G-NET: Effective GPU Sharing in NFV Systems

Kai Zhang, *Fudan University;* Bingsheng He, *National University of Singapore;* Jiayu Hu, *University of Science and Technology of China;* Zeke Wang, *National University of Singapore;* Bei Hua, Jiayi Meng, and Lishan Yang, *University of Science and Technology of China*

https://www.usenix.org/conference/nsdi18/presentation/zhang-kai

This paper is included in the Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI '18).

April 9–11, 2018 • Renton, WA, USA

ISBN 978-1-939133-01-4

G-NET: Effective GPU Sharing in NFV Systems

Kai Zhang[*], Bingsheng He[‡], Jiayu Hu[†], Zeke Wang[‡], Bei Hua[†], Jiayi Meng[†], Lishan Yang[†]
[*]*Fudan University* [‡]*National University of Singapore*
[†]*University of Science and Technology of China*

Abstract

Network Function Virtualization (NFV) virtualizes software network functions to offer flexibility in their design, management and deployment. Although GPUs have demonstrated their power in significantly accelerating network functions, they have not been effectively integrated into NFV systems for the following reasons. First, GPUs are severely underutilized in NFV systems with existing GPU virtualization approaches. Second, data isolation in the GPU memory is not guaranteed. Third, building an efficient network function on CPU-GPU architectures demands huge development efforts.

In this paper, we propose G-NET, an NFV system with a GPU virtualization scheme that supports spatial GPU sharing, a service chain based GPU scheduler, and a scheme to guarantee data isolation in the GPU. We also develop an abstraction for building efficient network functions on G-NET, which significantly reduces development efforts. With our proposed design, G-NET enhances overall throughput by up to 70.8% and reduces the latency by up to 44.3%, in comparison with existing GPU virtualization solutions.

1 Introduction

Network Function Virtualization is a network architecture for virtualizing the entire class of network functions (NFs) on commodity off-the-shelf general-purpose hardware. Studies show that NFs constitute 40–60% of the appliances deployed in large-scale networks [36]. This architecture revolutionized the deployment of middleboxes for its lower cost, higher flexibility, and higher scalability. With the fast increasing data volume, the networking speed is also under rapid growth to meet the demand for fast data transfer. Therefore, achieving high performance is a critical requirement for NFV systems.

With the massive number of cores and high memory bandwidth, GPUs are well known for the capability of significantly accelerating NFs. Existing GPU-based NFs include router [15], SSL proxy [20], SRTP proxy [47], OpenFlow switch and IPsec gateway [15]. By forming CPU and GPU processing in a pipeline, the heterogeneous architecture is capable of delivering a high throughput in packet processing. Moreover, due to the rise of deep learning and other data analytical applications, GPUs are widely deployed in data centers and cloud services, e.g., Amazon EC2 GPU instance [2] and Alibaba Cloud GPU server [1]. Therefore, GPUs serve as a good candidate for building high-performance NFV systems. However, GPUs still have not been widely and effectively adopted in NFV systems. We identify the main reasons as threefold.

GPU Underutilization: Although state-of-the-art GPU virtualization techniques [39, 40, 45] enable multiple VMs to utilize a GPU, a GPU can only be accessed by a VM exclusively at a time, i.e., temporal sharing. Consequently, VMs have to access the GPU in a round-robin fashion. These virtualization approaches fit for GPU kernels that can fully utilize the GPU, such as deep learning [46] and database queries [41]. In production systems such as cloud, the input network traffic volume of an NF is generally much lower than the throughput that a GPU can achieve. As a result, the workload of each kernel in NFV systems is much lighter, which would result in severe GPU underutilization. Batching more tasks can be a feasible way to improve the GPU utilization, but it would result in a much higher latency. This issue largely blocks the adoption of GPUs in NFV systems as the overall throughput may be not enhanced or even degraded.

Lack of Support for Data Isolation: In a GPU-accelerated NFV system, both packets and the data structures of NFs need to be transferred to the GPU memory for GPU processing. When multiple NFs utilize a GPU to accelerate packet processing, they may suffer from information leakage due to the vulnerabilities in current GPU architectures [33]. As a result, a malicious NF may eavesdrop the packets in the GPU memory or even manipulate traffic of other NFs. As security is one of the main requirements in NFV systems [16], the lack of the system support for data isolation in the GPU memory may cause concern for NFV users.

Demanding Significant Development Efforts: Building an efficient network function on heterogeneous CPU-GPU architectures demands lots of development efforts. First, the complex data flow in network processing should be handled, including I/O operations, task batching, and data transferring in the CPU-GPU pipeline. Second, achieving the throughput and latency requirements needs to carefully adjust several parameters, such as the GPU batch size and the number of CPU and GPU threads. These parameters are highly relevant with the hardware and workloads, which makes it hard for NF deployment. All of the above efforts are repetitive and time-consuming in NF development and deployment.

In this paper, we propose an NFV system called *G-NET* to address the above issues and support efficient executions of NFs on GPUs. The main idea of G-NET is to spatially share a GPU among multiple NF instances to enhance the overall system efficiency. To achieve this goal, G-NET takes a holistic approach that encompasses all aspects of GPU processing, including the CPU-GPU processing pipeline, the GPU resource allocation, the GPU kernel scheduling, and the GPU virtualization. The proposed system not only achieves high efficiency but also lightens the programming efforts. The main contributions of this paper are as follows.

- A GPU-based NFV system, G-NET, that enables NFs to effectively utilize GPUs with spatial sharing.
- A GPU scheduling scheme that aims at maximizing the overall throughput of a service chain.
- A data isolation scheme to guarantee the data security in the GPU memory with both compile and runtime check.
- An abstraction for building NFs, which significantly reduces the development and deployment efforts.

Through experiments with a wide range of workloads, we demonstrate that G-NET is capable of enhancing the throughput of a service chain by up to 70.8% and reducing the latency by up to 44.3%, in comparison with the temporal GPU sharing virtualization approach.

The roadmap of this paper is as follows. Section 2 introduces the background of this research. Section 3 outlines the overall structure of G-NET. Section 4 describes the virtualization scheme and data plane designs of G-NET. Sections 5 and 6 describe the scheduling scheme and the abstraction for NF development. Section 7 evaluates the prototype system, Section 8 discusses related work, and Section 9 concludes the paper.

2 Background and Challenges

In this section, we review the background of adopting GPUs in NFV systems and discuss the major challenges in building a highly-efficient system.

2.1 Network Functions on Heterogeneous CPU-GPU Architectures

GPUs are efficient at network processing because the massive number of incoming packets offers sufficient parallelism. Since CPUs and GPUs have different architectural characteristics, they generally work in a pipelined fashion to execute specific tasks for high efficiency [15, 48]. CPUs are usually in charge of performing I/O operations, batching, and packet parsing. The compute/memory-intensive tasks are offloaded to GPUs for acceleration, such as cryptographic operations [20], deep packet inspection [19], and regular expression matching[42].

Take software router as an example, the data processing flow is as follows. First, the CPU receives packets from NICs, parses packet headers, extracts IP addresses, and batches them in an input buffer. When a specified batch size or a preset time limit is reached, the input buffer is transferred to the GPU memory via PCIe, then a GPU kernel is launched to lookup the IP addresses. After the kernel completes processing, the GPU results, i.e., the NIC ports to be forwarded to, are transferred back to the host memory. Based on the results, the CPU sends out the packets in the batch.

A recent CPU optimization approach G-Opt [21] achieves compatible performance with GPU-based implementations. G-Opt utilizes group prefetching and software pipelining to hide memory access latencies. Comparing with GPU-based implementations, such optimizations are time-consuming to apply, and they increase the difficulty in reading and maintaining the code. Moreover, the optimizations have limited impact on compute-intensive NFs [12], and the performance benefits may depend on the degree of cache contention when running concurrently with other processes.

In the following of this paper, we use NVIDIA CUDA terminology in the GPU related techniques, which are also applicable to OpenCL and GPUs from other vendors.

2.2 GPU Virtualization in NFV Systems

We implement four NFs on CPU-GPU architectures, including an L3 Router, a Firewall, a Network Intrusion Detection System (NIDS), and an IPsec gateway. The implementation follows the state-of-the-art network functions [19, 20, 15], where the GPU kernels are listed

NF	Kernel algorithm
Router	DIR-24-8-BASIC [13]
Firewall	Bit vector linear search [24] (188 rules)
NIDS	Aho-Corasick algorithm [4] (147 rules)
IPsec	AES-128 (CTR mode) and HMAC-SHA1

Table 1: GPU kernel algorithms of network functions.

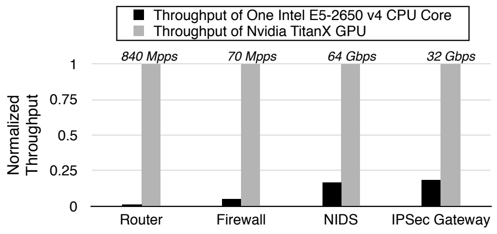

Figure 1: The throughputs of a CPU core and a GPU when forming a pipeline in GPU-accerated NFs (512-byte packet).

in Table 1. In the implementation of the Firewall, bit vector linear search is used to perform fast packet classification with multiple packet fields. For each field, we use trie [8] for IP address lookup, interval tree [7] for matching port range, and hash table for the protocol lookup. Based on the implementations, we conduct several experiments to evaluate the network functions and make the following observations. Please refer to Sect. 7.1 for the hardware and system configurations.

GPU is underutilized: In Figure 1, we show the throughput of the four GPU kernels, where a GPU demonstrates 840 Mpps for the L3 router, 70 Mpps for the Firewall, 64 Gbps for the NIDS, and 32 Gbps for the IPsec gateway. In cloud or enterprise networks, however, the traffic volume of an NF instance can be significantly lower than the GPU throughput. Consequently, such high performance can be overprovision for many production systems. Figure 1 also makes a comparison of the normalized throughput between a CPU core and a GPU when they form a pipeline in the NFs, where the CPU core performs packet I/O and batching, and the GPU performs the corresponding operations in Table 1. As shown in the figure, the throughput of a CPU core is significantly lower (5×-65×) than that of the GPU. As a result, being allocated with only limited number of CPU cores, an NF is unable to fully utilize a GPU. For the above reasons, a GPU can be severely underutilized in cloud or enterprise networks.

Temporal sharing leads to high latency: When only one NF runs on a server, the GPU is exclusively accessed by the NF. By overlapping the CPU processing and the GPU processing, the GPU timeline is shown in Figure 2(1). With the adoption of virtualization techniques [39, 40, 45] that enable temporal GPU sharing, NFs are able to access the GPU in a round-robin fashion. To maintain the same throughput with exclusive access, the GPU processing time of each NF should be reduced to allow multiple kernels utilizing the GPU. One may expect to utilize more GPU resources to reduce the GPU processing time. In Figure 3, we show that the GPU processing time quickly converges with first few Streaming Multiprocessors (SMs) allocated, and the reduction is moderate or even unnoticeable with more SMs. This is because that, with less than a certain number of jobs, an SM has a relatively fixed processing time in handling a specific task. As a result, although the batch size of an SM becomes smaller by assigning more SMs, the overall processing time cannot be further reduced. Consequently, temporal GPU sharing would not enhance the throughput of NFs, but the longer batching time would lead to a much higher latency. For instance, with another two kernels *B* and *C*, the GPU timeline would be like Figure 2(2), where the CPU batching time and the GPU processing time become significantly longer.

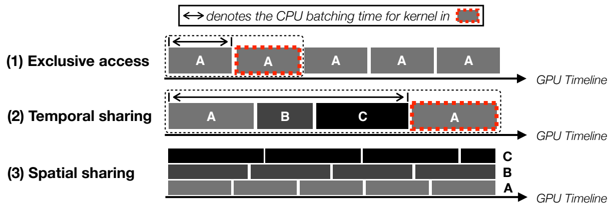

Figure 2: GPU execution with exclusive access, temporal sharing, and spatial sharing.

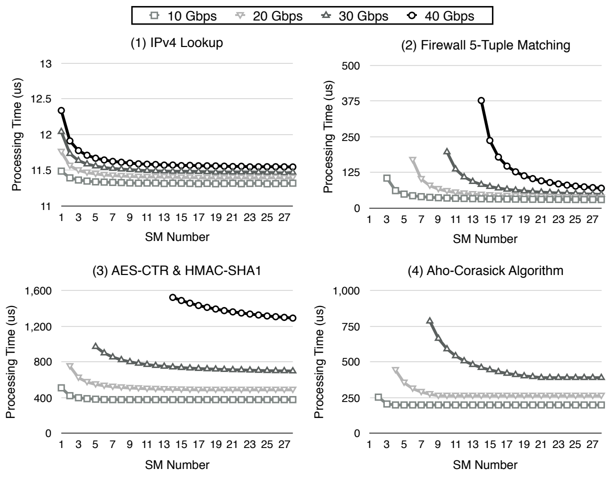

Figure 3: GPU processing time with different number of SMs (512-byte packet).

2.3 Opportunities and Challenges of Spatial GPU Sharing

With the lightweight kernels from multiple NFs, spatial GPU sharing is promising in enhancing the GPU efficiency. Spatial GPU sharing means multiple GPU kernels run on a GPU simultaneously (shown in Figure 2(3)), with each kernel occupying a portion of GPU cores. This technique has been proved to gain a significant performance improvement on simulated GPU architectures [3, 26]. A recently-introduced GPU hardware feature, *Hyper-Q*, exploits spatial GPU sharing. Adopting *Hyper-Q* in NFV systems faces several challenges.

Challenge 1: GPU virtualization. To run concurrently on a GPU with *Hyper-Q*, the kernels are required to have the same GPU context. Kernels of NFs in different VMs, however, are unable to utilize the feature for their different GPU contexts. Existing GPU virtualization approaches [9, 39, 40] are designed for the situation that the GPU is monopolized by one kernel at any instant, which do not adopt *Hyper-Q*. Utilizing *Hyper-Q* in NFV systems, therefore, demands a redesign of the GPU virtualization approach.

Challenge 2: Isolation. As NFs might come from different vendors, data isolation is essential for ensuring data security. With the same GPU context when utilizing *Hyper-Q*, all GPU memory regions are in the same virtual address space. As runtimes such as CUDA and OpenCL do not provide isolation among kernels from the same context, an NF is able to access memory regions allocated by other NFs. Consequently, utilizing the *Hyper-Q* hardware feature may lead to security issues.

Challenge 3: GPU scheduling. Virtualization makes an NF unaware of other coexisting NFs that share the same GPU, making them lack of the global view in resource usage. With spatial GPU sharing, if every NF tries to maximize its performance by using more GPU resources, the performance of the service chain can be significantly degraded due to resource contention. Existing GPU scheduling schemes [10, 22, 41] focus on sporadic GPU tasks and temporal GPU sharing, which mismatch the requirements and characteristics of NFV systems.

3 G-NET - An Overview

We propose G-NET, an NFV system that addresses the above challenges and effectively shares GPUs among NFs. In this section, we make an overview on the major techniques adopted in G-NET.

3.1 The G-NET Approach

To efficiently and safely share a GPU in an NFV environment, G-NET adopts the following major approaches.

Utilizing Hyper-Q in GPU virtualization: To enable spatial GPU sharing, G-NET utilizes the *Hyper-Q* feature in GPUs. We place a GPU management proxy in the hypervisor, which creates a common GPU context for all NFs. By utilizing the context to perform data transfer and

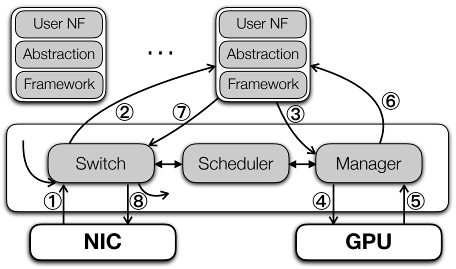

Figure 4: System architecture.

kernel operations, multiple kernels from different NFs can simultaneously run on the GPU.

isoPointer for data isolation: We implement *isoPointer*, a software memory access checking layer for GPU kernels. In G-NET, the GPU memory is accessed with *isoPointer*, which behaves like regular pointers but is able to check if the accessed memory address is legal, i.e., whether it belongs to the current kernel. *isoPointer* ensures the data isolation of NFs in the GPU memory.

Service chain based GPU scheduling: G-NET develops a service chain based GPU scheduling scheme that aims at maximizing the throughput of a service chain. Based on the workload of each NF, *Scheduler* calculates the corresponding GPU resources for each NF kernel to optimize the performance of the entire service chain. The scheduling algorithm is capable of dynamically adapting to workload changes at runtime.

Abstraction: We propose an abstraction for developing NFs. By generalizing the CPU-GPU pipelining, data transfer, and multithreading in a framework, NF developers only need to implement a few NF-specific functions. With the abstraction, the implementation efforts of an NF are significantly reduced.

3.2 The Architecture of G-NET

The architecture of G-NET is shown in Figure 4. There are three major functional units in the hypervisor layer of G-NET: *Switch*, *Manager*, and *Scheduler*. *Switch* is a virtual switch that performs packet I/O and forwards network packets among NFs. *Manager* is the proxy for GPU virtualization, which receives GPU execution requests from NFs and performs the corresponding GPU operations. *Scheduler* allocates GPU resources to optimize the overall performance of a service chain.

G-NET adopts a holistic approach in which the NF and the hypervisor work together to achieve spatial GPU sharing. A framework is proposed to handle the data flows and control flows in NF executions. Based on the programming interfaces of the framework, developers only need to implement NF-specific operations, which significantly reduces the development efforts. The processing data flow of an NF is shown in Figure 4. An NF receives packets from *Switch*, which can be from a NIC

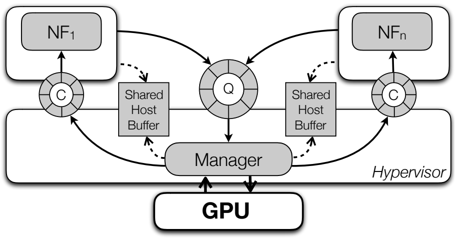

Figure 5: GPU virtualization in G-NET.

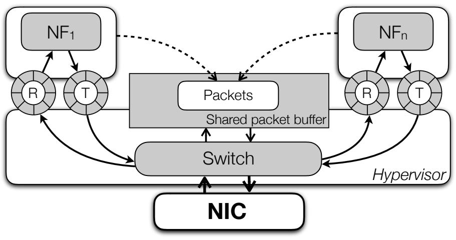

Figure 6: Data plane switching.

or other NFs. The NF batches jobs and utilizes *Manager* to perform GPU data transfer, kernel launch, and other operations. With the GPU processed results, the NF sends packets out through *Switch*.

4 System Design and Implementation

This section presents the main techniques adopted in G-NET, including the GPU virtualization approach, data plane switching, and GPU memory isolation.

4.1 GPU Virtualization and Sharing

As a common GPU context is demanded for spatial GPU sharing, GPU kernels launched in different VMs are unable to run simultaneously for their different contexts. To address this issue, G-NET creates a GPU context in the hypervisor and utilizes the context to execute GPU operations for all NFs. In order to achieve the goal, we virtualize at the GPU API level and adopt API remoting [14, 37] in GPU virtualization. API remoting is a virtualization scheme where device API calls in VMs are forwarded to the hypervisor to perform the corresponding operations. Figure 5 depicts our GPU virtualization approach. *Manager* maintains a response queue for each NF and a request queue that receives GPU requests from all NFs. To perform a GPU operation, an NF sends a message that includes the operation type and arguments to *Manager* via the request queue. *Manager* receives messages from NFs and performs the corresponding operations in the common GPU context asynchronously, so that it can serve other requests without waiting for their completion. Each NF is mapped to a CUDA stream by *Manager*, thus their operations run simultaneously in the same GPU context.

G-NET adjusts the number of thread blocks and the batch size of a kernel to achieve predictable performance (details in Sect. 5). In G-NET, the NF framework and the hypervisor work together to set these parameters. Upon receiving a kernel launch request, *Manager* uses the number of thread blocks that is calculated by *Scheduler* to launch the corresponding kernel. If the resource allocation scheme has been updated by *Scheduler*, *Manager* sends the batch size information to the NF via the response queue after the kernel completes execution. Then the framework in the NF uses the updated batch size to launch subsequent kernels.

There are several challenges for *Manager* to share the GPU. First, GPU kernels are located in VMs, which cannot be directly called by *Manager*. Second, the arguments should be set for launching a GPU kernel, but user-defined types cannot be known in the hypervisor. We utilize the CUDA driver API to address these issues. Each NF passes the source file and the name of its GPU kernel to *Manager* via a shared directory between the VM and the hypervisor. The hypervisor loads the kernel with CUDA driver APIs *cuModuleLoad* and *cuModuleGetFunction* and launches the kernel with *cuLaunchKernel*. Kernel arguments are stored in a shared host memory region by NFs, whose pointer can be directly passed to the kernel in *cuLaunchKernel*. In this way, *Manager* launches GPU kernels disregarding the specific argument details, which will be parsed automatically by the kernel itself.

For GPU operations such as kernel launch and PCIe data transfer, data is frequently transferred between a VM and the hypervisor. In G-NET, we develop a set of schemes to eliminate the overheads. When an NF requests to allocate a host memory region, *Manager* creates a shared memory region (shown in Figure 5) for each NF to transfer data by only passing pointers. For the allocation of the GPU memory, *Manager* directly passes the GPU memory pointer back to the NF, which would be passed back to perform PCIe data transfer or launch GPU kernels.

4.2 Data Plane Switching

Figure 6 shows the data plane switching in G-NET. Memory copy is known to have a huge overhead in high-speed network processing. To enhance the overall performance, we apply zero-copy principle in *Switch* to reduce the data transfer overhead of VM-to-NIC and VM-to-VM. Two communication channels are employed to move packets. One channel is a large shared memory region that stores packets. Packets are directly written into this memory region from NICs, allowing VMs to read and write packets directly. The other channel is two queues to pass packet pointers. Each VM has an input queue to receive packets and an output queue to send packets. As only packet pointers are transferred between VMs, the data transfer overhead is significantly alleviated.

There are two types of threads in *Switch*, which are called as *RX* thread and *TX* thread. *RX* threads receive packets from NICs. After analyzing packet headers, *RX* threads distribute packets to corresponding NFs through their input queues. An NF sends out packets by enqueueing packet pointers in its output queue. A *TX* thread in *Switch* is in charge of forwarding packets of one or several output queues. Based on the service chain information, a *TX* thread sends the dequeued packets out through NICs or to the following NFs for further processing.

4.3 Isolation

To address the data security issue in GPUs, we develop *isoPointer*, a mechanism to guarantee the data isolation in the GPU device memory. *isoPointer* acts as a software memory access checking layer that guarantees the read and write operations in the GPU memory do not surpass the bound of the legal memory space of an NF. An *isoPointer* is implemented as a C++ class, which overloads the pointer operators, including dereference, pre-increment, and post-increment. Each *isoPointer* instance is associated with the *base* address and the *size* of a memory region. At runtime, *dynamic checking* is enforced to ensure that the accessed memory address of an *isoPointer* is within its memory region, i.e., [*base, base+size*].

Despite dynamic checking, we ensure that all memory accesses in an NF are based on *isoPointer* with *static checking*. Static checking is performed on the source code of each NF, which needs to guarantee two aspects. First, an *isoPointer* can only be returned from system interfaces or passed as arguments, and no *isoPointer* is initiated or modified by users. Second, the types of all pointers in the GPU source code are *isoPointer*. This is performed with type checking, a mature technique in compiler. With both static and dynamic checking, G-NET guarantees data isolation in the GPU memory.

5 Resource Allocation and Scheduling

In this section, we introduce our GPU resource allocation and scheduling schemes. The main goal of the scheduling scheme is to maximize the throughput of a service chain while meeting the latency requirement.

5.1 Resource Allocation

Unlike CPUs, GPUs behave as black boxes, which provide no hardware support for allocating cores to applications. Threads are frequently switched on and off GPU cores when blocked by operations such as memory access or synchronization. As a result, it is unable to precisely allocate a GPU core to a GPU thread. Moreover, whether kernels can take advantage of *Hyper-Q* to spatially share the GPU depends on the availability of the GPU resources, including the registers and the shared memory. When the resources to be taken by a kernel exceed current available resources, the kernel would be queued up to wait for the resources becoming available. Consequently, an improper GPU resource allocation scheme is detrimental to the overall performance.

G-NET uses SM as the basic unit in the allocation of GPU computational resources. A thread block is a set of threads that can only run on one SM. Modern GPUs balance the number of thread blocks on SMs, where two thread blocks are not supposed to be scheduled to run on the same SM when there still exists available ones. We utilize this feature and allow the same or a smaller number of thread blocks as that of SMs to run on a GPU simultaneously, so that each thread block executes exclusively on one SM. By specifying the number of thread blocks of a GPU kernel, an NF is allocated with an exact number of SMs, and multiple GPU kernels can co-run simultaneously.

We propose a GPU resource allocation scheme that uses two parameters to achieve predictable performance: the batch size and the number of SMs. The scheme is based on a critical observation: there is only marginal performance impact ($< 7\%$ in our experiments) from thread blocks running on different SMs. When utilizing more SMs with each SM being assigned with the same load, the overall kernel execution time (w/o PCIe data transfer) stays relatively stable. The main reasons for this phenomenon are twofold. First, the memory bandwidth of current GPUs is high (480 GB/s on NVIDIA Titan X Pascal), which is sufficient for co-running several NFs on 10 Gbps network. Second, SMs do not need to compete for other resources such as register file or cache, as each SM has its independent resources. Therefore, the batch size of an SM controls its throughput and processing time, while allocating more SMs can reap a near-linear throughput improvement.

5.2 Performance Modeling

To achieve predictable performance by controlling the batch size of an SM, we model the relationship between the performance of an SM and the batch size with our evaluation results. Figure 7 shows the GPU kernel execution time on one SM, the PCIe data transfer time, and the corresponding throughputs of four NFs.

As shown in the figure, the throughput of GPU kernels have different patterns, where the throughputs of

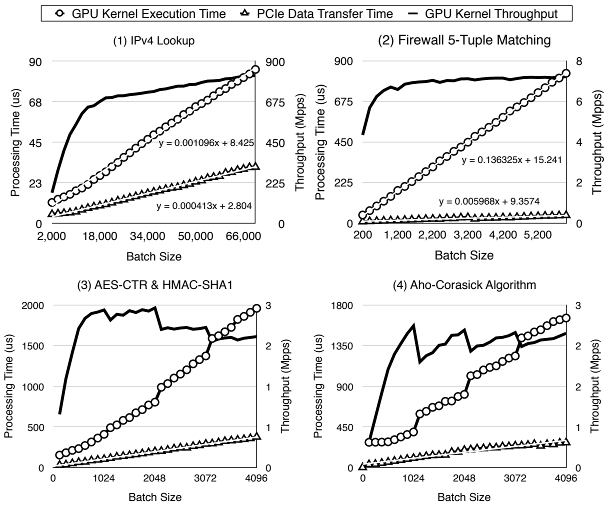

Figure 7: GPU processing time and throughput on one SM (Packet size: 512-byte; Thread block size: 1024).

NIDS and IPsec gateway increase and then drop in steps with the increase of batch size. As the kernels reach the highest throughputs before the first performance degradation, we use the batch size in the first step to adjust the SM throughput. The kernel execution time and the PCIe data transfer time form a near-linear relationship with the batch size (within the first step). Therefore, we adopt linear regression in the model. We use $L_k = k_1 \cdot B + b_1$ to describe the relationship between the batch size (B) and the GPU kernel execution time (L_k), and $L_m = k_2 \cdot B + b_2$ is used to describe the relationship between the batch size (B) and the PCIe data transfer time (L_m). As the kernel execution time stays stable with different number of SMs when the batch size assigned to each SM stays the same, we can get the overall GPU execution time of a kernel as

$$L = L_k + L_m = k_1 \cdot B_0 + b_1 + k_2 \cdot B_0 \cdot N + b_2 \quad (1)$$

, where N denotes the number of SMs, and B_0 denotes the batch size of each SM. The throughput can be derived as

$$T = N \cdot B_0 / L \quad (2)$$

The parameters of the linear equations are relative with hardware platform, number of rules installed (e.g., NIDS and Firewall), and packet size. In system deployment, we develop a tool called *nBench* to measure the PCIe data transfer time and the kernel execution time to derive the corresponding linear equations with locally generated packets. With *nBench*, our system can be quickly deployed on servers with different hardware and software configurations by only profiling each NF once.

We have implemented several GPU-based NFs, and we find that NFs can be classified into two main groups. 1) For NFs that inspect packet payloads, the performance has a similar pattern with NIDS and IPsec gateway. The length of the performance fluctuation step equals to the thread block size of the GPU kernel (1024 in our evaluation). 2) For NFs that only inspect packet headers, the performance exhibits the same pattern with Router and Firewall. Therefore, this simple but effective performance model can be applied to other NFs. Moreover, in the G-NET implementation, our scheduling scheme considers the potential model deviations (denote as C in Sect. 5.3) in the resource allocation, making our approach more adaptive in practice.

5.3 Service Chain Based GPU Scheduling

The GPU scheduling in NFV systems has the following specific aspects. (1) The packet arrival rate and the packet processing cost of each NF are dramatically different. If each NF is allocated with the same amount of resources, the NF with the heaviest load would degrade the overall throughput [23]. (2) The workload of an NFV system can be dynamically changing over time. For instance, under malicious attack, the throughput of NIDS should be immediately enhanced.

Based on the performance model, we propose a scheduling scheme that aims at maximizing the throughput of an entire service chain while meeting the latency requirements of NFs. Different with the modeling environment, the scheduling scheme needs to consider the costs brought by the implementation and hardware. First, there is overhead in the communication between NFs and *Manager* in performing GPU operations. Second, as there are only one host-to-device (HtoD) and one device-to-host (DtoH) DMA engine in current GPUs, the data transfer of an NF has to be postponed if the required DMA engine is occupied by other NFs. Third, the model may have deviations. Our scheduling scheme takes these overheads into consideration (denote by C), which works as follows.

At runtime, *Scheduler* monitors the throughput of each NF and progressively allocates GPU resources. We first find the NF that achieves the lowest throughput (denote by T') in the service chain, then allocate all NFs with enough GPU resources to meet the throughput $T = T' \cdot (1+P)$, where $P \in (0,1)$. If there are branches in the service chain, we first allocate resources for NFs in each branch. Then the sum of the throughputs of the child branches is used as the throughput for their father branch in resource allocation. This procedure repeats until GPU resources are exhausted, which improves the overall throughput by P in each round.

In each round, *Scheduler* calculates the minimum number of SMs and the batch size to meet the latency and throughput requirements of each NF. Starting from assigning one SM, the scheme checks if the current number

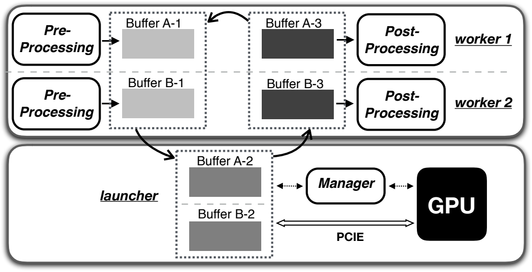

Figure 8: The framework of a network function.

of SMs (denoted as N) is able to meet the schedulability condition of an NF, i.e., achieving a latency lower than the user-specified latency requirement L and a throughput higher than the input speed T. According to Eq. 2, by substituting $N \cdot B_0/T$ for L into Eq. 1, the minimum batch size demanded to achieve the throughput T is derived as

$$B_0 = \lceil T \cdot (b_1 + b_2 + C)/(N - T \cdot (k_1 + k_2 \cdot N)) \rceil \quad (3)$$

Then, the overall processing time can be calculated as

$$L_0 = k_1 \cdot B_0 + b_1 + k_2 \cdot B_0 \cdot N + b_2 + C \quad (4)$$

If $L_0 \leq L$, it means the NF can meet both its latency and throughput requirements with N SMs and a batch size of $B_0 \cdot N$. If $L_0 > L$, one more SM is allocated and the above procedure is performed again. The procedure repeats until no available SMs left in the GPU.

In G-NET, the scheduling scheme runs when the traffic of an NF changes for more than 10%.

6 Abstraction and Framework for NFs

The implementation of NFs on CPU-GPU architectures should be efficient and easy to scale up. In G-NET, we propose an abstraction for NFs to reduce the development efforts.

6.1 Framework

We generalize the CPU-GPU pipeline of network functions as three main stages, namely *Pre-processing*, *GPU processing*, and *Post-processing*. Figure 8 shows our framework. The CPU is in charge of batching and packet I/O in *Pre-processing* and *Post-processing* stages, and the GPU is in charge of compute/memory-intensive operations in the *GPU processing* stage. Our main design choices are as follows.

Thread model in CPU-GPU pipelining: There are two types of CPU threads in the pipeline. *Worker* threads perform packet processing in *Pre-processing* and *Post-processing* stages. To communicate with *Manager* for GPU execution, a specific thread called *launcher* is implemented to manage GPU operations including data transfer and kernel execution. This avoids *worker* threads waiting for the GPU operations and makes them focus on processing continuously coming packets.

Buffer transfer among pipeline stages: We develop a buffer transfer mechanism to prevent *workers* from being stalled when transferring data to the GPU. There are three buffers in each pipeline. The *launcher* thread automatically switches the buffer of the *Pre-processing* stage when the *Scheduler*-specified batch size is reached. The following two stages pass their buffers to the next stages circularly after they complete their tasks.

Scale up: We scale up a network function when the CPU becomes its bottleneck, i.e., launching more *worker* threads to enhance the data processing capability. When executing a GPU operation, the *launcher* passes the thread id to *Manager*, which is mapped with a CUDA stream for independent execution. With an independent input and output queue for each *worker*, the design simplifies NF management and enhances throughput.

6.2 Abstraction

Based on our framework, we propose an abstraction to mitigate the NF development efforts. The abstraction mainly consists of five basic operations, i.e., *pre_pkt_handler*, *mem_htod*, *set_args*, *mem_dtoh*, *post_pkt_handler*. Called by the framework in the *Pre-processing* stage, *pre_pkt_handler* performs operations including packet parsing and batching. The framework manages the number of jobs in the batch, and developers only need to batch a job in the position of *batch->job_num* in the buffer. Before the framework launches the GPU kernel for a batch, *mem_htod* and *set_args* are called to transfer data from the host memory to the GPU memory and set arguments for the GPU kernel. Please note that the order of the arguments should be consistent with the GPU kernel function in *set_args*. Then the framework sends requests to *Manager* to launch the specified GPU kernel. After kernel completes execution, *mem_dtoh* is called to transfer data from the GPU memory to the host memory. *post_pkt_handler* is called for every packet after GPU processing.

As an example, Figure 9 demonstrates the major parts of a router implemented with our abstraction. First, the developer defines the specific batch structure (lines 1-7). In a router, it includes the number of jobs in a batch and the input and output buffers in the host and the GPU memory. Each *worker* thread is allocated with a batch structure for independent processing. With the *kernel_init* function (lines 8-11), developers install its kernel by specifying its *.cu* kernel file and the kernel function name ("*iplookup*"). Developers can also perform other initialization operations in *kernel_init*, such as building

```
1   struct my_batch {
2       uint64_t job_num;
3       isoPtr<uint32_t> host_ip;
4       isoPtr<uint32_t> dev_ip;
5       isoPtr<uint8_t> host_port;
6       isoPtr<uint8_t> dev_port;
7   }
8   void kernel_init(void) {
9       gInstallKernel("/pathto/router.cu", "iplookup");
10      build_routing_table();
11  }
12  void pre_pkt_handler(batch, pkt) {
13      batch->host_ip[batch->job_num] = dest_ip(pkt);
14  }
15  void memcpy_htod(batch) {
16      gMemcpyHtoD(batch->dev_ip, batch->host_ip,
17          batch->job_num * IP_SIZE);
18  }
19  void set_args(batch) {
20      gInstallArgNum(4);
21      gInstallArg(batch->dev_ip);
22      gInstallArg(batch->dev_port);
23      gInstallArg(batch->job_num);
24      gInstallArg(dev_lookup_table);
25  }
26  void memcpy_dtoh(batch) {
27      gMemcpyDtoH(batch->host_port, batch->dev_port,
28          batch->job_num * PORT_SIZE);
29  }
30  void post_pkt_handler(batch, pkt, pkt_idx) {
31      pkt->port = batch->host_port[pkt_idx];
32  }
```

Figure 9: The major parts of a router implemented with G-NET APIs.

the routing table. The *pre_pkt_handler* (lines 12-14) of the router extracts the destination IP address of a packet and batches it in the host buffer. Functions *mem_htod* and *mem_dtoh* transfer the ip address buffer into the GPU memory and the port buffer into the host memory, respectively. *post_pkt_handler* (lines 30-32) records the port number which will be used by the framework to send the packet out. With the abstraction, developers only need to focus on the implementation of specific tasks of an NF, reducing thousands of lines of development efforts.

Based on our abstraction, NFs can be developed by different vendors, where the GPU kernel source code should be provided so that the kernels can be loaded by the *Manager*. If vendors that do not want to leak their source code, they may have to deploy a whole package of the G-NET system, including the NFs and the functionalities in the hypervisor. A secure channel with cryptographic operations can be built between NFs and the *Manager* to pass the source code.

7 Experiment

7.1 Experimental Methodology

Experiment Platform: We conduct experiments on a PC equipped with an Intel Xeon E5-2650 v4 processor running at 2.2 GHz. The processor contains 12 physical cores with hyper-threading enabled. The processor has

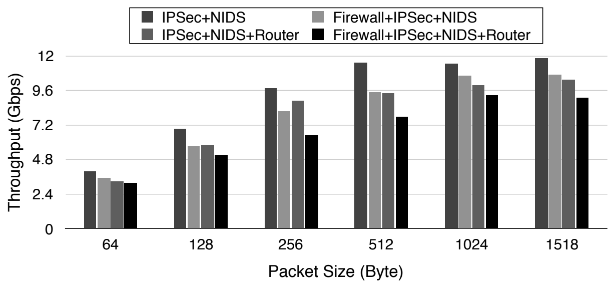

Figure 10: System throughput of G-NET.

an integrated memory controller installed with 4×16 GB 2400 MHz DDR4 memory. An NVIDIA Titan X Pascal GPU is deployed, which has 28 streaming multiprocessors and a total of 3584 cores. An Intel XL710 dual port 40 GbE NIC is used for network I/O, and DPDK 17.02.1 is used as the driver. The operating system is 64-bit CentOS 7.3.1611 with Linux kernel 3.8.0-30. Docker 17.03.0-ce is used as our virtualization platform. Each NF runs in a Docker instance, while *Manager*, *Switch*, and *Scheduler* run on the host.

Service Chains: We implement four network functions on G-NET, i.e., Router, Firewall, NIDS, IPsec gateway, as listed in Table 1. Composed by the NFs, four service chains are used to evaluate the performance of G-NET: ($\mathbb{S}_a$) IPsec + NIDS; ($\mathbb{S}_b$) Firewall + IPsec + NIDS; ($\mathbb{S}_c$) IPsec + NIDS + Router; ($\mathbb{S}_d$) Firewall + IPsec + NIDS + Router.

7.2 System Throughput

Figure 10 shows the throughput of G-NET for the four service chains. We set one millisecond as the latency requirement for the GPU execution of each NF, as we aim at evaluating the maximum throughput of G-NET. With the service chain $\mathbb{S}_a$ that has two NFs, the system throughput reaches up to 11.8 Gbps with the maximum ethernet frame size 1518-byte. For the service chain $\mathbb{S}_d$ with four NFs, the system achieves a throughput of 9.1 Gbps.

As depicted in the figure, the system throughput increases with the size of packet. When the packet size is small, the input data volume of service chains is limited by the nontrivial per-packet processing overhead, including switching, batching, and packet header parsing. The main overhead comes from packet switching. Switching a packet between two NFs includes at least two enqueue and dequeue operations, and the packet header should be inspected to determine its destination, which is known to have severe performance issues [17, 27, 32, 34]. In an NFV system with a service chain of multiple NFs, the problem gets more pronounced as a packet needs to be forwarded multiple times in the service chain. More-

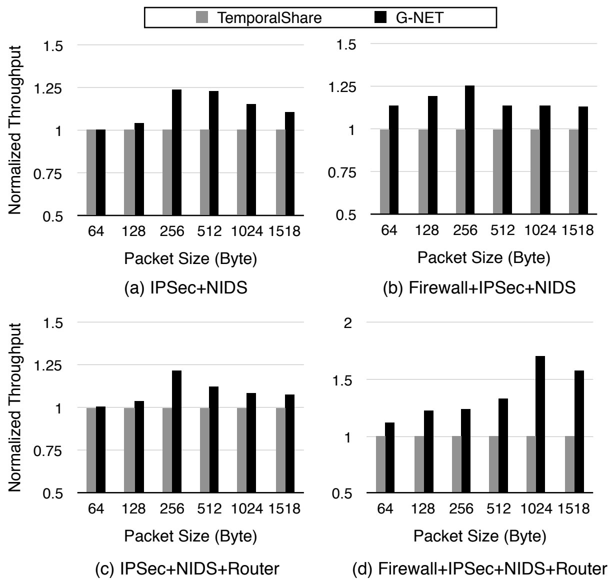

Figure 11: Throughput comparison between G-NET and *TemporalShare*. Performance is normalized to *TemporalShare*.

over, as the NFs in the system demand CPU cores for packet processing, leaving *Switch* limited resources for packet forwarding. Assigning the *Switch* with more threads, however, would deprive the cores allocated for the *worker* threads in NFs, resulting in overall performance degradation.

7.3 Performance Improvement From Spatial GPU Sharing

To evaluate the performance improvement with spatial GPU sharing, we implement a *TemporalShare* mode in G-NET for comparison. The *TemporalShare* mode represents existing GPU virtualization approaches, where kernels from NFs access the GPU serially instead of being executed simultaneously.

Figure 11 compares the throughput of G-NET and *TemporalShare*. As shown in the figure, the spatial GPU sharing in G-NET significantly enhances the overall throughput. For the four service chains, the throughput improvements reach up to 23.8%, 25.9%, 21.5%, and 70.8%, respectively. The throughput improvements for the service chain with four NFs are higher than that of service chains with less NFs. For a small number of NFs, a fraction of the GPU kernels and PCIe data transfer could overlap with *TemporalShare*, leading to less resource contention. When there are more NFs co-running in the system, they can reap more benefits of spatial GPU sharing for the fierce resource competition.

There are two main aspects that limit the performance improvement by spatial GPU sharing. First, although

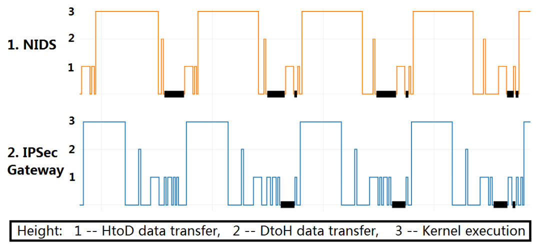

Figure 12: Data transfer conflicts in G-NET.

GPU kernels are able to utilize Hyper-Q to co-run on a GPU, their PCIe data transfer have to be sequentially performed. This is due to the limited number of DMA engines. Figure 12 plots the trace of G-NET with IPsec gateway and NIDS, which demonstrates the concurrent kernel executions with spatial GPU sharing and the DMA data transfer conflicts. The delays in NFs caused by the data transfer conflicts are marked as bold black lines in the figure. As shown in the figure, NFs spend a significant amount of time in waiting for the HtoD DMA engine when it is occupied by other NFs. The system performance could be further unleashed when hardware vendors equip GPUs with more DMA engines for parallel data transfer. Second, the bottleneck of G-NET on current evaluation platform lies in the CPU. Our GPU provides abundant computational resources, which is unable to be matched by the CPU. For instance, with four NFs, each NF can only be assigned with two physical cores, resulting in low packet processing capability. For workloads with large packets, the batching operations that perform *memcpy* on packet payloads limit the overall performance. Instead, the switching overhead becomes the main factor that affects the overall performance for workloads with small packets. It is our future work to investigate how to further reduce this overhead.

7.4 Evaluation of Scheduling Schemes

To evaluate the effectiveness of the GPU scheduling in G-NET, we use two other scheduling schemes for comparison, i.e., *FairShare* and *Uncohare*. Different with the scheduling scheme of G-NET, the SMs are evenly partitioned among all NFs in the *FairShare* mode. In the *UncoShare* mode, *Scheduler* is disabled, and each NF tries to use as many GPU resources as possible.

The throughput improvements of G-NET over *FairShare* and *UncoShare* are shown in Figure 13 and Figure 14, respectively. For the four service chains, the average throughput improvements of G-NET are 16.7-34.0% over *FairShare* and 50.8-130.1% over *UncoShare*. The throughput improvements over *UncoShare* are higher than that of *FairShare* for the following reasons. In the *FairShare* mode, although the GPU resources are parti-

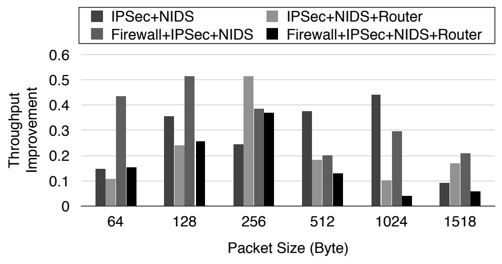

Figure 13: Throughput improvements in G-NET over *FairShare* GPU scheduling.

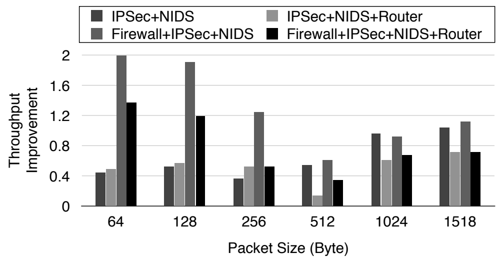

Figure 14: Throughput improvements in G-NET over *UncoShare* GPU scheduling.

tioned among the NFs, they are still able to co-run their GPU kernels simultaneously. In the *UncoShare* mode, however, each NF tries to use as many resources (SMs) as possible. This may exhaust the GPU resources, making part of the GPU kernels cannot run together. Moreover, the performance of a kernel can be degraded due to the interference of kernels running on the same SMs. To conclude, our scheduling scheme demonstrates very high efficiency in enhancing the performance of the service chains.

7.5 The Overhead of IsoPointer

To guarantee the isolation of different NFs, G-NET uses *IsoPointer* to check, validate, and restrict GPU memory accesses. These management activities may add some runtime overhead to NF executions.

We measure the overhead by comparing the performance of G-NET with and without *IsoPointer* under two groups of workloads. The first group consists of the solo executions of the NFs, and the second group comprises service chains with 2-4 NFs. The results are presented in Figure 15, where *SC-k* denotes a service chain with k NFs. We report the performance of service chain $\mathbb{S}_b$ for *SC-3*, as the measured overheads of $\mathbb{S}_b$ and $\mathbb{S}_c$ are similar. As shown in the figure, the overhead of *IsoPointer* ranges from 2.8% to 4.5%, which is negligibly low.

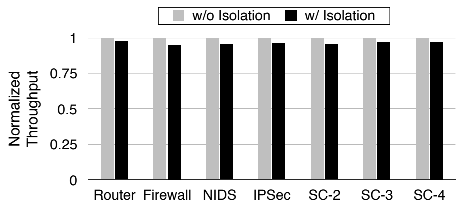

Figure 15: Overhead of IsoPointer.

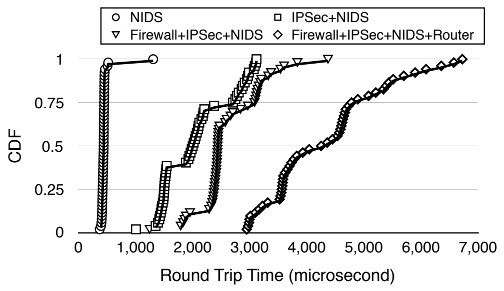

Figure 16: System latency of four service chains.

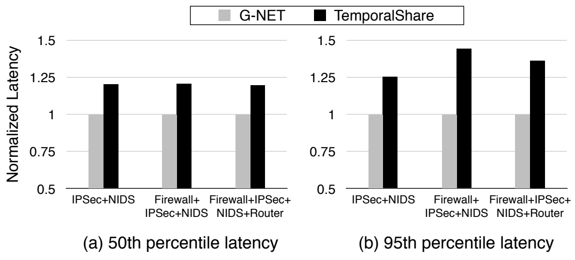

Figure 17: Latency comparison with *TemporalShare*.

7.6 Latency

We evaluate system processing latency by measuring the time elapsed from sending a packet to the NFV system to receiving the processed packet. The client keeps sending packets with source IP address increasing by one for each packet. By sample logging the sending/receiving time and the IP address, the round trip latency can be calculated as the time elapsed between the queries and responses with matched IP addresses.

Figure 16 shows the Cumulative Distribution Function (CDF) of the packet round trip latency with four service chains. The latency is measured by setting the maximum GPU execution time of each NF as one millisecond, as it demonstrates the system latency with the maximum throughput. As shown in the figure, the latency of one NF is low and stable. With two to four NFs, the service chains show piecewise CDFs, where latencies are clustered into three or more areas. The main reason for this phenomenon is the PCIe data transfer conflict. As there are only one HtoD and one DtoH DMA engine in current GPUs, a kernel would be postponed for execution if other NFs are utilizing the engine, leading to higher processing latency. If data is transferred while other NFs are running GPU kernels, they will overlap with no performance loss. Therefore, different degrees of resource competition in NF executions lead to the several clusters of latency distribution in the CDF. The latency in G-NET mainly comes from three parts, i.e., GPU processing, batching, and packet switching. With a lower input network speed, the latency would decrease for the lower batching time and GPU processing time.

Figure 17 compares the latency of G-NET with *TemporalShare*. With temporal GPU sharing, the 50th percentile latency is around 20% higher than that of G-NET, and the 95th percentile latency is 25.5%-44.3% higher than that of G-NET. It is worth noting that the throughput of *TemporalShare* is lower than G-NET (shown in Figure 11). Without spatial GPU sharing, both the kernel execution and the PCIe data transfer are serialized, resulting in low throughputs and high latencies.

8 Related Work

GPU Accelerated Network Functions: There are a class of work that utilize GPUs in accelerating network functions, including router [15], SSL reverse proxy [20], NIDS [19], and NDN system [42]. APUnet [12] studies NF performance on several architectures, which demonstrates the capability of GPUs in efficient packet processing and identifies the overhead in PCIe data transfer.

Network Function Virtualization: Recent NFV systems that are built with high performance data plane include NetVM [18] and ClickNP [25]. NetVM is a CPU-based NFV framework that implements zero-copy data transfer among VMs, which inspires the design of *Switch* in G-NET. NFVnice [23] is an NF scheduling framework that maximizes the performance of a service chain by allocating CPU time to NFs based on their packet arrival rate and the required computational cost. The goal of the CPU resource allocation in NFVnice is the same with that of the GPU resource allocation in G-NET. ClickNP and Emu [28] accelerate the data plane of network functions with FPGA. Same with G-NET, they also provide high-level languages to mitigate the development efforts. NFP [38] is a CPU-based NFV framework that exploits the opportunity of NF parallel execution to improve NFV performance. On the control plane, there is a class of work [6, 11, 29] focus on flexible and easy NF deployment including automatic NF placement, dynamic scaling, and NF migration.

GPU Virtualization: GPU virtualization approaches are generally classified into three major classes, namely, I/O pass-through [2], API remoting [14, 37], and hybrid [9]. Recent work includes alleviating the overhead of GPU virtualization [39, 40] and resolving the memory capacity limitation [45]. These systems do not explore the spatial GPU sharing, as most of them assume a kernel would saturate GPU resources.

GPU Sharing: Recent work on GPU multitasking studies spatial GPU sharing [3] and simultaneous multikernel [43, 44] with simulation. [31] shows that spatial GPU sharing has better performance when there is resource contention among kernels. Instead, simultaneous multikernel, which allows kernels to run on the same SM, has advantage for kernels with complimentary resource demands [31]. NF kernels have similar operations in packet processing, making spatial sharing a better choice.

Isolation: For the implementation of *IsoPointer*, G-NET adopts a similar technique with ActivePointer [35], which intercepts GPU memory accesses for address translation. Paradice [5] proposes a data isolation scheme in paravirtualization, which guarantees that the CPU code of a VM can only access its own host and device memory. Different with G-NET, Paradice is unable to prohibit malicious GPU code from accessing illegal GPU memory regions. Instead of virtualization, NetBricks [30] utilizes type checking and safe runtimes to provide data isolation for CPU-based NFs. This approach discards the flexibility brought by virtualization, such as NF migration and the ability to run on different software/hardware platforms. Moreover, we find the performance penalties caused by virtualization is negligibly low in G-NET (only around 4%).

9 Conclusion

We propose G-NET, an NFV system that exploits spatial GPU sharing. With a service chain based GPU scheduling scheme to optimize the overall throughput, a data isolation scheme to guarantee data security in the GPU memory, and an abstraction to significantly reduce development efforts, G-NET enables effective and efficient adoption of GPUs in NFV systems. Through extensive experiments, G-NET significantly enhances the throughput by up to 70.8% and reduces latency by up to 44.3% for GPU-based NFV systems.

10 Acknowledgement

We would like to thank our shepherd KyoungSoo Park and anonymous reviewers of NSDI'18 for their insightful comments and suggestions. We also acknowledge the support of NVIDIA Corporation with the donation of the Titan X Pascal GPU used for this research. This work was supported in part by a MoE AcRF Tier 2 grant (MOE2017-T2-1-122) in Singapore, NSFC (Grant No. 61732004, 61370080) and the Shanghai Innovation Action Project (Grant No. 16DZ1100200) in China.

References

[1] Alihpc. "`https://hpc.aliyun.com/product/gpu_bare_metal/`".

[2] Amazon high performance computing cloud using GPU. "`http://aws.amazon.com/hpc/`".

[3] ADRIAENS, J. T., COMPTON, K., KIM, N. S., AND SCHULTE, M. J. The Case for GPGPU Spatial Multitasking. In *HPCA* (2012), pp. 1–12.

[4] AHO, A. V., AND CORASICK, M. J. Efficient String Matching: An Aid to Bibliographic Search. *Communications of the ACM 18*, 6 (1975), 333–340.

[5] AMIRI SANI, A., BOOS, K., QIN, S., AND ZHONG, L. I/O Paravirtualization at the Device File Boundary. In *ASPLOS* (2014), pp. 319–332.

[6] BREMLER-BARR, A., HARCHOL, Y., AND HAY, D. OpenBox: A Software-Defined Framework for Developing, Deploying, and Managing Network Functions. In *SIGCOMM* (2016), pp. 511–524.

[7] CORMEN, T. H., STEIN, C., RIVEST, R. L., AND LEISERSON, C. E. *Introduction to Algorithms*, 3rd ed. 2009.

[8] DE LA BRIANDAIS, R. File Searching Using Variable Length Keys. In *Western Joint Computer Conference* (1959), pp. 295–298.

[9] DOWTY, M., AND SUGERMAN, J. GPU Virtualization on VMware's Hosted I/O Architecture. In *SIGOPS Oper. Syst. Rev.* (2009), pp. 73–82.

[10] ELLIOTT, G. A., AND ANDERSON, J. H. Globally Scheduled Real-time Multiprocessor Systems with GPUs. *Real-Time Syst.* (2012), 34–74.

[11] GEMBER-JACOBSON, A., VISWANATHAN, R., PRAKASH, C., GRANDL, R., KHALID, J., DAS, S., AND AKELLA, A. OpenNF: Enabling Innovation in Network Function Control. In *SIGCOMM* (2014), pp. 163–174.

[12] GO, Y., JAMSHED, M. A., MOON, Y., HWANG, C., AND PARK, K. APUNet: Revitalizing GPU as Packet Processing Accelerator. In *NSDI* (2017), pp. 83–96.

[13] GUPTA, P., LIN, S., AND MCKEOWN, N. Routing lookups in hardware at memory access speeds. In *INFOCOM* (1998), pp. 1240–1247.

[14] GUPTA, V., SCHWAN, K., TOLIA, N., TALWAR, V., AND RANGANATHAN, P. Pegasus: Coordinated Scheduling for Virtualized Accelerator-based Systems. In *USENIX ATC* (2011).

[15] HAN, S., JANG, K., PARK, K., AND MOON, S. PacketShader: A GPU-Accelerated Software Router. In *SIGCOMM* (2010).

[16] HAWILO, H., SHAMI, A., MIRAHMADI, M., AND ASAL, R. NFV: state of the art, challenges, and implementation in next generation mobile networks (vEPC). *IEEE Network 28*, 6 (2014), 18–26.

[17] HONDA, M., HUICI, F., LETTIERI, G., AND RIZZO, L. mSwitch: A Highly-scalable, Modular Software Switch. In *SOSR* (2015), pp. 1:1–1:13.

[18] HWANG, J., RAMAKRISHNAN, K. K., AND WOOD, T. NetVM: High Performance and Flexible Networking Using Virtualization on Commodity Platforms. In *NSDI* (2014), pp. 445–458.

[19] JAMSHED, M. A., LEE, J., MOON, S., YUN, I., KIM, D., LEE, S., YI, Y., AND PARK, K. Kargus: A Highly-scalable Software-based Intrusion Detection System. In *CCS* (2012), pp. 317–328.

[20] JANG, K., HAN, S., HAN, S., MOON, S., AND PARK, K. SSLshader: Cheap SSL Acceleration with Commodity Processors. In *NSDI* (2011).

[21] KALIA, A., ZHOU, D., KAMINSKY, M., AND ANDERSEN, D. G. Raising the Bar for Using GPUs in Software Packet Processing. In *NSDI* (2015), pp. 409–423.

[22] KATO, S., LAKSHMANAN, K., RAJKUMAR, R., AND ISHIKAWA, Y. TimeGraph: GPU Scheduling for Real-time Multi-tasking Environments. In *USENIX ATC* (2011).

[23] KULKARNI, S. G., ZHANG, W., HWANG, J., RAJAGOPALAN, S., RAMAKRISHNAN, K. K., WOOD, T., ARUMAITHURAI, M., AND FU, X. NFVnice: Dynamic Backpressure and Scheduling for NFV Service Chains. In *SIGCOMM* (2017), pp. 71–84.

[24] LAKSHMAN, T. V., AND STILIADIS, D. High-speed Policy-based Packet Forwarding Using Efficient Multi-dimensional Range Matching. In *SIGCOMM* (1998), pp. 203–214.

[25] LI, B., TAN, K., LUO, L. L., PENG, Y., LUO, R., XU, N., XIONG, Y., CHENG, P., AND CHEN, E. ClickNP: Highly Flexible and High Performance Network Processing with Reconfigurable Hardware. In *SIGCOMM* (2016), pp. 1–14.

[26] LIANG, Y., HUYNH, H. P., RUPNOW, K., GOH, R. S. M., AND CHEN, D. Efficient GPU Spatial-Temporal Multitasking. *IEEE Transactions on Parallel and Distributed Systems 26* (2015), 748–760.

[27] MOLNÁR, L., PONGRÁCZ, G., ENYEDI, G., KIS, Z. L., CSIKOR, L., JUHÁSZ, F., KŐRÖSI, A., AND RÉTVÁRI, G. Dataplane Specialization for High-performance OpenFlow Software Switching. In *SIGCOMM* (2016), pp. 43–56.

[28] N, S., S, G., D, G., M, W., J, S., R, C., L, M., P, B., R, S., R, M., P, C., P, P., J, C., AW, M., AND N, Z. Emu: Rapid Prototyping of Networking Services. In *USENIX ATC* (2017), pp. 459–471.

[29] PALKAR, S., LAN, C., HAN, S., JANG, K., PANDA, A., RATNASAMY, S., RIZZO, L., AND SHENKER, S. E2: A Framework for NFV Applications. In *SOSP* (2015), pp. 121–136.

[30] PANDA, A., HAN, S., JANG, K., WALLS, M., RATNASAMY, S., AND SHENKER, S. NetBricks: Taking the V out of NFV. In *OSDI* (2016), pp. 203–216.

[31] PARK, J. J. K., PARK, Y., AND MAHLKE, S. Dynamic Resource Management for Efficient Utilization of Multitasking GPUs. In *ASPLOS* (2017), pp. 527–540.

[32] PFAFF, B., PETTIT, J., KOPONEN, T., JACKSON, E., ZHOU, A., RAJAHALME, J., GROSS, J., WANG, A., STRINGER, J., SHELAR, P., AMIDON, K., AND CASADO, M. The Design and Implementation of Open vSwitch. In *NSDI* (2015), pp. 117–130.

[33] PIETRO, R. D., LOMBARDI, F., AND VILLANI, A. CUDA Leaks: A Detailed Hack for CUDA and a (Partial) Fix. *ACM Trans. Embed. Comput. Syst. 15* (2016), 15:1–15:25.

[34] RIZZO, L., AND LETTIERI, G. VALE, a Switched Ethernet for Virtual Machines. In *CoNEXT* (2012), pp. 61–72.

[35] SHAHAR, S., BERGMAN, S., AND SILBERSTEIN, M. ActivePointers: A Case for Software Address Translation on GPUs. In *ISCA* (2016), pp. 596–608.

[36] SHERRY, J., AND RATNASAMY, S. A Survey of Enterprise Middlebox Deployments. Tech. rep., 2012.

[37] SHI, L., CHEN, H., SUN, J., AND LI, K. vCUDA: GPU-Accelerated High-Performance Computing in Virtual Machines. *IEEE Transactions on Computers 61*, 6 (2012), 804–816.

[38] SUN, C., BI, J., ZHENG, Z., YU, H., AND HU, H. NFP: Enabling Network Function Parallelism in NFV. In *SIGCOMM* (2017), pp. 539–552.

[39] SUZUKI, Y., KATO, S., YAMADA, H., AND KONO, K. GPUvm: Why Not Virtualizing GPUs at the Hypervisor? In *USENIX ATC* (2014), pp. 109–120.

[40] TIAN, K., DONG, Y., AND COWPERTHWAITE, D. A Full GPU Virtualization Solution with Mediated Pass-Through. In *USENIX ATC* (2014), pp. 121–132.

[41] WANG, K., ZHANG, K., YUAN, Y., MA, S., LEE, R., DING, X., AND ZHANG, X. Concurrent Analytical Query Processing with GPUs. *VLDB* (2014), 1011–1022.

[42] WANG, Y., ZU, Y., ZHANG, T., PENG, K., DONG, Q., LIU, B., MENG, W., DAI, H., TIAN, X., XU, Z., WU, H., AND YANG, D. Wire Speed Name Lookup: A GPU-based Approach. In *NSDI* (2013), pp. 199–212.

[43] WANG, Z., YANG, J., MELHEM, R., CHILDERS, B., ZHANG, Y., AND GUO, M. Simultaneous Multikernel GPU: Multi-tasking throughput processors via fine-grained sharing. In *HPCA* (2016), pp. 358–369.

[44] XU, Q., JEON, H., KIM, K., RO, W. W., AND ANNAVARAM, M. Warped-slicer: Efficient intra-SM Slicing Through Dynamic Resource Partitioning for GPU Multiprogramming. In *ISCA* (2016), pp. 230–242.

[45] XUE, M., TIAN, K., DONG, Y., MA, J., WANG, J., QI, Z., HE, B., AND GUAN, H. gScale: Scaling up GPU Virtualization with Dynamic Sharing of Graphics Memory Space. In *USENIX ATC* (2016), pp. 579–590.

[46] ZHANG, H., ZHENG, Z., XU, S., DAI, W., HO, Q., LIANG, X., HU, Z., WEI, J., XIE, P., AND XING, E. P. Poseidon: An Efficient Communication Architecture for Distributed Deep Learning on GPU Clusters. In *USENIX ATC* (2017), pp. 181–193.

[47] ZHANG, K., HU, J., AND HUA, B. A Holistic Approach to Build Real-time Stream Processing System with GPU. *JPDC 83*, C (2015), 44–57.

[48] ZHANG, K., WANG, K., YUAN, Y., GUO, L., LEE, R., AND ZHANG, X. Mega-KV: A Case for GPUs to Maximize the Throughput of In-memory Key-value Stores. *VLDB* (2015), 1226–1237.