ether to hold ingress/egress packets within the GPU global memory. The DMA-able address of the memory pool is exposed to the NIC driver when initializing GPU-Ether and replaces the original DMA addresses with proper offsets. Then, it becomes possible for the NIC to forward network traffic to the GPU memory directly. The DMA-able buffer address for each descriptor must be newly specified on every packet departure or arrival, and the persistent kernel threads perform this task. A packet buffer structure is a fixed-size object containing metadata and a fixed-sized area for packet data.

Ring descriptors and doorbell registers: To perform packet I/O directly within a GPU, GPU-Ether maps Rx/Tx descriptors and doorbell registers of the NIC driver into the GPU memory using different mapping techniques for two different components. General *mmap* + CUDA mapping is used for doorbells and P2P-DMA is for descriptors. The reason for different mappings is explained in the next section. GPU threads in the persistent kernel are mapped one-to-one with descriptors, and they update the information of each dedicated descriptor. To minimize the overhead, only the last thread of each warp accesses the doorbell register. The doorbell is accessed after the warp number of packets are processed for batching while notifying the NIC.

Persistent kernel threads: Two persistent kernel threads are the key enablers of packet I/O operating on GPU. Their main tasks are to allocate and free packet buffers[2], update descriptor information and control the NIC through doorbell registers. The doorbell registers need to be sequentially accessed for NIC's operation procedure but a GPU warp scheduler schedules threads in an indeterminate order. It may cause abnormal behavior in packet retrieval and transmission.

To resolve this issue, we make only a warp unit of threads process packets at a time. The warp responsible for packet processing is rotated from the first warp to the last within the kernel to realize the sequential packet I/O process (Fig. 3).

C. Enhancing parallelism

Since the GPU architecture is optimized for parallel processing, sequential processing with just a warp-unit may cause performance degradation. To avoid this degradation, we introduce the concept of *warp_batch* which is the number of threads concurrently performing packet I/O (Fig. 4) and increase it to make multiple warp units run together. It gives

[1]Data can be copied into the application buffer depending on purposes.

[2]Allocating a packet buffer means extracting it from the memory pool, and freeing means the opposite.

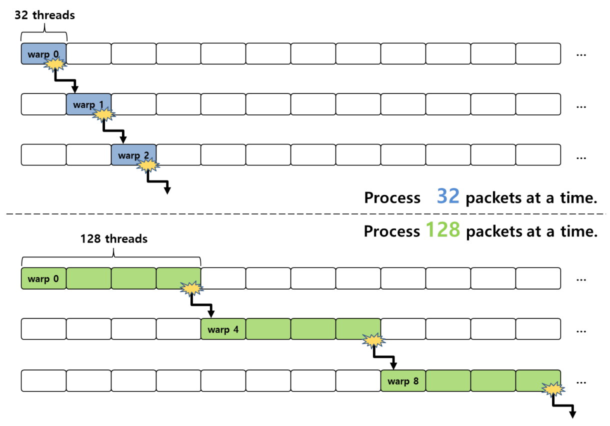


Fig. 4. Enhancement of parallelism in packet processing.

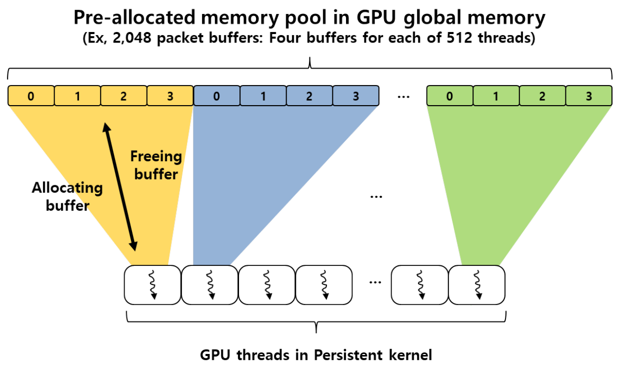

Pre-allocated memory pool in GPU global memory
(Ex, 2,048 packet buffers: Four buffers for each of 512 threads)
0 1 2 3 0 1 2 3 ... 0 1 2 3
Freeing buffer
Allocating buffer
GPU threads in Persistent kernel

Fig. 5. Mini-mempool per GPU thread for lock-free memory pool access.

GPU-Ether a certain level of parallelism while maintaining sequentiality. Based on extensive testing with various batch sizes, we set 128 as the default batching (four consecutive warps) considering trade-offs between thread scheduling and performance. With this optimization, we obtain significant performance gain in packet forwarding (Section V).

D. Mini-mempool

The GPU-Ether prototype has a memory pool that can hold a total of 2,048 packet buffers, and this value can be changed at the initialization phase. The packet I/O operating on the host-side usually manages the ring descriptors and packet buffers sequentially in a loop with a single thread. However, in GPU-Ether where multiple GPU threads operate in parallel, packet buffer allocation requests may co-occur and lock operation is required. Unfortunately, the current CUDA architecture supports only warp-level or thread-block-level locking rather than individual thread-level [24]. Also, lock operation in the CUDA architecture can significantly degrade GPU threads' performance and there is also a risk of live-locks. To resolve the issue, we divide the memory pool into separate *mini-mempools* for each thread and eliminate the need for lock operation during the memory pool access from GPU threads (refer to Fig. 5).

IV. Implementation

To implement GPU-Ether, GPUdirectRDMA API is required for P2P-DMA configuration. Instruction-level compute preemption is also needed to allow GPU-Ether to run persistent kernels with other active GPU applications. There are several candidate GPUs available in the market, and we choose Nvidia Quadro P4000 GPUs for our prototype. For networking, we use Intel X520-DA2 NICs (Dual-port 10GbE).

A. GPU-Ether operations inside of GPU

According to our tests, one GPU thread per packet is enough to achieve the line-rate and more threads allocation causes inefficiencies due to synchronization overhead between them. In our GPU-Ether prototype, both kernels (Rx and Tx) have 512 threads, respectively, and each thread is mapped to a descriptor one-to-one to process a packet. The number is equal to the default number of descriptors for the Intel ixgbe driver. As both kernels are launched in separate CUDA streams, they can operate concurrently.

Determining the owner of packet buffers: In GPU-Ether, each thread has an independent mini-mempool, so packet buffers are not shared between threads within a kernel. However, to hand over packets from one kernel to another, we have to copy the packets or make kernels to share the buffer. Here we choose the latter to ensure low latency via zero-copy data transfer. To determine the current owner of a packet buffer shared by multiple kernels, we add a *status_register* field in packet buffer to specify the owner's ID.[3]

Each kernel has a unique ID and only the kernel that matches the owner's ID written in the *status_register* of a packet buffer can access the buffer. Note that the Rx-kernel does not require an ID because it always allocates new buffers from the memory pool. When a packet buffer is initially allocated from the memory pool, the value of the *status_register* is zero. After the buffer is filled with a newly arrived packet, it is changed into a value indicating a specific kernel, and then packet processing is performed by threads in the corresponding kernel.[4] In the end, the packet buffer is returned to the memory pool, and the *status_register* value is set back to zero.

Here, we describe an example scenario. Let us assume that three persistent kernels with the same number of threads are running: an Rx-kernel, a router kernel (1), and a Tx-kernel (2). The value within parentheses is an ID for each kernel and Rx-kernel has no ID as explained above. When the fourth thread with an empty packet buffer in the Rx-kernel receives a packet, the value of the *status_register* is changed into one, the ID for the router kernel. Then, the fourth thread in the router kernel becomes the owner of the buffer and processes the packet. After packet processing is completed, the thread changes the value of the *status_register* into two, the ID for the Tx-kernel,

[3]The owner of a packet buffer can be a thread within a Tx-kernel or an application kernel and is limited to one thread at a time.

[4]There can be multiple threads for one packet.

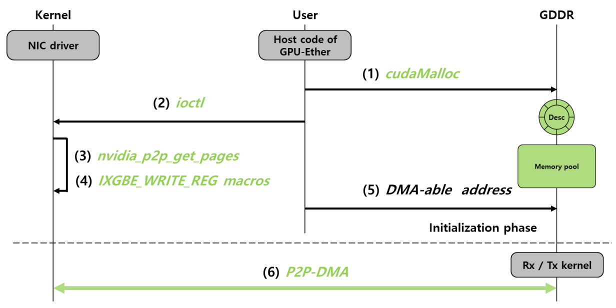


Fig. 6. The NIC initializing process for using the memory pool and descriptors created in the GPU memory.

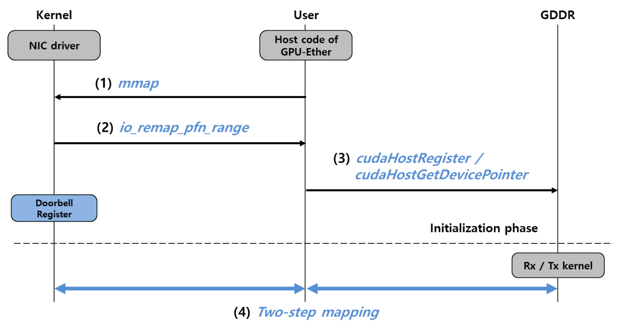


Fig. 7. The process of mapping the doorbell register in the NIC driver to the GPU memory.

and finally, the fourth thread in the Tx-kernel prepares the descriptor for the packet and sends it to the network. We could quickly implement network applications that operate in the form of a persistent kernel, and compatibility with existing GPU applications will be supported through minor API modifications.

Rx-kernel: Each thread in the Rx-kernel polls the Descriptor-Done field in its own Rx descriptor to confirm the completion of a packet transfer into the packet buffer in GPU memory. This field in descriptor is set by NIC hardware after completing DMA for a packet to the corresponding descriptor's buffer. When the Rx-kernel starts, each thread gets a packet buffer from its mini-mempool, attaches it to the descriptor and waits for a packet. At the notification of packet reception via the marked Descriptor-Done field, the corresponding thread makes the buffer available to other GPU applications (e.g., Tx-kernel or NF applications in our case) by changing the *status_register* value of the buffer. After that, a new buffer is assigned to the thread for an another packet. Note that threads also reset descriptors for reusing after passing buffers.

These tasks are executed in parallel by the threads in the current *warp_batch*, and then tasks are completed, the last thread in this *warp_batch* updates information to the NIC by accessing the doorbell register. At the same time, the last thread passes the work permission to threads in the next *warp_batch*. For sequential processing, a branch is created to ensure that only threads in current *warp_batch* perform tasks, and then the *__syncthreads* command is needed for post-branch synchronization. Since frequent thread synchronization seriously degrades persistent kernels' performance, we try to devise a mechanism that minimizes intra-kernel synchronization. In Rx-kernel, synchronization is used in only two places, the starting point of the persistent loop, and right before the last thread in current *warp_batch* accesses the doorbell after updating descriptor information.

Tx-kernel: Tx-process operates similarly as Rx-process, but it is more complicated because the interface between packet I/O and the NIC is asynchronous. In the Rx-process, when a new packet is received, the NIC marks the Descriptor-Done in the Rx descriptor so that the descriptor can be cleaned up immediately. However, in the Tx-process, there is a time gap between requesting the NIC to send a packet and when the NIC actually sends the packet. Hence, it is inevitable to compose the Tx-kernel with two parts: getting available Tx descriptors via Descriptor-Done field set by the NIC and then placing new packets on the descriptors.

When a Tx-kernel starts, threads in the kernel check whether there are descriptors marked with Descriptor-Done fields to obtain available descriptors. They firstly clean up the descriptors by returning buffers to the memory pool, because marked Descriptor-Done fields convince completions of packet transmissions. Then, each thread within the current *warp_batch* checks whether any of the packet buffers in its mini-mempool have *status_register* marked with Tx-kernel's unique ID. If found, it gets the buffer, makes the buffer used by the descriptor and sets relevant information. When all threads in the current *warp_batch* have completed setting descriptors for packets to be sent, the last thread accesses the doorbell register to instruct the NIC to send packets.

As an alternative design, we may consider implementing both Rx/Tx kernels as functions to maintain only one persistent kernel for handling both processes. In this design, all processes are sequentially executed and the thread receiving a packet is also in charge of transmitting the packet. It eliminates concerns about changing the packet buffer ownership but performance degradation occurs because transmission and reception cannot be performed simultaneously.

B. Initializing GPU-Ether

Before executing persistent kernels that perform packet I/O in the GPU, an initializing process for interaction of a GPU and a NIC is required. This is done only once during initialization and the GPU kernel handles all subsequent executions.

Setting memory pool and descriptors via P2P-DMA: Firstly, memory pool and descriptors are created in the GPU memory using *cudaMalloc* and the addresses and sizes are informed to the NIC driver by the *ioctl* system call in the host-side code of GPU-Ether. The NIC driver obtains DMA-able addresses of them using GPUdirectRDMA API, *nvidia_p2p_get_pages*. The DMA-able address of the memory pool is then passed to the Rx/Tx kernel during initialization so that packet buffers can be accessed with appropriate offsets from GPU kernels. It enables GPU kernels of GPU-

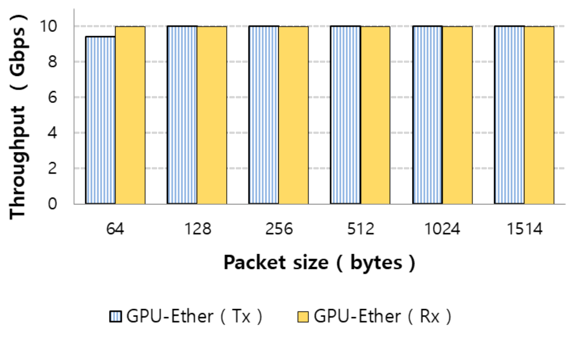


Fig. 8. Rx and Tx throughput of GPU-Ether with various packet sizes.

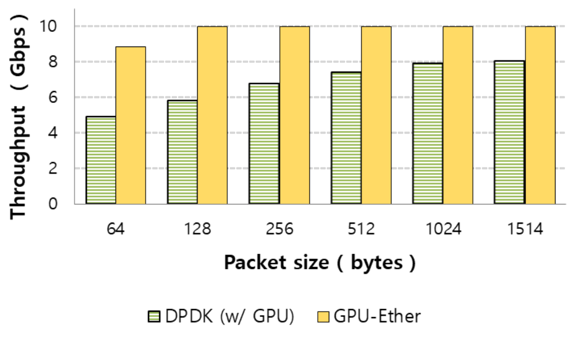


Fig. 9. Forwarding throughput comparison between GPU-Ether and DPDK with various packet sizes.

Ether to update DMA addresses of packet buffers for a NIC with DMA-able GPU memory addresses. For the descriptors, the DMA-able addresses of them are configured using the IXGBE_WRITE_REG macro. After that, the NIC can be controlled through descriptors created in the GPU memory (Fig. 6). We exploit only one TX/RX descriptor ring for simplicity in our implementation. It is sufficient to achieve 10GbE network performance. Note that load balancing through hardware-level hashing called RSS (Receive-Side Scaling) is usually done automatically on 10GbE NIC or higher, so we force only one ring pair to be active with *ethtool* command.

Setting doorbell registers through two-step mapping: The doorbell registers reside on the NIC hardware and the NIC driver uses them after mapping them to the virtual address space with *ioremap* system call. Since they cannot be mapped directly from the NIC to the GPU memory, a two-step mapping is required to map the virtual address extracted by the NIC driver to the user level and then remaps them to the GPU memory. To map the doorbell registers to the user level, one must implement *mmap* system call in the NIC driver. When *mmap* is called, the NIC driver maps the virtual addresses of the doorbell registers to the user level address space by calling *io_remap_pfn_range* system call. Then this address is mapped again to the GPU memory via *cudaHostRegister* and *cudaHostGetDevicePointer*, making them accessible to the GPU kernels (Fig. 7).

V. Evaluation

A. Experiment Setup

We run our experiments on two nodes directly connected by a 10 Gbps Ethernet link (Intel X520-DA2 10GbE NIC). Both nodes have the same hardware and software configuration. They are equipped with Intel i7-6800K CPU 3.4 GHz, which has six cores and supports hyper-threading. The installed memory capacity is 16GB and operates at 2,133 MHz. An Nvidia Quadro P4000 Pascal GPU is deployed. It has 14 streaming multiprocessors, a total of 1,792 cores, and 8 GB of memory. The operating system is 64-bit Ubuntu 18.04.2 LTS with Linux kernel 4.18.15. CUDA 10.1 and Nvidia-driver 418.67 are used for GPU driver. DPDK-pktgen is used to generate network traffic at line-rate [25].

In Fig. 10, we describe the experimental setup with two different packet I/Os (DPDK and GPU-Ether) to access the GPU. In Fig. 10(a), each packet received by DPDK is stored within rte_mbuf structure. To ensure that packets are allocated in contiguous space in the memory pool, we rearrange the received packets in contiguous space before copying them to the GPU memory. Here, we use the master-worker thread model similar to [1], [5] in which a worker thread is responsible for packet I/O through DPDK and a master thread is responsible for data transfer with the GPU.

When a worker thread receives packets, it copies them to the master thread's queue. The master thread copies them again to the GPU memory and the transmission goes vice versa. Besides, we include 64-byte of headroom and tailroom when transferring each packet to the GPU as they are required for storing additional headers and authentication data generated when processing one of our test application, IPsec gateway. The GPU's packet buffer is set to four times the master queue size to avoid processing gaps on the GPU due to data copy delays.

In GPU-Ether (Fig. 10(b)), packets are directly transferred

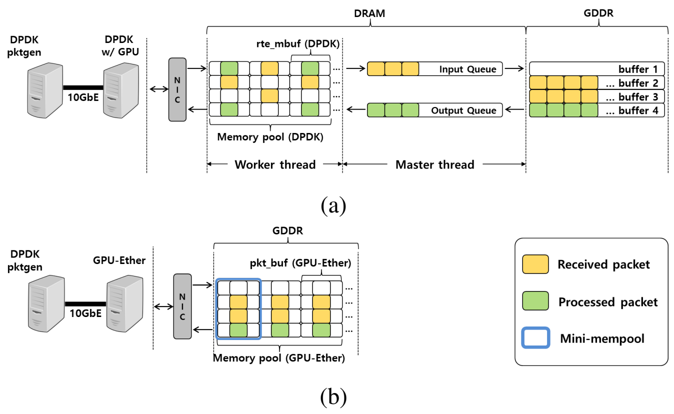


Fig. 10. Packet delivery process in two different packet I/Os used for evaluation. (a) DPDK with GPU. (b) GPU-Ether.

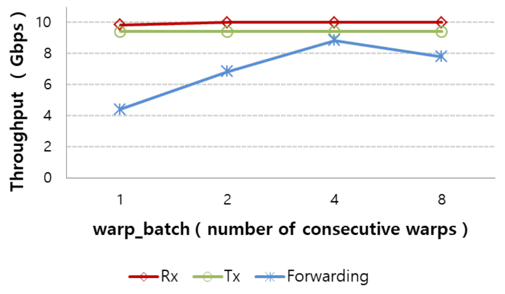


Fig. 11. Effect of changing the number of warps performing packet I/O at a time. All packets are 64 bytes.

into the packet buffers in the GPU memory without any copy operation, and each thread processes packets in its mini-mempool. GPU-Ether also includes headroom space in the packet buffer structure and metadata and the rest are aligned with 32 bytes and 128 bytes, respectively, taking into account the cache-line size of CUDA architecture [26]. In Rx and Tx kernels, each GPU thread counts the numbers of receiving and sending packets by calling *atomicAdd* function with variables shared among threads. This accumulated packet count is copied into the monitoring loop running on the host via *cudaMemcpy* every second, and throughput is calculated with it. The configurations explained above are applied to all experiments described below.

B. Performance of GPU-Ether Packet I/O

In Fig. 8, we present preliminary experimental results to see Rx and Tx throughput of GPU-Ether with various packet sizes. Each throughput is independently measured. It is observed that GPU-Ether can achieve line-rate for all packet sizes except for transmitting 64-byte packets (94%). According to studies in [27], [28], the effective bandwidth of a PCIe link is drastically reduced when the link is full of small packets. In GPU-Ether, small control packets for accessing doorbells and descriptors flow in the same direction with transmitted packets (from GPU to NIC). With *warp_batch* of 128, 108K doorbell register and 13.9M descriptor accesses occur every second, which could be the source of throughput loss. Also, unlike host-side packet I/O where PCIe is used only between the NIC and DRAM, communication among NIC, DRAM, and GPU via PCIe is required in GPU-Ether. It complicates the delivery process of PCIe traffic. However, it is noted that 64 byte packets are unusually small, and GPU-Ether can achieve the line-rate with the reasonable size of packets.

C. Packet forwarding

In Fig. 9, we compare the forwarding throughput of DPDK and GPU-Ether. In this experiment, the received packets are transferred to the GPU via DPDK or GPU-Ether. MAC and IP addresses are modified for forwarding by CPU for DPDK and by GPU for GPU-Ether, and packets are forwarded back to the sending host. When we use DPDK to access the GPU, multiple copies are inevitable and the performance of DPDK is lower than that of GPU-Ether. In DPDK, as the packet size increases from 64 bytes (49.12%) to 1,514 bytes (80.58%), the number of incoming packets per second (PPS) decreases and the number of memory copies also decreases. On the other hand, GPU-Ether, which generates only two P2P-DMAs without memory copy, shows 100% forwarding performance at all packet sizes except 64 bytes (88.5%). The size of I/O burst (number of packets received or sent in one function call) in DPDK is 32 and the batch size used for data transfer between host and GPU is 512.

TABLE I. Execution time comparison with co-located host application.

Configuration	Elapsed time (ms)	Slowdown
Idle	8023.4	100%
w/ DPDK	23969.2	298.74%
w/ GPU-Ether	8078.4	100.69%

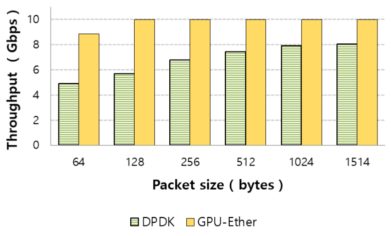
D. Varying parallel execution degree

In Fig. 11, we present the performance of GPU-Ether with various sizes of *warp_batch*. The size of *warp_batch* indicates the degree of parallelism, as explained in Section III. In this experiment, the packet size is fixed as 64 bytes to observe the worst performance. Unlike Rx and Tx, it is observed that the forwarding throughput is maximized when *warp_batch* is 128 and decreases after that. In the forwarding scenario, a Tx-kernel can proceed on packets after an Rx-kernel processed, and the Rx-kernel performance is limited by the buffer-freeing rate of the Tx-kernel due to buffer sharing. In addition to this mutual bound, a packet forwarding process consists of buffer allocation/free, buffer owner verification and source and destination MAC/IP header modifications. Low parallelism (a small *warp_batch*) under this condition leads to significant performance degradation. On the other hand, the work-permission can move to the next *warp_batch* only after all threads in the current *warp_batch* completed their tasks. Since Rx/Tx kernels reside in different SMs, thread scheduling for the same processing range of threads for independent kernels becomes inefficient as the *warp_batch* size increases. If *warp_batch* size exceeds 128, it is confirmed that forwarding performance is rather reduced.

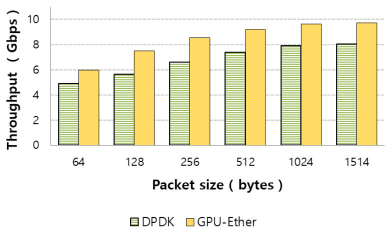
Rx and Tx-processes show a slight performance improvement as *warp_batch* size increases, but there is no significant difference since they have no overhead for buffer allocation/free.

E. Interference with co-located host applications

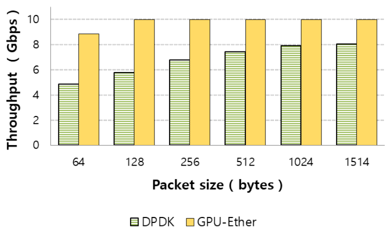
While DPDK uses multiple CPU cores to handle packet I/O for GPU, GPU-Ether does not use any CPU core since every task is done on GPU after initialization. Checking the CPU usage with *top* command shows that all CPU cores are idle during GPU-Ether is running. To examine the actual effect of GPU-Ether on host applications, we conduct a noisy-neighbor experiment as in [13]. We model a host application that multiplies two integer matrices of size 1,140 x 1,140 to



(a) IPv4 forwarding.



(b) IPsec Gateway.



(c) Network intrusion detection system.

Fig. 12. Performance of various network applications.

fully occupy our machine's Last Level Cache, 15MB. Then, the host application's elapsed time is compared when DPDK or GPU-Ether is executed concurrently with this application. TABLE I shows the execution time. DPDK goes through host memory for every packet I/O performed and makes the CPU cache dirty, which causes three times higher execution time of the host application. On the other hand, GPU-Ether does not show much difference from the idle state. A slight difference is due to the access to the mapped doorbell registers.

F. Applications

To demonstrate the performance and practicality of GPU-Ether, we implement three network applications: IPv4 forwarding, IPsec gateway, and NIDS. They are also implemented as persistent kernels and follow the zero-copy principle in passing packet buffers between kernels for the low latency. Since they are launched once and keep running until the end of the program, kernel launching overhead is excluded in the evaluation. For IPv4 forwarding, allocating one thread per packet is sufficient, but the rest of the applications need multiple threads per packet to alleviate each thread's burden. The specific configuration for each application is described later in this section.

To determine the impact of the packet I/O on the application, DPDK and GPU-Ether are connected to the same application to measure performance.

IPv4 forwarding: We use DIR-24-8-BASIC [29] as the table lookup algorithm as in previous studies [1], [5]. For realistic measurements, we create the routing table from a snapshot of BGP tables collected on August 1, 2018, by Route-Views [30]. The number of unique prefixes in the snapshot is 474,319, and only 2% of the prefixes are longer than 24 bits. Note that all packets used in the test contain a randomized destination IP address.

IPsec gateway: We use AES-128 (CTR mode) and HMAC-SHA1 algorithms to implement the application that operates in the IPsec Gateway ESP tunnel mode. For AES, we maximize parallelism by allowing each thread to process in units of AES blocks (16 bytes). However, in SHA1 processing, since authentication data is generated by accumulating processed results for each SHA1 block (64 bytes), we could only parallelize SHA1 at the packet level. Meanwhile, since the persistent kernel can contain 1,024 threads and occupies one SM per thread block persistently, it runs out of threads when processing AES for packets larger than 512 bytes on a Quadro P4000 GPU. For larger than 512 byte packets, each thread handles multiple AES blocks. In this mode, the IPsec gateway increases packet size by adding extra headers, padding, and authentication data, so we use 1,460 bytes packet as the largest one to prevent fragmentation due to exceeding the MTU size. All packets used in this test contain randomized source and destination IP addresses and payload.

Network intrusion detection system: Our NIDS application is implemented based on the codes from Snort [31], one of the most popular open-source projects in this category. Snort reads the ruleset file to store it in a TRIE format and then performs pattern matching with the Aho-Corasick (AC) algorithm. Snort project is not only massive in code but also has too many linked-list to apply on GPU; we implement a simplified version of it. We change the linked-list style TRIE generated by Snort into a two-dimensional array and put it into GPU memory. After each thread in application checks the destination port of a packet, if there exists a TRIE for the corresponding port, it executes pattern matching for the payload of a packet. To maximize parallelism, we chop payload into multiple pieces and map each piece to a thread, except 64-byte packet processing. For 64 byte packets, we allocate one thread per packet because the payload is 18 bytes, excluding headers and Ethernet CRC (46 bytes). In testing, we use innocent synthetic traffic with a randomized payload and destination port.

G. Application throughput

Fig. 12 shows the performance of three applications. The IPv4 forwarding and the NIDS show the same throughput as in forwarding scenario in both DPDK and GPU-Ether cases (Fig. 9), which implies packet I/O is the main bottleneck of these applications (Fig. 12(a) and Fig. 12(c)). The IPv4 forwarding performs table lookup for destination IP, and the NIDS performs pattern matching for payload that is properly partitioned into multiple threads. The simplicity of both operations is the reason for the intact throughput. The IPsec gateway is the only application that includes packet modification among the three test cases. Despite allocating multiple threads for each packet, performance degradation is inevitable because many read/write and memory copy operations are included (Fig. 12(b)). Considering that the IPsec gateway is a compute/memory-intensive application to handle traffic at line-rate, GPU-Ether is considered to be very useful for typical GPU applications where the majority of communication is composed of MTU sized packets (1,514 bytes).

VI. Related works

Packet processing acceleration on GPUs: Studies such as PacketShader [1], SSLShader [3], Kargus [2], and GASPP [4] leveraged the high processing power of a GPU to maximize network processing. GASPP improves the performance of applications by relocating workload so that similar-sized packets are processed in the same warp, and we expect that GPU-Ether will show performance improvement by implementing this technique in it. In these studies, a CPU is responsible for network I/O, so data transfer between a CPU and a GPU becomes a performance bottleneck. To address this, APUnet [5] eliminated data transfers by using Accelerated Processing Units (APUs), where a CPU and a GPU use single unified memory space. However, since an integrated GPU in an APU has a lower memory bandwidth than a dedicated GPU and shares the memory space with a CPU, heavyweight synchronization operations are required.

GPU-native network I/O frameworks: There are several approaches to provide native network I/O for a GPU to avoid data transfer from a CPU. GPUnet provides GPU native socket abstraction that runs over a reliable RDMA connection in [16]. GPUrdma [17] eliminates information exchange between a CPU and a GPU by addressing the issue of direct doorbell register access within the GPU. NVSHMEM [32] provides abstraction for the partitioned global address space on top of RDMA, and dCUDA [15] partially implements Message Passing Interface (MPI) between GPUs. All of these works allow GPUs to perform networking directly, resulting in low latency and improved performance compared to traditional CPU-based solutions but they need specialized hardware, Infiniband RDMA HCA. GPU-Ether is the first native packet I/O for GPUs running on commodity Ethernet.

Peer-to-peer DMA: In Morpheus [33], object deserialization is offloaded to a storage device and P2P-DMA is utilized for communication between NVMe SSD and GPU. SPIN [28] also improves communication performance between an SSD and a GPU by integrating P2P-DMA into the standard OS file I/O stack. However, SPIN shows that P2P-DMA may slower than CPU-mediated I/O for very short reads as P2P-DMA does not support read-ahead mechanism. Neugebauer et al. [27] build a model based on the PCIe specification to show the effective bandwidth achievable on a PCIe link depending on a TLP size. According to their model, when the transfer size is small (small TLPs), the effective bandwidth of the PCIe link drops sharply. We believe this is because PCIe links have been used primarily to move large data to connected devices. Many recent studies including GPU-Ether encounter a performance issue with P2P-DMA and it is expected to be solved in the new generation of PCIe standard.

Offloading network I/O to other hardware: FlexNIC [14] proposes a new architecture for offloading network I/O to programmable NICs. It provides flexibility using Reconfigurable Match-action Tables (RMT), which enables packet routing to the appropriate target core or device. Lynx [13] facilitates SmartNICs to make accelerated network services run on various accelerators upon RDMA. These studies have in common that they delegate network I/O processing to separate hardware instead of using GPU's resource. Their biggest advantage is that extra hardware can handle network I/O for multiple GPUs, and traffic from multiple GPUs is aggregated into one NIC. It is not suitable for GPU applications such as distributed machine learning that require large bandwidth [6]. The native packet I/O such as GPU-Ether, which utilizes the minimal level of GPU's resource, maybe a more realistic alternative.

VII. Conclusion

In this paper, we presented GPU-Ether, a novel GPU native packet I/O framework that works on commodity Ethernet devices. We confirmed that GPU direct networking could provide the minimum level of latency of the Ethernet link via latency analysis of GPU traffic delivery. In addition, as a GPU performs packet I/O directly without CPU intervention, implementation design is simplified and interference with other CPU applications is minimized. GPU-Ether effectively managed the inconsistency between GPU parallel processing hardware architecture and sequential packet I/O processing with careful design choices. We demonstrated the effectiveness of our approach with IPv4 forwarding, IPSec gateway and a network intrusion detection system. In the recent trend of near data processing and disaggregation, we believe GPU-Ether provides native networking without the need for expensive hardware or CPU, which can be a useful platform.

Acknowledgment

This work was supported by the National Research Foundation of Korea(NRF) grant funded by the Korea government(MSIT). (NRF-2018R1D1A1B07049834, NRF-2020R1A2C2009809 and NRF-2016M3C4A7952587.) We would like to thank the anonymous reviewers for their insightful comments and suggestions.

References

[1] S. Han, K. Jang, K. Park, S. Moon, Packetshader: a gpu-accelerated software router, ACM SIGCOMM Computer Communication Review 40 (4) (2010) 195–206.

[2] M. A. Jamshed, J. Lee, S. Moon, I. Yun, D. Kim, S. Lee, Y. Yi, K. Park, Kargus: a highly-scalable software-based intrusion detection system, in: Proceedings of the 2012 ACM conference on Computer and communications security, 2012, pp. 317–328.

[3] K. Jang, S. Han, S. Han, S. B. Moon, K. Park, Sslshader: Cheap ssl acceleration with commodity processors., in: NSDI, 2011, pp. 1–14.

[4] G. Vasiliadis, L. Koromilas, M. Polychronakis, S. Ioannidis, Gaspp: A gpu-accelerated stateful packet processing framework, in: 2014 USENIX Annual Technical Conference (USENIX ATC 14), 2014, pp. 321–332.

[5] Y. Go, M. A. Jamshed, Y. Moon, C. Hwang, K. Park, Apunet: Revitalizing gpu as packet processing accelerator, in: 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17), 2017, pp. 83–96.

[6] H. Zhang, Z. Zheng, S. Xu, W. Dai, Q. Ho, X. Liang, Z. Hu, J. Wei, P. Xie, E. P. Xing, Poseidon: An efficient communication architecture for distributed deep learning on gpu clusters, in: 2017 USENIX Annual Technical Conference USENIX ATC 17), 2017, pp. 181–193.

[7] J. Gu, M. Chowdhury, K. G. Shin, Y. Zhu, M. Jeon, J. Qian, H. Liu, C. Guo, Tiresias: A gpu cluster manager for distributed deep learning, in: 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19), 2019, pp. 485–500.

[8] Fighting COVID-19. [n.d.]. Simulation, AI, data analytics and visualization come together on NVIDIA's scientific computing platform to fight the global pandemic., https://blogs.nvidia.com/blog/2020/06/22/fighting-covid-19-scientific-computing/.

[9] Amazon Elastic Inference. [n.d.]. Amazon Elastic Inference: Add GPU acceleration to any Amazon EC2 instance for faster inference at much lower cost., https://aws.amazon.com/machine-learning/elastic-inference/.

[10] Google Cloud GPUs. [n.d.]. High-performance GPUs on Google Cloud for machine learning, scientific computing, and 3D visualization., https://cloud.google.com/gpu.

[11] Top 500 Supercomputer Sites. [n.d.]. The TOP500 project ranks and details the 500 most powerful non-distributed computer systems in the world., https://www.top500.org.

[12] R. Ammendola, A. Biagioni, O. Frezza, G. Lamanna, A. Lonardo, F. L. Cicero, P. S. Paolucci, F. Pantaleo, D. Rossetti, F. Simula, et al., Nanet: a flexible and configurable low-latency nic for real-time trigger systems based on gpus, Journal of Instrumentation 9 (02) (2014) C02023.

[13] M. Tork, L. Maudlej, M. Silberstein, Lynx: A smartnic-driven accelerator-centric architecture for network servers, in: Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems, 2020, pp. 117–131.

[14] A. Kaufmann, S. Peter, T. Anderson, A. Krishnamurthy, Flexnic: Rethinking network dma, in: 15th Workshop on Hot Topics in Operating Systems (HotOS XV), 2015.

[15] T. Gysi, J. Bar, T. Hoefler, dcuda: hardware supported overlap of computation and communication, in: SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, IEEE, 2016, pp. 609–620.

[16] M. Silberstein, S. Kim, S. Huh, X. Zhang, Y. Hu, A. Wated, E. Witchel, Gpunet: Networking abstractions for gpu programs, ACM Transactions on Computer Systems (TOCS) 34 (3) (2016) 1–31.

[17] F. Daoud, A. Watad, M. Silberstein, Gpurdma: Gpu-side library for high performance networking from gpu kernels, in: Proceedings of the 6th international Workshop on Runtime and Operating Systems for Supercomputers, 2016, pp. 1–8.

[18] Intel DPDK: Data Plane Development Kit., https://www.dpdk.org/.

[19] PSIO: Packet I/O Engine., https://shader.kaist.edu/packetshader/io_engine/index.html.

[20] L. Rizzo, Netmap: a novel framework for fast packet i/o, in: 21st USENIX Security Symposium (USENIX Security 12), 2012, pp. 101–112.

[21] L. Gao, Y. Xu, R. Wang, Z. Luan, Z. Yu, D. Qian, Thread-level locking for simt architectures, IEEE Transactions on Parallel and Distributed Systems 31 (5) (2019) 1121–1136.

[22] A. Nguyen, Y. Fujii, Y. Iida, T. Azumi, N. Nishio, S. Kato, Reducing data copies between gpus and nics, in: 2014 IEEE International Conference on Cyber-Physical Systems, Networks, and Applications, IEEE, 2014, pp. 37–42.

[23] S. Larsen, P. Sarangam, R. Huggahalli, S. Kulkarni, Architectural breakdown of end-to-end latency in a tcp/ip network, International journal of parallel programming 37 (6) (2009) 556–571.

[24] Nvidia Developers, Try to use lock and unlock in CUDA., https://forums.developer.nvidia.com/t/try-to-use-lock-and-unlock-in-cuda/50761.

[25] Pktgen: Traffic Generator powered by DPDK., https://github.com/pktgen/Pktgen-DPDK.

[26] H. Kim, S. Hong, H. Lee, E. Seo, H. Han, Compiler-assisted gpu thread throttling for reduced cache contention, in: Proceedings of the 48th International Conference on Parallel Processing, 2019, pp. 1–10.

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

[28] S. Bergman, T. Brokhman, T. Cohen, M. Silberstein, Spin: Seamless operating system integration of peer-to-peer dma between ssds and gpus, ACM Transactions on Computer Systems (TOCS) 36 (2) (2019) 1–26.

[29] P. Gupta, S. Lin, N. McKeown, Routing lookups in hardware at memory access speeds, in: Proceedings. IEEE INFOCOM'98, the Conference on Computer Communications. Seventeenth Annual Joint Conference of the IEEE Computer and Communications Societies. Gateway to the 21st Century (Cat. No. 98, Vol. 3, IEEE, 1998, pp. 1240–1247.

[30] RouteViews. [n.d.]. University of Oregon Route Views Project., http://www.routeviews.org/routeviews/.

[31] Snort. [n.d.]. Network Intrusion Detection and Prevention System., https://www.snort.org/.

[32] NVSHMEM. [n.d.]. GPU-side API for remote data access, collectives and synchronization., http://www.openshmem.org/site/sites/default/site_files/SC2017-BOF-NVIDIA.pdf.

[33] H.-W. Tseng, Q. Zhao, Y. Zhou, M. Gahagan, S. Swanson, Morpheus: creating application objects efficiently for heterogeneous computing, ACM SIGARCH Computer Architecture News 44 (3) (2016) 53–65.