Gaviss : Boosting the Performance of GPU-Accelerated NFV Systems via Data Sharing

Liangchen Guo, Kai Zhang, and X. Sean Wang, *Senior Member, IEEE*

Abstract—GPUs have demonstrated the capability of significantly improving the performance of network functions (NF). In an Network Function Virtualization (NFV) system, multiple NFs form a service chain to provide services. However, NFs in state-of-the-art GPU-accelerated NFV systems still utilize a GPU independently where each NF needs to transfer data to the GPU memory for acceleration. As a result, a packet might be transferred into the GPU memory by each NF when it passes through the service chain. We find these expensive and repetitive transfers are the main factor that limits the overall performance of an NFV system. We propose Gaviss, a GPU-accelerated NFV system with effective data sharing. By sharing packets in the GPU memory among network functions, a packet needs to be transferred to the GPU only once, eliminating the performance overhead caused by repetitive transfers. Extensive experimental results show that Gaviss can improve the overall throughput by 2.6-13.2× and reduce the latency by up to 37.9%, when compared with state-of-the-art approaches. Moreover, Gaviss also demonstrates up to 2.5× higher price-performance ratio than CPU-based implementations, making GPUs competitive for building NFV systems.

Index Terms—Communications technology, computer networks, network function virtualization

1 Introduction

Network Function Virtualization (NFV) is a network architecture concept that virtualizes network functions (NFs) to provide agile implementation, deployment and management. With the fast growing of network bandwidth, the performance of NFV systems is a critical metric for their supporting of Internet services. Thus, how to enhance the performance of NFV systems is a hot spot in the field of networked system implementations.

Current approaches for building high-performance NFV systems can be categorized into two classes. Some studies [1], [2] discard virtualization since the data movement between virtual machines incurs non-trivial overheads. While the approaches can improve system performance, the flexibility offered by virtualization is compromised. Another class of research advocates the use of heterogeneous general-purpose hardware to improve the performance of NFV systems, such as programmable NICs [3] and GPUs [4], [5].

GPUs are promising accelerators for building NFV systems for their massive number of cores and high memory bandwidth. It has been demonstrated that compute-intensive algorithms widely employed in network applications can benefit from the parallel computation capacity of GPUs [6], [7], [8], [9]. To utilize a GPU for acceleration, packets are batched in the host memory and transferred into the GPU memory for processing. In an NFV system, several NFs generally form a service chain to provide customized services, where the output of an NF can be the input of the following one. Therefore, different with a single GPU-accelerated network function, the GPU is accessed by multiple NFs concurrently in an NFV system.

G-NET [4] is one of the state-of-the-art GPU-based NFV systems, which improves its efficiency by enabling multiple NFs to spatially share the GPU computing resources. In G-NET, each NF in a service chain works independently and transfers packets from the host memory to the GPU memory to offload tasks. This way of utilizing GPUs in an NFV system, however, has inherent drawbacks. With a service chain consisting of n NFs running on one node, a packet may be transferred into the GPU memory for n times by each NF in the chain. PCIe data transfer is known to incur high overheads [10], [11], [12] and may result in lower system throughput than CPU-based NFs [13], [14]. Even worse, we find that the repetitive transfers in an NFV system would further exert a huge impact on the performance from two aspects. 1) *Data transfer conflicts:* Since there is only one DMA engine in each direction in a commodity GPU, there would be conflicts when multiple NFs are transferring data between the host memory and the GPU memory. Consequently, other data transfers will be postponed when the PCIe bus is occupied, degrading the overall throughput. Performance would be further degraded when NFs need to transfer packets back to the host memory, such as an IPsec gateway that performs encryption in the GPU. 2) *Bandwidth bottleneck:* The repetitive data transfers may limit the maximum system performance to only $1/n$ of the PCIe bandwidth. We have measured the maximum PCIe transfer speed to the GPU on our server as 101.3Gbps. As a result, the theoretical maximum overall throughput is limited to 25.3Gbps when four NFs in a service chain need to transfer entire packets to the GPU. Besides the overhead of PCIe data transfers, our experimental evaluations also show that current systems cannot fully exploit the computation

• *The authors are with the School of Computer Science and Technology, Fudan University, Shanghai 200437, China. E-mail: {lcguo17, zhangk, xywangCS}@fudan.edu.cn.*

Manuscript received 18 February 2022; revised 30 May 2022; accepted 20 July 2022. Date of publication 25 July 2022; date of current version 23 August 2022.
This work was supported by VMware under Grant 10.13039/100016682.
(Corresponding author: Kai Zhang.)
Recommended for acceptance by M. Si.
Digital Object Identifier no. 10.1109/TPDS.2022.3193368

1045-9219 © 2022 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See https://www.ieee.org/publications/rights/index.html for more information.
Authorized licensed use limited to: FUDAN UNIVERSITY. Downloaded on June 10,2024 at 11:12:27 UTC from IEEE Xplore. Restrictions apply.

power of GPU due to the imbalanced CPU-GPU pipeline. Overall, the design of current NFV systems constrains the power of GPUs and leads to sub-optimal performance with low resource utilization.

In this paper, we propose Gaviss, an NFV system that adopts data sharing to address the issue of the repetitive data transfers on the CPU-GPU heterogeneous architecture. In Gaviss, packets in the GPU memory are shared among NFs to avoid the repetitive PCIe data transfers and mitigate the potential conflicts. Different with other systems, an NFV system should be able to build various service chains and incorporate NFs with different implementations. The main challenges to reduce data transfers in Gaviss are 1) dynamically generating the synchronization plan for each service chain; 2) minimizing the amount of synchronized data according to the NF implementations and the service chain. By proposing an automatic data synchronization scheme between the host memory and the GPU memory, Gaviss exploits domain knowledge and hints from NFs to enable fine-grained data synchronization. Gaviss derives the optimal synchronization plan which dramatically reduces the amount of transferred data for a service chain. With the proposed set of techniques, Gaviss unleashes the power of GPUs to significantly improve the overall performance by data sharing. As far as we know, Gaviss is the first NFV system that effectively shares GPU memory among NFs. The main contributions of this paper are as follows.

- We proposed an adaptive GPU data sharing mechanism to reduce repetitive PCIe data transfers and potential conflicts for GPU-accelerated NFV service chains.
- We proposed a fine-grained data synchronization plan generation scheme which utilizes domain knowledge and read/write hints from NFs to minimize synchronization costs.
- We designed and implemented Gaviss, a high-performance GPU-accelerated NFV system prototype with effective and efficient data sharing and synchronization.

Through extensive evaluations, we show that Gaviss is capable of enhancing the overall throughput by 2.6–13.2× and reducing the latency by up to 37.9%, for service chains with complex functionalities. Moreover, Gaviss shows up to 2.5× higher price-performance ratio than CPU-based NFV implementations.

2 Background and Motivation

2.1 Utilizing GPUs in NFV Systems

Network functions are the essential building blocks in the network infrastructure. Typical NFs include firewall, network intrusion detection system (NIDS), IPsec gateway and router. In a production system, multiple NFs are typically formed into a service chain to deliver customized network functionalities. Recently, several NFV systems [1], [2], [15], [16], [17] have been proposed to build NF service chains on commodity CPUs. The systems have different designs in terms of virtualization and resource sharing. To improve performance on CPUs, some works [1], [2] reduce data movement overheads by multiplexing the packet processing pipeline of a service chain, i.e. executing multiple network functions on the same core to exploit the locality of packet data. These approaches discard the benefits of virtualization, including independent NF development, flexible deployment and migration.

Another class of research focuses on utilizing general-purpose commodity accelerators like GPUs to improve the performance of NFV systems. With high-bandwidth device memory and massive execution cores, GPUs show great potential in accelerating data-intensive and compute-intensive tasks in NFs, such as pattern matching in NIDS [6], [7], encryption/decryption [8] and packet switching [9]. A CPU and a GPU generally form a pipeline for cooperative packet processing, where operations like packet switching and network stack processing are performed by CPU, while compute-intensive operations like encryption and pattern matching are offloaded to GPU. For GPU processing, packet data should be transferred from host memory to GPU device memory via PCIe. Then the processed results are transferred back to the host memory for further actions.

G-NET [4] is the state-of-the-art GPU-accelerated NFV system that addresses the GPU underutilization and performance issues caused by temporal GPU sharing. By exploiting concurrent kernel execution (CKE) in modern GPUs, G-NET enables multiple NFs to utilize a GPU with spatial sharing, i.e., kernels from different NFs can run on a GPU simultaneously. There are three major components in the hypervisor of G-NET. Manager acts as the proxy for GPU virtualization, which creates a common GPU execution context for all NFs to utilize the CKE feature. Manager receives kernel execution and data transfer requests from NFs and performs the corresponding operations within the context. Scheduler monitors and optimizes the overall performance of a service chain by dynamically allocating demanded GPU resources for each NF. Then Scheduler sends messages to Manager to specify resource allocation parameters such as the thread blocks in kernel execution. Switch performs packet I/O with NIC and forwards packets among NFs. G-NET significantly enhances system performance by sharing GPU compute resources.

2.2 Inefficiencies in State-of-the-Art GPU-Accelerated NFV Systems

After quantitative analysis of state-of-the-art GPU-accelerated NFV system, we find that the current way of using GPUs has following inherent drawbacks, which severely constrains the GPU power.

Repetitive PCIe data transfers cause conflicts and limit overall performance. GPU-accelerated NFs transfer packets to the GPU device memory for kernel processing, and packets should be transferred back if they are modified by the kernel. For instance, an IPsec gateway not only needs to transfer data into the GPU memory for encryption/decryption, but also requires the encrypted/decrypted packets being transferred back into the host memory. As a result, when several NFs co-run on one physical node, the number of PCIe data transfer operations grows linearly with the number of NFs.

PCIe data transfer is the main factor that affects the performance of GPU-accelerated NFs, which will compromise the advantage of GPU over CPU [13]. Specifically, a well-

Authorized licensed use limited to: FUDAN UNIVERSITY. Downloaded on June 10,2024 at 11:12:27 UTC from IEEE Xplore. Restrictions apply.

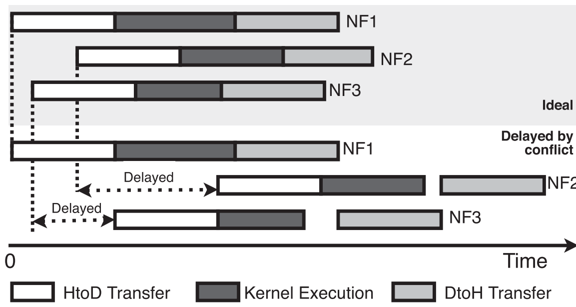


Fig. 1. Data transfer conflicts postpone kernel executions.

optimized CPU implementation with G-Opt [14] may easily outperform a GPU-accelerated NF with PCIe data transfers. G-Opt is an optimization framework for automatically hiding DRAM latency for data structure lookup algorithms with prefetching. Our experiments also prove it, where the data transfer time of four implemented NFs, i.e., router, firewall, IPsec gateway and NIDS, takes 89.0%, 27.3%, 46.7%, 28.7% of the overall GPU execution time, respectively. In an NFV system with multiple NFs, a packet would be transferred to the GPU memory by each NF. We find that the repetitive data transfers are unnecessary and would severely impact the overall performance of an NFV system for two main aspects.

1) PCIe transfer conflict postpones data transfer and kernel execution of other NFs. As demonstrated in Fig. 1, since there are only one Host-to-Device (HtoD) and one Device-to-Host (DtoH) DMA engine in a commodity GPU, a data transfer request would be queued and executed sequentially if the demanded DMA engine is occupied by another NF [4]. Furthermore, the execution of subsequent kernels would also be postponed. The impact to the overall performance increases with more NFs. Fig. 2 shows the adverse performance impact of data transfer conflict on the HtoD data transfer performance. We evaluate three service chains with different lengths, which are composed of following four NFs: firewall (F), NIDS (N), router (R) and IPsec gateway (I). In the figure, *R-I*, for instance, denotes a service chain with a router and an IPsec gateway. We measure the HtoD data transfer time of the router in the three service chains. As shown in the figure, the HtoD time of the router increases with longer service chains, which implies more intense PCIe conflicts. For the service chain with four NFs, the HtoD transfer time is 3.2$\times$ longer than the case when the GPU is exclusively accessed by the router. This demonstrates that PCIe conflicts can cause severe performance degradations to an NFV system, especially with a long service chain.

2) PCIe data transfer bandwidth limits the performance of an NFV system. Previous studies only focus on accelerating NFs with exclusive accessing to a GPU. For an NFV system running at T Gbps with n GPU-accelerated NFs, the PCIe is required to deliver $T \cdot n$ Gbps HtoD transfer performance. Moreover, another $T \cdot n$ Gbps DtoH transfer performance is demanded if the processed packets need to be transferred back. We have measured the PCIe data transfer performance of an NVIDIA Titan X (Pascal) GPU on PCI Express 3.0 $\times$ 16, which achieves 101.3 Gbps HtoD and 95.7 Gbps DtoH data transfer throughputs. It limits the upper bound of the system theoretical throughput to only $101.3/n$ Gbps if n NFs require transferring entire packets to the GPU. For instance, with four NFs, the maximum throughput of the system is limited to be only 25.3 Gbps. The limited data transfer bandwidth would also result in low GPU utilization.

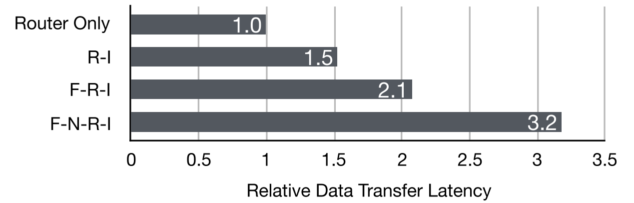


Fig. 2. Latency penalty to HtoD data transfers of a router by PCIe conflicts.

Overall, the repetitive PCIe data transfers in existing GPU-accelerated NFV systems severely degrade the performance and block its adoption in production systems.

3 Gaviss Overview

Our goal is to fully exploit the GPU power to enhance the performance of GPU-accelerated NFV systems. In this section, we introduce our main approaches and make an overview of our system architecture.

3.1 Main Approach

We make a key observation that the main overhead that hinders the overall NFV system performance comes from the repetitive PCIe data transfers. Based on the observation, we propose Gaviss, a GPU-accelerated NFV system that enhances the overall performance by effective data sharing among NFs in a service chain. NetVM [17] and NetBricks [2] have demonstrated the benefits of zero-copy approaches. However, they do not need to consider any sycnrhonization costs and adaptivity for different service chains, because all NFs access shared data in the host memory. Instead, the data sharing and synchronization in Gaviss is adaptive and flexible for minimizing the data transfer overhead for various service chain compositions and NF implementations. In Gaviss, we employ the zero-copy principle to manage the life cycle of a packet, where each packet can be transferred into the GPU for only once. GPU kernels of subsequent NFs directly access packets in the GPU memory, avoiding transferring packets in and out repeatedly. With an automatic data synchronization scheme, NFs are allowed to access the latest packet in both the CPU memory and the GPU memory. Developers are provided with an API to build NFs with no knowledge to the underlying system implementation details.

To flexibly derive the optimal data synchronization plan for each service chain, Gaviss proposes a dynamic fine-grained data synchronization scheme in GPU memory sharing. The way of synchronization is special for reducing data transfer overheads because it exploits NFV domain knowledge and utilizes application layer hints. The dynamic data synchronization scheme takes hints from NFs to derive the optimal synchronization plan for a service chain. With fine-grained hints on each packet field, such as source IP address in IP header, Gaviss only transfers modified fields instead of synchronizing entire packets. This avoids unnecessary

Authorized licensed use limited to: FUDAN UNIVERSITY. Downloaded on June 10,2024 at 11:12:27 UTC from IEEE Xplore. Restrictions apply.

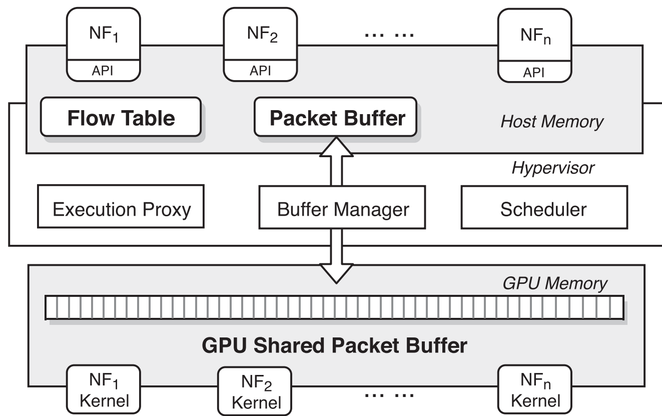


Fig. 3. Gaviss architecture.

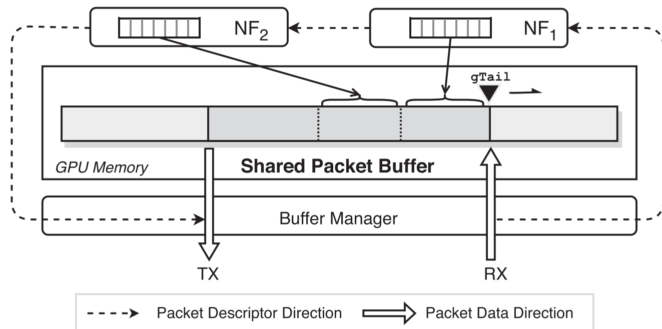


Fig. 4. Shared packet buffer in GPU memory. Show case for a 2-NF service chain.

synchronizations and significantly saves the amount of transferred data. Moreover, by knowing the read and write behaviors of each NF in the service chain, Gaviss further avoids transferring modified packet fields that will not be read by the following NF. The scheme fully exploits potential opportunities in an NFV system to reduce both the number of data transfers and the amount of transferred data.

3.2 System Architecture

Fig. 3 shows the system architecture of Gaviss. Gaviss adopts the state-of-the-art GPU virtualization techniques including spatial GPU sharing and API remoting. Different from previous works, the overall architecture of Gaviss is designed and optimized for effective data sharing. There are three major functional components in the hypervisor layer: *Buffer Manager*, *Execution Proxy*, and *Scheduler*. Buffer Manager manages the shared GPU memory space and controls the data transfer between the host memory and the GPU memory. Execution Proxy is in charge of GPU virtualization, which creates a common GPU execution context for all NFs so that their kernels can share the GPU memory space and execute simultaneously. It receives GPU execution requests from NFs or Buffer Manager, and then executes the corresponding commands. Scheduler monitors the performance of NFs, allocates GPU resources dynamically.

Gaviss adopts a holistic approach to apply the zero-copy design principle in both the GPU memory and the host memory. As shown in Fig. 3, a packet buffer in the GPU memory is shared among NFs, allowing packets in the buffer to be accessed and processed by different NF kernels. In the host memory, there is a shared packet buffer and a flow table for fast data accessing. Packets received from the NIC are stored in the host packet buffer, and the parsed state information is stored in the flow table. With a shared network stack that performs common packet processing procedures like parsing and flow state management, NFs can access the packet fields and shared states directly. For flexible development, NFs are not responsible for transferring packets and managing host and GPU memory. Instead, Buffer Manager batches received packets and transfers them into the GPU memory. With hints passed by NFs, Buffer Manager employs a scheme to automatically synchronize packet data between the host and the GPU packet buffer, so that both the CPU stage and the GPU stage of an NF can access the latest packets if needed. By effectively sharing packets in a service chain, Gaviss avoids extra PCIe data transfers to unleash the overall performance, and NFs are provided with a consistent view of packets in both CPU and GPU.

4 System Design and Implementation

The basic idea of GPU memory sharing is to manage a shared packet buffer for incoming packets in the GPU memory and allow GPU kernels from different NFs to directly perform packet processing operations on the packets. In this section, we demonstrate system design and implementation of Gaviss.

4.1 Shared GPU Ring Buffer

Sharing packets in the GPU memory requires the system to perform memory management on GPU packet buffers. In host memory management solutions like DPDK [18], packet buffers are managed independently, where a buffer is directly freed or reclaimed once the packet is dropped by an NF. However, such a fine-grained memory management scheme should not be simply adopted by GPU-accelerated systems. As the PCIe data transfer throughput of small memory regions can be several orders of magnitude lower than that of large continuous ones, packets are batched and transferred into a large buffer in the GPU memory. The reclamation of the buffer requires the system to confirm that all the packets in it will no longer be used by any NF. For a service chain that contains branches or has NFs dropping packets, the overhead of maintaining the status of all packets is high, which would significantly impact the overall performance.

In Gaviss, we design *Shared Packet Buffer* (SPB), a first-in-first-out ring buffer in the GPU memory shared by all NFs in a service chain. As shown in Fig. 4, SPB is a pre-allocated large continuous memory region, which is initialized at system startup. Unlike previous ring buffers, SPB only maintains one pointer called *gTail*, which points to the tail of the queue. Incoming packets are directly transferred to the memory space referenced by *gTail*, overwriting obsolete packets in the buffer circularly. We use a calculated buffer size to guarantee that the overwritten packets have already been processed by all NFs, so that their memory space can be reused safely. In general, for a service chain with k branches, all NFs in the service chain share the same SPB. With the designed maximum system throughput as T and

Authorized licensed use limited to: FUDAN UNIVERSITY. Downloaded on June 10,2024 at 11:12:27 UTC from IEEE Xplore. Restrictions apply.

the allowed maximum latency of the ith branch as L_i, the size of SPB is calculated as

$$S = T \times \max\{L_i, i = 1, 2, \ldots, k\} \times \alpha \quad (1)$$

For instance, with $T = 100$ Gbps and $L_{max} = 10$ ms, the size of SPB is calculated as 200 MB with α set to 1.6. Since commodity GPUs are generally equipped with more than 4GB device memory, the scheme can easily scale up in an NFV system with even higher throughput.

4.2 GPU Buffer Management

In Gaviss, we develop *Buffer Manager*, a functional component in the hypervisor that is in charge of data transfer and GPU buffer management. Buffer Manager batches packets received from NICs and transfers them to the GPU memory referenced by *gTail*. Packets stay in the GPU memory and are transferred back to the host memory after NFs in a service chain have performed corresponding operations. The performance of packet data transfer heavily depends on the batch size, where a larger batch size lead a much higher throughput but also suffers higher latency. At run time, Buffer Manager dynamically set the batch size to ensure the input PCIe data transfer can meet the performance requirement while minimizing the latency.

Because NFs may drop packets based on their policies (such as firewall and NIDS), packets to be processed by a GPU kernel may not be stored in a continuous memory region. To support random accesses in the packet buffer, a pointer to the packet's location in the GPU memory called *gpointer* is used. gpointers are generated by Buffer Manager when the packet HtoD transfer is performed, and both the gpointer and the *cpointer* (pointer to the same packet in the host memory) of a packet are recorded in its packet descriptor. Pointers to packet descriptors are passed among NFs, while each NF passes an array of gpointers of designated packets to GPU kernels for data processing.

After being processed by all NF kernels, packets should be transferred back into the host memory and sent via a NIC. As some packets may be dropped by NFs in the service chain, the continuous SPB can be split into several small memory regions. If we selectively transfer packets which are not dropped, the memory copy batch size can be much smaller, but will also result in low PCIe transfer performance. In Gaviss, Buffer Manager transfers continuous memory regions back to the host memory, including the dropped packets. With the packet descriptors passed from the last NF of the service chain, Buffer Manager only processes and forwards valid packets, where dropped packets are skipped. This scheme delivers a robust and predictable performance.

4.3 CPU-GPU Data Synchronization

In a GPU-accelerated NFV system, the CPU and the GPU may access and process packets in turn by multiple NFs in a service chain. The modification made to packets in the memory of one processor should be synchronized into that of the other processor to ensure the correctness of the following packet processing procedures. For instance, packets decrypted by an IPsec gateway have to be transferred into the host memory if the following NF needs to access packets in its CPU processing stage.

To free NF developers from handling such complex data synchronizations and minimize the overhead, Gaviss provides a dynamic CPU-GPU data synchronization scheme by which NFs always see the latest and consistent packet data in both the host memory and the GPU memory. Our provided API allows NFs to pass their read-write hints to notify Gaviss about their CPU and GPU behaviors in packet processing. For instance, *gpu_write* denotes that the GPU kernels of an NF would modify packets. Simply applying the hints on the entire packet, however, would result in unnecessary data synchronizations. For instance, if an NF writes packet payloads in the GPU while the following NFs only read the packet headers in the CPU, packets do not need to be synchronized. In Gaviss, four 32-bit vectors are used to represent the CPU/GPU behavior on the fields in the packet header and the packet payload, i.e., CPU read (CR), CPU write (CW), GPU read (GR) and GPU write (GW). Each bit in a vector represents the CPU/GPU behavior on a field of the packet header or the payload. For instance, setting the 5th bit of GW to 1 means that this NF would modify the destination IP address in the GPU kernel.

Algorithm 1. Data Synchronization Plan Generation

1: $n \leftarrow$ Number of NFs in the service chain
2: $RV_i \leftarrow (GR_i \ll 32) \mid CR_i$
3: $WV_i \leftarrow (CW_i \ll 32) \mid GW_i$
4: $LMASK \leftarrow$ 0xFFFFFFFF
5: $HMASK \leftarrow LMASK \ll 32$
6: $plan_1 \leftarrow 0$
7: **for** $i = 1 \rightarrow n$ **do**
8: $\quad plan_1 \leftarrow plan_1 \mid (RV_i \& HMASK) \mid (WV_i \ll 32)$
9: **end for**
10: $dirty \leftarrow WV_1$.
11: **for** $i = 2 \rightarrow n$ **do**
12: $\quad plan_i \leftarrow dirty \& RV_i$
13: $\quad dirty \leftarrow (dirty \oplus plan_i) \mid WV_i$
14: **end for**
15: $lastPlan \leftarrow dirty \& LMASK$
16: **return** $plan, lastPlan$

Based on the read and write vectors of NFs in the service chain, Buffer Manager performs Algorithm 1 to derive the places where synchronizations should be performed. With the four vectors passed through APIs, the system derives a 64-bit read vector (RV_i) and a 64-bit write vector (WV_i) for NF $_i$ (lines 2-3). $plan_i$ denotes the data synchronization plan for NF$_i$, whose higher and lower 32 bits indicate the packet fields that are required to be synchronized from host to device and device to host, respectively. The algorithm first identifies the packet fields that would be accessed by the GPU in NFs of a service chain (lines 7-9). The $dirty$ vector is initialized to WV_1 to denote the modified fields by NF_1 (line 10). $dirty$ denotes the packet fields that are modified but have not been synchronized in the host memory and the GPU memory. For the i th NF, $plan_i$ is calculated as $dirty\&RV_i$, which gets the modified packet fields that will be accessed by NF$_i$ but are not synchronized yet (line 12). The $dirty$ vector is updated at line 13, which marks the modified fields and resets the bits of synchronized fields as 0.

Authorized licensed use limited to: FUDAN UNIVERSITY. Downloaded on June 10,2024 at 11:12:27 UTC from IEEE Xplore. Restrictions apply.

Finally, Buffer Manager identifies the fields that should be transferred into the host memory for forwarding, which are recorded in the lower 32 bits of $lastPlan$(line 15). The scheme transfers data on a per-field basis to reduce the amount of transferred data. By transferring the dirty fields of multiple packets in a batch, which is a common practice in GPU acceleration, the PCIe transfer can still achieve good performance. By performing synchronizations according to the plans, Gaviss enables flexible NF implementations and avoids unnecessary data transfers.

4.4 System Framework

When repetitive PCIe data transfers are avoided with data sharing, the performance of the GPU pipeline stage is further enhanced. As the overhead of per-packet processing and data movement is non-trivial [19], the CPU stage can be unable to provide enough tasks to fully exploit GPUs. Therefore, after Gaviss enhances GPU performance with memory sharing, the CPU performance should also be improved to balance the pipeline. Gaviss alleviates the CPU overhead by 1) avoiding repetitive packet parsing among NFs, and 2) minimizing data copying overheads in the host memory.

In Gaviss, we implement a shared stack processing module whose data is shared across NFs in a service chain. The processing module parses incoming packets, performs IP defragmentation, manages the flow states and providing entries for per-flow NF state storage in shared host memory. Removing repetitive and redundant packet processing to improve the performance of an NFV system have proven to be effective previous studies [15], [20]. In the shared stack, information encoded in the packet descriptor includes L3 and L4 header pointers with which protocol fields can be accessed by adding an offset. There is a pointer to the current flow in the flow table. Besides common flow states, maintaining NF states is demanded for NFs such as automaton states in an NIDS and AES keys in an IPsec gateway. In the flow table, a field is reserved for each NF in a flow table entry. The size of the field is 64 bits in our implementation, which is sufficient for storing state in most NFs, and more memory can be allocated to store if needed.

Besides shared packet information, the framework applies zero-copy principle through its design. When launching a GPU kernel, buffers that store packets and input parameters are shared between NFs and the hypervisor. Execution Proxy in the hypervisor directly transfers the buffers to the GPU memory, avoiding extra memory copies in the host memory. Moreover, with Buffer Manager managing packet lifecycle in the host memory, only packet pointers are passed between NFs. Gaviss initiates an input queue for each NF, where an NF directly inserts packet pointers into the queue of the next NF with provided APIs. For the last NF on the node, the framework automatically sends packets via NICs. In summary, the framework of Gaviss is designed to balance the CPU-GPU pipeline, thereby effectively utilizing the performance advantage brought by GPU data sharing.

5 Experiments

5.1 Experimental Setup

Platform. We evaluate Gaviss on a server with an Intel Xeon E5-2695 v4 CPU (2.10 GHz, 18 physical cores) and 128 GB main memory (DDR4, 2400MT/s, 16GB x8). The server is equipped with a dual-port Intel XL710 40GbE NIC for network I/O. An NVIDIA Titan X (Pascal) GPU with 28 streaming multiprocessors (3584 CUDA cores in total) is installed. The server runs on Ubuntu 16.04 with Linux kernel 4.15.0-43-generic. Docker 18.09.3 is used as our virtualization platform, with each NF running as a Docker instance. The GPU kernels of NFs are compiled and executed with NVIDIA CUDA Toolkit 9.2, and the Gaviss infrastructure and the NF binaries are compiled with gcc 7.3.0.

System Implementation. We implement Gaviss based on OpenNetVM [21] with Intel DPDK 18.02 for userspace packet I/O and queue operations. The implementation of the shared stack processing module is based on Libnids 1.24 [22]. The hypervisor layer of Gaviss is deployed as a system process, with its major components running as separate threads, including Buffer Manager, Execution Proxy and Scheduler. Each NF takes 2 threads to execute, one for CPU packet processing stages and the other for GPU kernel function execution requests submission. Each thread is bound to a unique CPU core with *sched_setaffinity*, which avoids context switching initiated by the scheduling mechanism in the operating system. NFs communicate with the Gaviss infrastructure in the hypervisor layer with shared memory provided by DPDK's multi-process support.

NFs and Service Chains. We implement four NFs with GPU acceleration for our experimental evaluation: (a) *Router* is an IPv4 router that performs DIR-24-8-BASIC lookup algorithm [23] in packet routing; (b) *Firewall* performs packet filtering with bit vector linear search on the 5-tuple of packets [24]; (c) *IPsec gateway* performs packet encryption with HMAC-SHA1 and AES-128 (CTR mode) algorithms; (d) *NIDS* performs deep packet inspection with the Aho-Corasick algorithm [25] consisting of 147 rules. Composed by the NFs, four service chains are used in the evaluation of Gaviss. *C1:* NIDS - IPsec gateway, *C2:* NIDS - Router - IPsec gateway, *C3:* Firewall - NIDS - IPsec gateway, *C4:* Firewall - NIDS - Router - IPsec gateway.

Test Traffic. We implement a traffic generator based on DPDK, which runs on a separate server equipped with a dual-port Intel XL710 40 GbE NIC. The two servers are connected with a 40 Gbps optical fiber. The packet generator consistently generates TCP packets with fixed pattern payload and different destination IP addresses to simulate packets from multiple TCP flows. The input traffic consists of 10,000 TCP flows by default unless specified.

5.2 System Throughput

In this subsection, we show the throughput that Gaviss achieves on our evaluation platform. GPU processing requires batching to achieve high throughput, but a larger batch size also results in higher latency in the packet processing pipeline. In the experiments, we set 1 ms (ms) as the latency upper bound for each NF. According to the input network speed and the latency requirements, Scheduler would allocate GPU resources for each NF to meet the requirements. The method of calculating the demanded GPU resource is the same as that in G-NET [4], which uses the regression method to approximate the computational

Authorized licensed use limited to: FUDAN UNIVERSITY. Downloaded on June 10,2024 at 11:12:27 UTC from IEEE Xplore. Restrictions apply.

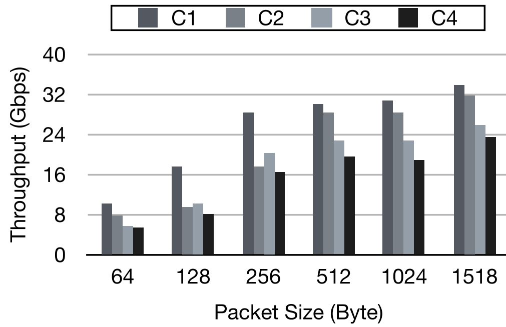


Fig. 5. System throughput.

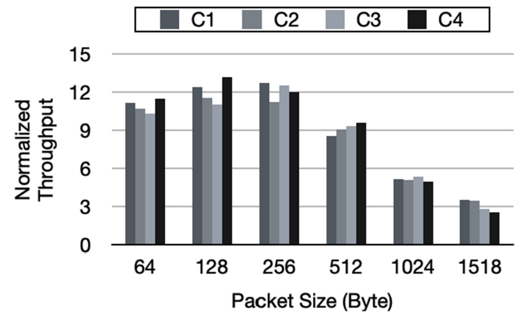


Fig. 6. Throughput improvements of Gaviss compared with the baseline.

resource needed for each NF to achieve maximum performance within a certain latency bound.

Fig. 5 shows the system throughput of Gaviss on the four service chains with different packet sizes. As shown in the figure, Gaviss achieves up to 33.8 Gbps throughput on the service chain *C1*, which consists of two network functions, and it achieves up to 23.5 Gbps throughput on the service chain *C4*. Since the GPU kernels used in our experiment are not based on state-of-the-art GPU-optimized algorithms [6], [8], [9], we believe that the throughput of Gaviss can be further unleashed with highly optimized GPU implementations.

As shown in Fig. 5, the throughput of a service chain increases with the packet size. The main reason is that, for the same network speed, more packets need to be processed when packet size is small. As a result, higher per-packet processing overhead is introduced into the CPU processing stage, including packet parsing, protocol stack processing, state retrieval, and enqueue/dequeue operations. In the experiments, the CPU becomes the bottleneck that limits the overall performance for workloads with small packet sizes. With the CPU workload alleviated with larger packet sizes, system throughput increases accordingly.

Fig. 6 shows the throughput improvements of Gaviss over a baseline system. The baseline system for comparison works in the same way as G-NET, which does not share the GPU memory and the protocol stack across NFs. Therefore, NFs in the baseline system have to perform PCIe data transfer and protocol stack processing independently. The only difference is that the baseline system does not have a software switch, and the NFs pass packet descriptor pointers directly to the next NF, which is the same with Gaviss. As shown in the figure, Gaviss achieves 2.6–13.2× higher throughput compared with the baseline system. Specifically, Gaviss achieves 10.3–13.2× higher throughput when the packet size is smaller than or equal to 256 bytes. The reason is that the per-packet stack processing is the major overhead of workloads with small packet sizes, especially in the baseline system since it performs the operations repetitively in each NF. Thus, the workload with small packets can benefit more from the optimized CPU-GPU framework with alleviated CPU processing overhead.

5.3 Comparison With CPU-Based NFV Systems

CPU-Based System Implementation. To investigate the efficiency of CPUs and GPUs in building NFV systems, we build a CPU-based NFV system and compare its performance with Gaviss. The CPU-based system is based on OpenNetVM [21], an open-source NFV solution with the support for building service chains. With OpenNetVM, we built the four network functions optimized with G-Opt [14]. G-Opt is a source-to-source transformation framework that hides DRAM latency for data structure lookup operations with batch prefetching. In order to adopt G-Opt in NF implementations, we made modifications to the NF APIs of OpenNetVM, allowing a user-implemented network function to get access to a batch of input packets in the callback function, instead of one at a time. By applying G-Opt, network functions like *NIDS* and *Firewall* achieve 1-2× throughput improvements for variant packet sizes. For *IPsec gateway*, we use Intel AES-NI [26] hardware instructions to accelerate AES encryption. Compared with the open source implementation in the OpenNetVM repository, our implementation with AES-NI achieves around 10× higher throughput. We also add a network stack in NIDS and Firewall NFs for stateful packet processing, making the CPU-based system have similar workloads with Gaviss.

TABLE 1
The Single Core Performance of Our CPU-Based NFs (Gbps)

Packet Size	64B	128B	256B	512B	1024B	1518B
NIDS	3.8	4.9	5.5	5.9	6.1	7.2
IPsec	0.38	0.61	0.89	1.2	1.4	1.7
Router	6.5	13.0	26.0	38.2	39.1	39.7
Firewall	1.8	3.4	7.1	14.6	29.1	39.4

Single Core Performance. Running on the same platform with Gaviss, the single core performance of our CPU-based NFs is listed in Table 1. As shown, the NF throughput increases with larger packet sizes, due to the lower per-packet processing overheads. The throughputs of NIDS and IPsec gateway are less sensitive to packet sizes compared with other NFs, because they scan packet payloads instead of only accessing packet headers. IPsec gateway achieves the lowest throughput among the four NFs. The compute-intensive operations in IPsec gateway are AES-128-CTR encryption and HMAC-SHA1 authentication digest generation. When the packet size is 1024 bytes, AES-NI enhances encryption performance from 0.3 Gbps to 2.9 Gbps. As the performance of SHA1 operation is 2.8 Gbps, the overall IPsec throughput is halved to 1.4 Gbps. Studies have already demonstrated that GPUs achieve much higher efficiency than CPUs for cryptographic algorithms such as SHA-1 [8], [13].

Throughput Comparison. We evaluate the performance of CPU-based NFV system with the four service chains. In the

Authorized licensed use limited to: FUDAN UNIVERSITY. Downloaded on June 10,2024 at 11:12:27 UTC from IEEE Xplore. Restrictions apply.

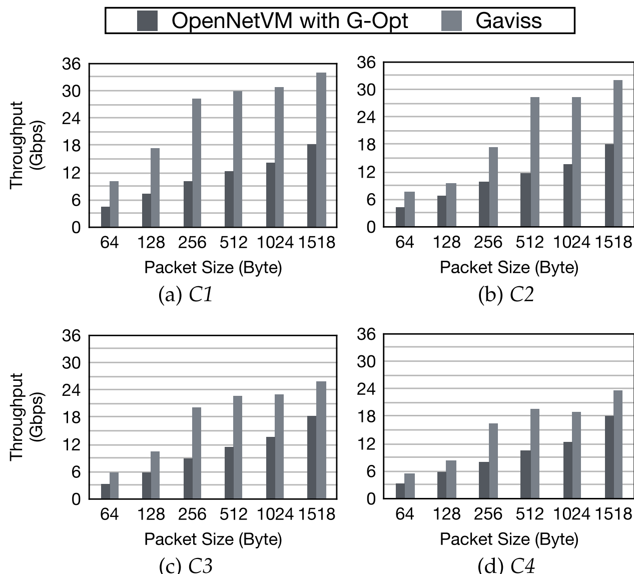


Fig. 7. Performance comparison between CPU and GPU NFV systems on four service chains.

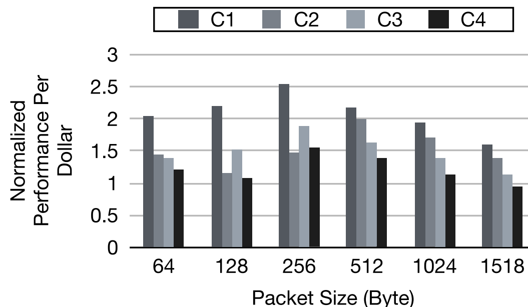


Fig. 8. Comparison of performance-per-dollar of Gaviss and the CPU-based system. Performance-per-dollar values are normalized to that of CPU.

experiments, we manually adjust the core allocations to NFs to maximize the performance of the service chains. With three cores allocated to RX, TX and performance monitor, the rest 15 cores are shared by NFs. The throughput comparison of the CPU-based NFV and Gaviss is shown in Fig. 7. Among the four service chains, Gaviss achieves 1.3-2.8× higher throughput than the CPU-based system. The performance of the CPU-based system is constrained by the IPsec gateway, even when most computational resources are allocated to it (9-13 cores in our experiments). Instead, the IPsec gateway with GPU acceleration can reach up to 37.7Gbps with 1ms NF processing latency. This is also the main reason that Gaviss achieves more significant throughout improvements for service chains with less NFs. When more NFs are added in the service chain, less than 4 cores that are reclaimed from the IPsec gateway would be enough for Router or Firewall to reach the performance requirement, while the performance of IPsec gateway would not be severely influenced. For GPU-based implementations, although spatial sharing is enabled in Gaviss, the API calls and driver operations from NFs are still serialized and performed in a queued manner [27], which overhead is non-trivial. Our profiling results show that there is higher GPU driver latency and more data transfer conflicts with more kernels launched. As a result, for service chains with more NFs, the advantage of Gaviss over CPU-based system would be less notable.

Performance-Cost Ratio. As the two system setups are based on different processors, we adopt the performance-per-dollar metric to compare their efficiency. The release price of the 18-core Intel Xeon E5-2695v4 CPU we use in the evaluation is $2424, and the CPU-based system uses all cores in the experiments. The release price of NVIDIA TITAN X Pascal GPU is $1200, and we add it to the price of CPU cores used in the experimental setup. The per-core price of the CPU is calculated as the total price divided by the number of cores. Fig. 8 compares the performance-per-dollar of Gaviss and the CPU-based system. We can see Gaviss surpasses the CPU-based system in terms of the performance-per-dollar metric for almost all cases, with up to 2.53× improvements. For C4 with 1518-byte packets, as an exception, the performance-per-dollar is 5% lower than that of the CPU-based system. Considering the 30% throughput improvement in the case (as shown in Fig. 7d), GPUs still demonstrate the notable advantages as processors in NFV systems.

Overall, we keep an optimistic outlook for GPU-accelerated NFV systems, for (1) GPUs are good at cryptographic algorithms which are prevalent and important in server workloads; (2) GPUs are still under fast development for more cores and higher performance thus are easier to scale for heavier loads. Our experiments show that GPUs are cost-efficient for building NFV systems.

5.4 Reduced Data Transfer With GPU Memory Sharing

To evaluate the performance improvements brought by the GPU memory sharing technique, we disable the GPU memory sharing in Gaviss as the baseline for comparison, which is called *No-Memshare*. In No-Memshare, GPU memory is not shared, and explicit HtoD and DtoH packet data transfers are required for GPU kernels of each NF. In the evaluation, we set the same key performance parameters in both the systems to get the same GPU kernel performance, including the batch size and the number of utilized GPU streaming multiprocessors.

To understand the performance improvement brought by shared GPU memory, we measure the amount of data transferred in No-Memshare and Gaviss in a fixed period of time. For a fair comparison, the input network speed and the throughput of the two systems are the same. Fig. 9 demonstrates the normalized volume of HtoD transferred data in the two systems. For the four service chains, the GPU memory sharing reduces the amount of transferred data by 24.3–48.4% in the HtoD direction. Note that in No-Memshare, the amount of data transferred in the DtoH direction is the same as Gaviss in our 4 service chain setups, but it can also benefit from GPU memory sharing in certain service chains. For instance, consider a service chain consisting of two NFs that need to alter the packet data within the GPU memory buffer. No-Memshare needs to perform DtoH packet data transfer on each of these NFs. However, Gaviss only needs to perform DtoH data transfer once because they can make the modifications in place. With significantly less amount of transferred data, the GPU memory sharing

Authorized licensed use limited to: FUDAN UNIVERSITY. Downloaded on June 10,2024 at 11:12:27 UTC from IEEE Xplore. Restrictions apply.

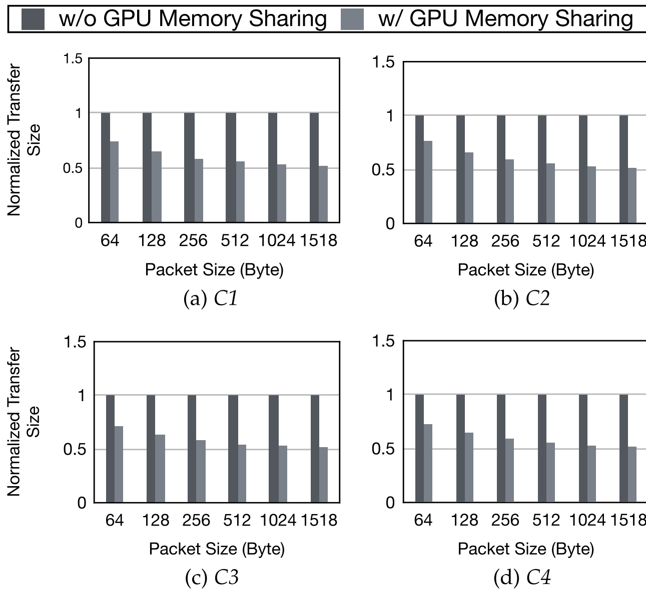


Fig. 9. The reduction of GPU data transfer size.

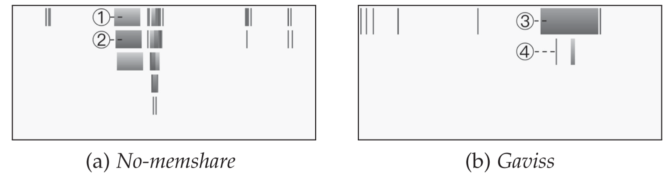


Fig. 10. Alleviated PCIe data transfer conflicts in Gaviss. The horizontal axis represents time and the blocks represents the copy operations.

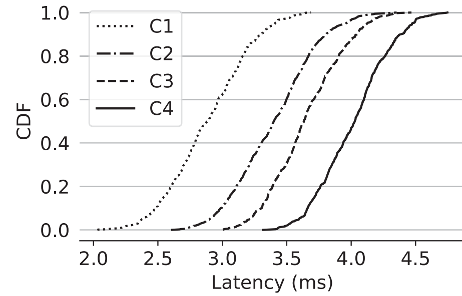


Fig. 11. Gaviss latency of four service chains.

technique has the potential to achieve higher throughput when the PCIe bandwidth becomes a scarce resource in high speed network processing.

As shown in Fig. 9, there is a larger reduction in the data transfer size with larger packet size. The reason lies in the data transfer of NF states, such as AES keys and HMAC keys in IPsec gateway and DFA automaton states in NIDS. With larger packet size, an NF handles less number of packets under the same throughput, which results in less amount of NF states to be transferred in the HtoD direction. Therefore, for workloads with large packet sizes, the relatively reduced amounts of transferred data are larger than those with small packet sizes.

Another influential factor for the performance improvement of memory sharing comes from mitigating GPU data transfer conflicts. As shown in Fig. 2, the average HtoD data transfer latency of a router can be increased by 3.2× by PCIe conflicts. Instead, with the memory sharing in Gaviss, the HtoD transfer latency of the router in chain *C4* is only 2.0× higher, which demonstrates a huge alleviation of PCIe conflicts. Fig. 10 shows part of the profiled timeline of GPU host-to-device data transfers in No-Memshare and Gaviss under the same input traffic. In the figure, the horizontal axis represents the timeline, and each bar represents an HtoD data transfer, while the vertically stacked bars indicate those data transfers are overlapped. ① and ③ in Fig. 10a indicate two individual data transfer operations in No-memshare initiated by two different NFs: IPsec gateway and NIDS. As they are initiated nearly at the same time, there's a data transfer conflict, where ④ can only be performed after ③ completes, leading to higher latency. Different from No-memshare, the HtoD data transfers in Gaviss are initiated by Buffer Manager in Gaviss which is shown as ⑤ in Fig. 10b. In Gaviss, there is less packet data that need to be transmitted on the PCIe, thus the frequency of data transmission conflicts will be lower. Even when there is another NF-issued data transfer ⑦ overlaps with ⑤, ⑥ is roughly unaffected. The reason is that, with GPU memory sharing, the data transferred by NFs are only NF states, whose sizes are typically small. When the transferred data size is below a certain threshold (64KB in some NVIDIA GPUs), the data can be sent via the command stream, avoiding the DMA copy engine being involved[1]. In this way, the executions of NFs are unaffected even when its data transfers overlap with that initiated by Buffer Manager.

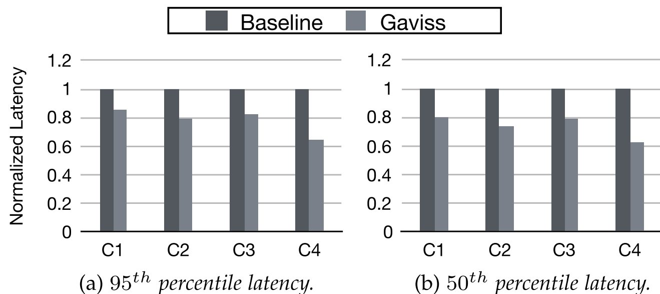
5.5 Latency

Fig. 11 shows the Cumulative Distribution Function (CDF) of the overall packet processing latency in Gaviss. As shown in the figure, the 50th percentile latencies of the four service chains are $2.9ms$, $3.4ms$, $3.6ms$ and $4.0ms$, and the 95th percentile latencies of the four service chains are $3.4ms$, $4.0ms$, $4.1ms$ and $4.5ms$. The system latency comes with a variation of $\pm 0.5ms$ for all the four service chains. This variation mainly comes from packet batching in the processing pipeline. As GPU acceleration requires batch processing to achieve high performance, packets are batched before being transferred to the GPU memory. In a batch, some packets arrive earlier than others, which leads to latency variations.

Fig. 12 shows the latency reduction of Gaviss compared with the baseline system which has no GPU memory sharing and shared stack processing, given the same input throughput. Gaviss dramatically reduces the 50th percentile latency and the 95th percentile latency by up to 35.2% and 37.9%, respectively. By utilizing the GPU memory sharing and shared stack processing, Gaviss eliminates the repetitive packet processing and PCIe transfer overhead in the service chains, which are the major sources of system latency (described in Section 2.2). In addition, the alleviation of PCIe conflicts also reduces the average waiting time of data transfer. As shown in the figure, Gaviss has a larger latency reduction on longer service chains. For instance, 14.9–19.8% and 35.2–37.9% of the 95th percentile latencies are reduced for *C1* and *C4*, respectively. This is because the two more NFs in *C4* almost double the PCIe data transfer and per-packet processing overheads. Therefore, the data

1. NVIDIA DevTalk Post:http://bit.ly/2kBHdQ6

Authorized licensed use limited to: FUDAN UNIVERSITY. Downloaded on June 10,2024 at 11:12:27 UTC from IEEE Xplore. Restrictions apply.



(a) 95th *percentile latency.* (b) 50th *percentile latency.*

Fig. 12. Latency improvements of Gaviss.

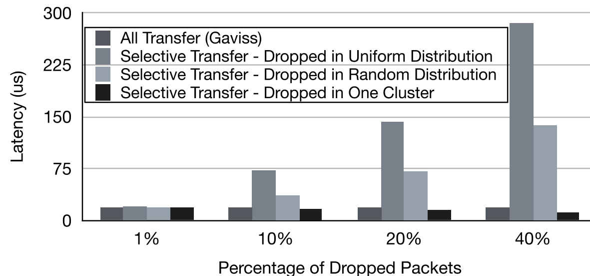


Fig. 13. Performance comparison of data transfer schemes.

sharing techniques in Gaviss make it more efficient than the state-of-the-art GPU-accelerated implementations, when processing packets in long service chains.

5.6 Performance of Batched Data Transfer

In a service chain, packets may be dropped by certain NFs such as Firewall and NIDS, where the dropped packets may split the buffer into several parts. Because transferring a small amount of data would result in low PCIe performance, the Buffer Manager in Gaviss transfers all processed packets and lets CPU decides which packets to drop. In Fig. 13, we compare the efficiency of our scheme (*all-transfer*) with *selective-transfer*, i.e., without transferring dropped packets. We evaluate the performance in a 200 KB buffer with a workload of 1024-byte packets. As shown in the figure, when dropped packets are uniformly or randomly distributed in the buffer, the speed of *selective-transfer* can be up to 15.3 times slower than the *all-transfer* scheme in Gaviss (40% packets are dropped). It performs worst when dropped packets are uniformly distributed, because the buffer is divided into many small regions for data transferring. Only when the dropped packets gather as one cluster in the buffer, the performance of *selective-transfer* can be slightly higher than that of Gaviss. But this is the optimal situation which is almost unlikely to happen. Overall, the evaluation proves that the *all-transfer* scheme adopted by Gaviss achieves almost ideal performance with only simple implementations.

6 Related Work and Discussion

6.1 Related Works

CPU-Based NFV Systems. NetVM [17] is a CPU-based NFV system that utilizes data sharing in the host memory. In the system, packets are shared among NFs and are transferred by only passing pointers. As the followup work, Microboxes [28] exposes publish/subscribe-based APIs to facilitate modular NF implementation. Data sharing can bring significant performance advantages for NFV systems where NFs in a service chain cooperatively process packets. Gaviss is the first system that adopts data sharing in GPU memory.

In NFV systems, virtualization allows the deployment of NFs from different vendors with different software dependencies, independently management of resources, instantiating and consolidating NF instances with ease, and at the same time, providing isolation for the NF runtime [17]. While virtualization is the de-facto standard in the NFV domain, it may have an impact on the system performance. There is another class of work including OpenBox [15], SNF [16], Metron [1] and Netbricks [2] that improves packet processing performance by discarding virtualization. Although these approaches can ensure NF execution isolation by providing proper programming abstractions, other benefits brought by virtualization are discarded.

GPU-Accelerated Network Function and NFV Systems. Developing high performance NFs with GPUs has been studied by a wide class of work, including router [9], NIDS [6], NDN system [29], SSL reverse proxy [8] and network monitoring [30]. G-NET [4] and Grus [5] are NFV systems that utilize GPUs to enhance system performance. However, both of them face the inefficiencies described in Section 2.2. FlowShader [31] concentrates on exploiting GPUs for L7 flow processing in NFs. APUNet [13] addresses the PCIe transfer overheads by utilizing APUs, where a GPU is integrated on the same die with a CPU. The CPU and the GPU share the same physical memory to eliminate the data transfer overhead. This is a promising solution. However, the performance of current APUs is still much lower than that of discrete GPUs. Since discrete GPUs are widely deployed in enterprise infrastructures, PCIe data transfers are still required in utilizing GPUs for high performance network processing.

A bunch of works study and optimize the performance of network stacks for end hosts [32], [33], [34] and middleboxes [15], [20], [28], [35], [36]. The goal of adopting a shared stack in Gaviss is to unleash the power of GPUs. There're research efforts like GASPP [34] and GPUnet [37] that implement GPU-based protocol stacks, but they are targeted for pure GPU-based network functions. Instead, Gaviss is designed for enhancing NFV system performance on CPU-GPU heterogeneous architectures, where the CPU is in charge of stateful packet processing in the pipeline.

6.2 Processors for Implementing NFs

Previous studies have shown that both CPUs and GPUs are good at building certain types of NFs [13], [14]. GPUs always demonstrate higher performance for compute-intensive algorithms such as RSA decryption, while CPUs, with techniques including memory prefetching, perform better for memory-intensive algorithms such as IPv4 forwarding [13]. Therefore, we believe the hybrid NFV system with a mix of CPU-based NFs and GPU-based NFs has the potential of achieving higher performance. However, this proposes new challenges because a CPU-based NF and a GPU-based NF process packets in the host memory and the GPU device memory, respectively. To allow CPU-GPU co-processing, it requires providing the latest copy of packets in the

Authorized licensed use limited to: FUDAN UNIVERSITY. Downloaded on June 10,2024 at 11:12:27 UTC from IEEE Xplore. Restrictions apply.

corresponding memory space to make an NF work. As a result, a new bi-directional synchronization mechanism needs to be designed to address the new challenge. We leave it as a future work.

A Data Processing Unit (DPU) or a SmartNIC is a NIC with integrated cores and memory to directly process received packets. We find that current SmartNICs do not fit for high performance network processing because the processor in a SmartNIC is much weaker than commodity off-the-shelf CPUs and GPUs. In our evaluation of a Mellanox Bluefield-2 SmartNIC with 16 ARM cores, its performance for running NFs is an order of magnitude slower than either CPUs or GPUs. In our evaluation, the SmartNIC is incompetent in building complex service chains for 1 Gbps+ network, not mentioning the 10Gbps+ speed achieved in our paper. Instead, a DPU or a SmartNIC fits for low latency packet processing because packets do not need to be moved to the host memory through the PCIe bus. In the future, for DPUs with a custom instruction set and coupled hardware acceleration, we believe the performance can be significantly enhanced.

7 Conclusion

Having conducted thorough experiments and analyses, we have identified the bottleneck of GPU-accelerated NFV systems, which is the PCIe conflict and overall performance limitations imposed by repetitive PCIe data transfers. We have made a strong case by designing and implementing Gaviss for effectively sharing data in the GPU memory among NFs to address the bottleneck. Our evaluation results show that Gaviss achieves 5.0–13.2× higher throughput while reducing the latency by up to 37.9%, compared with state-of-the-art GPU-accelerated NFV system solutions. Moreover, Gaviss achieves 1.1–2.5× higher performance-per-cost than CPU-based NFV implementations on service chains with computational intensive NFs.

References

[1] G. P. Katsikas, T. Barbette, D. Kostić, R. Steinert, and G. Q. M. Jr, “Metron: NFV service chains at the true speed of the underlying hardware,” in *Proc. 15th USENIX Symp. Netw. Syst. Des. Implementation*, 2018, pp. 171–186. [Online]. Available: https://www.usenix.org/conference/nsdi18/presentation/katsikas

[2] A. Panda, S. Han, K. Jang, M. Walls, S. Ratnasamy, and S. Shenker, “NetBricks: Taking the V out of NFV,” in *Proc. 12th USENIX Symp. Operating Syst. Des. Implementation*, 2016, pp. 203–216. [Online]. Available: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/panda

[3] Z. Ni, G. Liu, D. Afanasev, T. Wood, and J. Hwang, “Advancing network function virtualization platforms with programmable nics,” in *Proc. IEEE Int. Symp. Local Metrop. Area Netw.*, 2019, pp. 1–6.

[4] K. Zhang *et al.*, “G-NET: Effective GPU sharing in NFV systems,” in *Proc. 15th USENIX Symp. Netw. Syst. Des. Implementation*, 2018, pp. 187–200. [Online]. Available: https://www.usenix.org/conference/nsdi18/presentation/zhang-kai

[5] Z. Zheng *et al.*, “Grus: Enabling latency slos for GPU-accelerated nfv systems,” in *Proc. IEEE 26th Int. Conf. Netw. Protoc.*, 2018, pp. 154–164.

[6] M. A. Jamshed *et al.*, “Kargus: A highly-scalable software-based intrusion detection system,” in *Proc. ACM Conf. Comput. Commun. Secur.*, 2012, pp. 317–328.

[7] G. Vasiliadis, M. Polychronakis, and S. Ioannidis, “MIDeA: A multi-parallel intrusion detection architecture,” in *Proc. 18th ACM Conf. Comput. Commun. Secur.*, 2011, pp. 297–308.

[8] K. Jang, S. Han, S. Han, S. B. Moon, and K. Park, “SSLShader: Cheap SSL acceleration with commodity processors,” in *Proc. 8th USENIX Symp. Netw. Syst. Des. Implementation*, 2011, pp. 1–14.

[9] S. Han, K. Jang, K. Park, and S. Moon, “PacketShader: A GPU-accelerated software router,” in *Proc. ACM SIGCOMM Conf.*, 2010, pp. 195–206.

[10] D. Tarjan, K. Skadron, and P. Micikevicius, “The art of performance tuning for cuda and manycore architectures,” *Birds-of-A-Feather Session SC*, vol. 9, 2009. [Online]. Available: https://www.cs.virginia.edu/~skadron/Papers/cuda_tuning_bof_sc09_final.pdf

[11] J. Hestness, S. W. Keckler, and D. A. Wood, “GPU computing pipeline inefficiencies and optimization opportunities in heterogeneous CPU-GPU processors,” in *Proc. IEEE Int. Symp. Workload Characterization*, 2015, pp. 87–97.

[12] B. Van Werkhoven, J. Maassen, F. J. Seinstra, and H. E. Bal, “Performance models for CPU-GPU data transfers,” in *Proc. 14th IEEE/ACM Int. Symp. Cluster Cloud Grid Comput.*, 2014, pp. 11–20.

[13] Y. Go, M. A. Jamshed, Y. Moon, C. Hwang, and K. Park, “Apunet: Revitalizing GPU as packet processing accelerator,” in *Proc. 14th USENIX Symp. Netw. Syst. Des. Implementation*, 2017, pp. 83–96. [Online]. Available: https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/go

[14] A. Kalia, D. Zhou, M. Kaminsky, and D. G. Andersen, “Raising the bar for using gpus in software packet processing,” in *Proc. 12th USENIX Symp. Netw. Syst. Des. Implementation*, 2015, pp. 409–423. [Online]. Available: https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/kalia

[15] A. Bremler-Barr, Y. Harchol, and D. Hay, “Openbox: A software-defined framework for developing, deploying, and managing network functions,” in *Proc. ACM SIGCOMM Conf.*, 2016, pp. 511–524.

[16] G. P. Katsikas, M. Enguehard, M. Kuźniar, G. Q. Maguire, Jr., and D. Kostić, “Snf: Synthesizing high performance nfv service chains,” *PeerJ Comput. Sci.*, vol. 2, 2016, Art. no. e98.

[17] J. Hwang, K. K. Ramakrishnan, and T. Wood, “NetVM: High performance and flexible networking using virtualization on commodity platforms,” in *Proc. 11th USENIX Symp. Netw. Syst. Des. Implementation*, 2014, pp. 445–458. [Online]. Available: https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/hwang

[18] D. Scholz, “A look at intel’s dataplane development kit,” *Network*, vol. 115, pp. 115–122, 2014.

[19] D. D. Clark, V. Jacobson, J. Romkey, and H. Salwen, “An analysis of TCP processing overhead,” *IEEE Commun. Mag.*, vol. 27, no. 6, pp. 23–29, Jun. 1989.

[20] V. Sekar, N. Egi, S. Ratnasamy, M. K. Reiter, and G. Shi, “Design and implementation of a consolidated middlebox architecture,” in *Proc. 9th USENIX Symp. Netw. Syst. Des. Implementation*, 2012, pp. 323–336. [Online]. Available: https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/sekar

[21] W. Zhang *et al.*, “OpenNetVM: A platform for high performance network service chains,” in *Proc. Workshop Hot Top. Middleboxes Netw. Function Virtualization*, 2016, pp. 26–31.

[22] R. Wojtczuk, “Libnids,” 2010. [Online]. Available: https://github.com/korczis/libnids

[23] P. Gupta and N. McKeown, “Packet classification on multiple fields,” *ACM SIGCOMM Comput. Commun. Rev.*, vol. 29, no. 4, pp. 147–160, 1999.

[24] T. Lakshman and D. Stiliadis, “High-speed policy-based packet forwarding using efficient multi-dimensional range matching,” in *ACM SIGCOMM Comput. Commun. Rev.*, vol. 28, no. 4, 1998, pp. 203–214.

[25] A. V. Aho and M. J. Corasick, “Efficient string matching: An aid to bibliographic search,” *Commun. ACM*, vol. 18, no. 6, pp. 333–340, 1975.

[26] J. Rott, “Intel advanced encryption standard instructions (aes-ni),” Oct. 2019. [Online]. Available: https://software.intel.com/en-us/articles/intel-advanced-encryption-standard-instructions-aes-ni

[27] D. Lustig and M. Martonosi, “Reducing gpu offload latency via fine-grained CPU-GPU synchronization,” in *Proc. IEEE 19th Int. Symp. High Perform. Comput. Archit.*, 2013, pp. 354–365.

[28] G. Liu, Y. Ren, M. Yurchenko, K. Ramakrishnan, and T. Wood, “Microboxes: High performance NFV with customizable, asynchronous tcp stacks and dynamic subscriptions,” in *Proc. Conf. ACM Special Int. Group Data Commun.*, 2018, pp. 504–517.

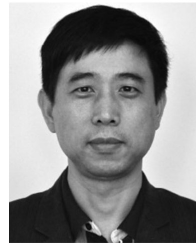
[29] Y. Wang *et al.*, “Wire speed name lookup: A gpu-based approach,” in *Proc. 10th USENIX Symp. Netw. Syst. Des. Implementation*, 2013, pp. 199–212. [Online]. Available: https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/wang_yi

Authorized licensed use limited to: FUDAN UNIVERSITY. Downloaded on June 10,2024 at 11:12:27 UTC from IEEE Xplore. Restrictions apply.

[30] W. Wu, P. DeMar, D. J. Holmgren, A. Singh, and R. Pordes, "G-NetMon: A GPU-accelerated network performance monitoring system for large scale scientific collaborations," 2011. [Online]. Available: http://arxiv.org/abs/1108.1785
[31] X. Yi *et al.*, "FlowShader: A generalized framework for GPU-accelerated VNF flow processing," in *Proc. 27th IEEE Int. Conf. Netw. Protoc.*, 2019, pp. 1–12.
[32] E. Jeong *et al.*, "mTCP: A highly scalable user-level TCP stack for multicore systems," in *Proc. 11th USENIX Symp. Netw. Syst. Des. Implementation*, 2014, pp. 489–502. [Online]. Available: https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/jeong
[33] R. Laufer, M. Gallo, D. Perino, and A. Nandugudi, "Climb: Enabling network function composition with click middleboxes," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 46, no. 4, pp. 17–22, 2016.
[34] G. Vasiliadis, L. Koromilas, M. Polychronakis, and S. Ioannidis, "GASPP: A GPU-accelerated stateful packet processing framework," in *Proc. USENIX Annu. Tech. Conf.*, 2014, pp. 321–332. [Online]. Available: https://www.usenix.org/conference/atc14/technical-sessions/presentation/vasiliadis
[35] M. A. Jamshed, Y. Moon, D. Kim, D. Han, and K. Park, "mOS: A reusable networking stack for flow monitoring middleboxes," in *Proc. 14th USENIX Symp. Netw. Syst. Des. Implementation*, 2017, pp. 113–129. [Online]. Available: https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/jamshed
[36] T. Barbette, "Architecture for programmable network infrastructure," Ph.D. dissertation, University of Liege, Liége, Belgium, 2018.
[37] M. Silberstein *et al.*, "Gpunet: Networking abstractions for GPU programs," *ACM Trans. Comput. Syst.*, vol. 34, no. 3, 2016, Art. no. 9.

Liangchen Guo received the master's degree from the school of software engineering, Fudan University. His research interests include parallel and distributed computing, network function virtualization, and high performance computing.

Kai Zhang received the PhD degree from the University of Science and Technology of China in 2016. He is an associate professor with the School of Computer Science and Engineering, Fudan University. He was a research fellow with the National University of Singapore from 2016 to 2017. He was a visiting scholar with The Ohio State University from 2013 to 2015. His research interests are mainly in the fields of parallel and distributed computing and database systems.



X. Sean Wang (Senior Member, IEEE) received the PhD degree in computer science from the University of Southern California. Heis a distinguished professor with the School of Compute Science, Fudan University, Shanghai, China. Before joining Fudan University in 2011, he was the dorothean chair professor in computer Sscience with the University of Vermont. His research interests include data systems and data security. He is the fellow of CCF and the member of ACM.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/csdl.

Authorized licensed use limited to: FUDAN UNIVERSITY. Downloaded on June 10,2024 at 11:12:27 UTC from IEEE Xplore. Restrictions apply.