Simmer: Rate proportional scheduling to reduce packet drops in vGPU based NF chains

Avinash Kumar Chaurasia
avinashk@cse.iitb.ac.in
Indian Institute of Technology
Bombay
India

Anshuj Garg
anshujgarg@cse.iitb.ac.in
Indian Institute of Technology
Bombay
India

Bhaskaran Raman
br@cse.iitb.ac.in
Indian Institute of Technology
Bombay
India

Uday Kurkure*
Hari Sivaraman
Lan Vu
ukurkure@vmware.com
hsivaraman@vmware.com
lanv@vmware.com
VMware
USA

Sairam Veeraswamy
sveeraswamy@vmware.com
VMware
India

ABSTRACT

Network Function Virtualization (NFV) paradigm offers flexibility, cost benefits, and ease of deployment by decoupling network function from hardware middleboxes. The service function chains (SFC) deployed using the NFV platform require efficient sharing of resources among various network functions in the chain. Graphics Processing Units (GPUs) have been used to improve various network functions' performance. However, sharing a single GPU among multiple virtualized network functions (virtual machines) in a service function chain has been challenging due to their proprietary hardware and software stack. Earlier GPU architectures had a limitation: a single physical GPU can only be allocated to one virtual machine (VM) and cannot be shared among multiple VMs. The newer GPUs are virtualization-aware (hardware-assisted virtualization) and allow multiple virtual machines to share a single physical GPU. Although virtualization-aware, these GPUs still lack support for custom scheduling policies and do not expose the preemption control to users. When network functions (hosted within virtual machines) with different processing requirements share the same GPU, virtualization-aware GPUs' default round-robin scheduling mechanism proves to be inefficient, resulting in packet drops and lower throughput. This paper presents *Simmer*, an efficient mechanism for scheduling a network function service chain on virtualization-aware GPUs. Our scheduling solution considers the processing requirement of NFs in a GPU-based SFC, thus improving overall throughput by up to 29% and reducing the packet drop to zero compared to vanilla setup.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ICPP '22, Aug 29–Sep 01, 2022, Bordeaux, France
© 2022 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-9733-9/22/08…$15.00
https://doi.org/10.1145/3545008.3545068

ACM Reference Format:
Avinash Kumar Chaurasia, Anshuj Garg, Bhaskaran Raman, Uday Kurkure, Hari Sivaraman, Lan Vu, and Sairam Veeraswamy. 2022. Simmer: Rate proportional scheduling to reduce packet drops in vGPU based NF chains. In *51st International Conference on Parallel Processing (ICPP '22), August 29-September 1, 2022, Bordeaux, France.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3545008.3545068

1 INTRODUCTION

A typical network infrastructure consists of various network functions (NFs) working together to offer a specific network service. Multiple such NFs often process a network packet/traffic in a designated order. This arrangement of network functions is called a Network Function Chain (NFC) or Service Function Chain (SFC) [16]. Traditionally, network functions like firewalls, Intrusion Detection Systems (IDS), proxies, Network Address Translators (NAT), etc., were deployed using proprietary and special-purpose hardware middleboxes [30]. Hardware middleboxes deliver good performance but have higher deployment costs and are difficult to configure, manage, and upgrade. [32]

Network Function Virtualization (NFV) technology addressed the limitations of hardware middleboxes. NFV decoupled the network functions from hardware middleboxes and transformed how network functions were deployed and managed. NFV enables network function deployment over general-purpose commercial-of-the-shelf hardware (e.g., x86 servers), thereby removing specialized hardware middleboxes. With NFV, a network function runs as a software service over the commodity servers instead of a hardware middlebox. This software-based network function implementation is also known as virtual network functions (VNF). VNFs offer cost-effectiveness, flexibility, agility, and ease of scalability. Network operators often use a virtual machine or a container to deploy VNFs [3, 9].

Although NFV offers many benefits, one of the significant concerns with NFVs is their performance. Efforts have been made [20, 28] to improve NFV techniques and achieve a performance closer to the hardware middleboxes. Programmable hardware like Graphics

Processing Units (GPUs), Field Programmable Gate Arrays (FPGAs), etc., have also been explored for deploying network functions to improve the NFs performance [27, 29, 43].

Graphics Processing Units are specialized hardware devices capable of executing millions of threads in parallel. GPUs are now widely used for general-purpose computing given the computation power offered by them [1, 35]. At the same time, various works have shown the effectiveness of GPUs in improving the performance of the network functions like packet routing [17, 23, 36], SSL proxy [22] and SRTP [44] reverse proxy. Previous works on using GPUs for deploying network functions focused on running only single network functions on GPUs. Real-world networks often employ multiple network functions working together as a service function chain. In setups up with multiple NFs, efficient sharing of GPU among NFs is desirable.

Earlier GPU architectures were not virtualization-aware, and the PCI passthrough technique [41] was used to provide GPU access to virtual machines. PCI passthrough gives exclusive access of the GPU to a virtual machine, i.e., only one VM can use the GPU at a time. The proprietary hardware and drivers of GPUs constrain the possibility of designing a custom virtualization solution. The existing software-based virtualization solution work at the API level and require modification in either application programming interface (API) [11, 18, 33] or uses inefficient open-source GPU drivers [37]. The API-based virtualization solutions provide limited isolation between virtual machines, have compatibility issues, and are low performing [42].

In a multi NFV setup (involving a network function chain), where multiple virtual machines (network functions) share resources, efficient resource management is crucial. The newer NVIDIA GPUs [7] are virtualization-aware, i.e., they allow multiple virtual machines to share a single GPU. NVIDIA's virtualization technology exposes multiple virtual GPUs (vGPU), which can be assigned to virtual machines (vGPU VM). With this technology, each vGPU-enabled VM can host a GPU-based NF and be chained to form a GPU-based network function chain. However, these virtual machines (vGPU VM) share the physical GPU in a round-robin manner. Also, the GPU hardware/software does not expose the vGPU preemption control to the GPU user. The lack of preemption control prevents GPU users from incorporating custom vGPU scheduling policies.

The network functions in a service function chain (SFC) usually have heterogeneous compute requirements. The round-robin vGPU scheduling algorithm does not take the compute heterogeneity of NFs into account and gives an equal time slot (GPU share) to each vGPU VM (NF). The compute heterogeneity arises due to differences in the per-packet processing time of the network function. The lower the per-packet processing time, the higher the throughput. Round-robin NF scheduling in an SFC where a slow (bottleneck) NF is downstream and faster NF is upstream results in a packet loss. In such SFC, slow NFs will drop the packets already being processed by the upstream NF, causing the wastage of work (GPU cycles) and reducing overall throughput.

Ideally, the GPU share allocated to each NF (vGPU VM) should be proportional to the compute requirement of the NF. The challenges in implementing the rate-proportional scheduling of the vGPU VMs are two folds. First, there is a lack of preemption support in virtualization-aware hardware. Second, the virtualization-aware

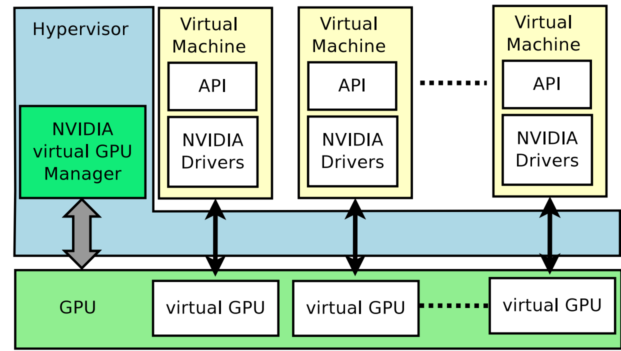


Figure 1: Architecture of virtualization aware GPUs.

hardware and associated software (drivers) are proprietary, so one cannot design a custom scheduling solution. Towards addressing these challenges following are the contributions of our work:

- Identify the issues and challenges of using a hardware-assisted GPU virtualization solution to deploy virtual network functions.
- Propose an approach to control the GPU share of VNFs hosted inside vGPU virtual machines (vGPU VMs).
- Design and implement *Simmer*, an efficient scheduling solution for deploying a virtual network service chain over the virtualization-aware GPU.
- Demonstrate the efficacy of *Simmer* by comparing it against vanilla setup and with various GPU virtualization modes.

The rest of the paper is organized as follows. Section 2 discusses background and motivation. Section 3 describes the design and implementation of *Simmer*. Section 4 presents the evaluation of our work. Section 5 discusses the related work and Section 6 concludes the paper.

2 BACKGROUND AND MOTIVATION

2.1 virtualization aware GPU hardware

Graphics Processing Units are specialized hardware that consists of thousands of cores working in a Single Instruction Multiple Thread (SIMT) manner. The GPU hardware can launch and schedule millions of threads that execute the same function on different data. GPUs were developed for accelerating the problems of the image processing [31] and computer vision domains [12]. However, they are now widely used as general-purpose compute accelerators [1, 35].

The earlier GPU architecture did not have support for virtualization. NVIDIA recently introduced virtualization-aware GPU hardware [13]. Figure 1 shows the architecture of hardware-assisted virtualization solution. With the help of the NVIDIA virtual GPU manager, a virtual-aware GPU hardware enables and exposes multiple instances of virtual GPUs (vGPUs). These vGPUs can be assigned to virtual machines and are accessible after installing NVIDIA GPU drivers.

The virtualization aware hardware can operate in two modes—virtual GPU(vGPU) mode and Multi-instance GPU (MIG) mode. These two modes differ in how multiple virtual GPUs share the physical GPU. The following is the description of the different modes:

Table 1: Time require to process a 1472B packet by the NF

	NF-1	NF-2	NF-3
Time (microseconds)	0.04	1.63	0.52

2.1.1 vGPU mode. vGPU mode multiplexes the physical GPU among multiple vGPU in a round-robin manner, i.e., the physical GPU is time-shared among multiple vGPUs. The memory of the physical GPU is statically partitioned among the vGPUs. NVIDIA virtual GPU manager exposes two parameters viz. vGPU profiles and vGPU scheduling algorithm to configure vGPU mode. vGPU profile determines the total number of vGPUs exposed and memory per vGPUs. The vGPU mode supports three vGPU scheduling algorithms— Fixed share, Equal share, and Best-effort. All these algorithms work in a round-robin manner. The best-effort scheduling algorithm is work conserving, and in this work, we configure the vGPU mode with the best-effort scheduling algorithm.

2.1.2 Multi-instance GPU(MIG) mode. NVIDIA MIG mode[8] partitions NVIDIA GPU into multiple GPU instances. Each MIG instance has a dedicated set of GPU cores and is predefined based on the MIG profile chosen at VM instantiation. Furthermore, each MIG instance has a separate and isolated path throughout the memory stack to ensure that workloads executing on one instance have a predictable performance regardless of the workload and its behavior on other instances. Dedicated GPU cores and isolated memory allow MIG instances to execute workloads simultaneously on a single physical GPU.

2.2 Heterogeneous compute requirement of NFs

Network infrastructure can include various network functions like firewalls, network address translation (NAT), routers, intrusion detection systems (IDS), etc. Each network function differs by its operation on the incoming network packet. We implement three network functions viz. Router [15], IPsec [25] and NIDS [2]. Table 1 shows the packet processing time of one packet of size 1472 bytes by these network functions. In Table 1, NF-1 is Router, NF-2 is IPsec and NF-3 is NIDS. Henceforth, we will mention these network function using their aliases NF-1, NF-2 and NF-3.

In a service function chain, a fast (low compute intensive) NF is often followed by a slow (high compute-intensive) NF. In such heterogeneous chains, to balance the processing rates (throughput) across the chain, the compute resources should be allocated to each NF in proportion to the incoming packet rate and packet processing time [26]. The slowest network function in the chain determines the overall throughput of the chain and is often termed bottleneck NF. Ideally, we would want to allocate the compute resources to each NF such that their throughputs are similar.

2.3 Issues with vGPU scheduling

The virtualization-aware hardware does not expose the vGPU preemption control to the GPU users. At the same time, the vGPU scheduling algorithms work in a round-robin manner with non-configurable time slice duration. The lack of preemption control and non-configurable scheduling algorithms limits the GPU users from employing custom scheduling policies. The NFV setups have

Table 2: Packet drop in vGPU setup for a chain of three NF when NF-2 is the bottleneck NF

	Packet Size (B)	NF-1	NF-2	NF-3
Packets Dropped (KPPS)	128	2.26	48.82(4%)	0
	256	2.35	97.65 (6%)	0
	512	2.01	289.14 (16%)	0
	1024	1.27	510.52 (36%)	0
	1472	0.167	722.33 (48%)	0

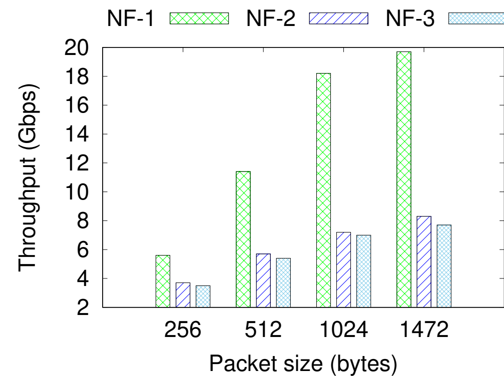


Figure 2: Throughput of each NF in the chain when NF-2 is the bottleneck NF

multiple NFs with heterogeneous compute requirements hosted inside virtual machines with one NF per VM. In such a setup, round-robin scheduling that equally shares the GPU among all the NFs results in packet loss and low throughput.

We implement three GPU-based NFs: NF-1(Router), NF-2(IPsec) and NF-3(NIDS). We created a simple network chain of NF-1, NF-2, and NF-3 in that order. Each vGPU VM hosts an NF inside, and every network packet traverses through all the NF. The vGPU VMs share the physical GPU in a default round-robin manner. In this setup, NF-2 is the bottleneck of the chain. From figure 2, we can observe that NF-2 suffers a drastic drop in throughput compared to NF-1 because NF-2 is compute-heavy NF yet not given enough GPU slots to match NF-1 throughput. Table 2 shows the packet dropped in thousand packets per second (KPPS) for different packet sizes for the same chain. Furthermore, from table 2, we can observe that NF-2, which is the bottleneck NF, experiences the highest packet drop (up to 48% of the total traffic), whereas the NF-1 has negligible packet drops, and NF-3 never suffers a packet drop.

In a network function chain, if one of the network functions has a slow processing rate (bottleneck NF), it affects the overall throughput of the chain. Also, the slow downstream NFs drop the packet that the upstream NF already processed. This action of dropping the partially processed packet wastes the work done by the upstream NF and, at the same time, it results in lower throughput.

3 DESIGN

We already discussed in previous sections the limitations of scheduling algorithms of state-of-the-art hardware-assisted GPU virtualization solutions. This section presents the design of *Simmer* — a software solution for efficient scheduling of vGPU-based NFs.

3.1 Solution requirements

There are two main requirements for designing a solution for the efficient scheduling of NFs. First, the NFs should get the GPU share in proportion to their processing requirement. Second, we should be able to quickly identify the bottleneck NF responsible for the slow down of the entire service function chain.

3.1.1 Rate proportional sharing of GPU. In a virtual GPU (vGPU) based Service Function Chain (SFC), the network functions sharing a single physical GPU (via vGPU VMs) should be given the scheduling opportunity according to their processing requirements. Ideally, the slower NF should get the larger share of the physical GPU. The NVIDIA vGPU scheduler controls VM scheduling on the physical GPU at the hypervisor. Though the scheduler is suitable for most workloads, it does not efficiently utilize GPU when NFV chaining is concerned because the scheduler is agnostic to the workloads running inside VMs. Since the scheduler is unaware of the workload and its dependency, the scheduling decisions often lead to poor performance of the NFV chains. Furthermore, the scheduler can not be replaced by a custom scheduler, nor can it be altered.

NVIDIA's best effort scheduler works as follows: It schedules the active VMs on the physical GPU in a round-robin. However, previous works on NFV deployment over vGPU [4] observed that the best effort scheduler employs some heuristic to assign GPU to the VM with an active GPU workload. If all the VMs have active workloads, it schedules them in the round-robin. In summary, the scheduler uses *active GPU workload* as meta information while scheduling the VMs on the GPU. Based on this empirical knowledge, we wish to control VM assignment to the physical GPU by altering the *active GPU workload* meta information.

3.1.2 Identification of the bottleneck NF. The second requirement of the solution is to determine the bottleneck NF and then coordinate with other VMs to allow the bottleneck NF a larger share of the GPU. If incoming packet flow is not controlled and matched with the bottleneck NF, excess packets get dropped or queued for later processing. Since packet drop is never desirable in any network, queuing excess packets is better. For a bottleneck NF, packets in the queue continue to increase until the queue gets full. Afterward, packets get dropped. A solution must avoid these situations. A queue occupancy is an excellent metric to measure the speed of NF processing, hence can be used to find a bottleneck NF. Since GPU is a shared resource among all the NFs of the chain, an increase in the share of the bottleneck NF on the GPU may starve another NF and temporarily make it a bottleneck NF. Hence, a solution must be able to handle such situations.

3.2 Design Choices

3.2.1 Centralized vs Distributed design. There can be two ways to measure individual NF's Receive (RX) queue and modify the meta information in each NF: central and distributed. One machine (virtual or physical machine) can act as an arbitrator in a central approach and gather RX queue information from all the NFs. Based on the collected information, the arbitrator chooses one NF and informs the selected NF that it can use a physical GPU. The arbitrator must inform the rest of the NFs to wait (pause their operation) and not use the GPU; otherwise, the vGPU scheduler will schedule all the VMs in a round-robin. If there are n NFs, then the central system will periodically receive n messages and have to reply to n VMs regarding who is allowed to use GPU or not. Furthermore, the system must define the frequency at which the central arbitrator must collect these messages.

Though central arbitrator systems have advantages such as a global view of the chain and control over the scheduling via meta-information modification, it suffers from drawbacks such as single point of failure and scalability. These drawbacks are well known to researchers for similar systems such as SDN (software-defined networks) [34]. Similar to our central arbitrator concept, SDN (software-defined networks) also employ a central controller to solve complex networking problems and suffer from identical problems [24]. However, SDN does not require frequent updates, and it only updates routers for a new packet arrival for which routes are not available at the data plane. In contrast, our arbitrator has to continuously update the meta-information to control the VM execution on the GPU. Hence the delay caused by the messages to and from the central arbitrator may result in sub-optimal scheduling. Furthermore, the central arbitrator must inform each NF regarding the GPU slot availability, which may require complex scheduling techniques.

The distributed approach allows scalability and quick response but lacks the global view of the RX queue of NFs. In the distributed approach, each NF collects the information required to make an independent decision regarding the GPU slot's availability. The distributed approach becomes complex if each NF wishes to have a global view and then sync their decision regarding the GPU slot availability. Having a global view makes the system less scalable because the number of messages required for a system to have a global view increases exponentially per VM addition. Instead, suppose each NF has local information and can guess or make an informed decision regarding the GPU slot availability based on that local information. In that case, that system becomes highly scalable and can make quick decisions, possibly without complex techniques. We choose to adopt this local information-based distributed approach due to its apparent benefit of scalability. We created a software-based receive (RX) queue to enqueue all the incoming packets. This RX queue feeds packets to GPU-based NFs in our system. The idea is that each NF assesses whether they are the bottleneck or not based on its RX queue occupancy. If any NF finds itself a bottleneck, it informs its upstream NF to take appropriate action. Since upstream NF is responsible for pushing packets to bottleneck NF, the upstream NF must pause computation on the GPU to reduce contention for GPU access. Furthermore, the RX queue occupancy of this upstream NF continues to increase, resulting in it becoming the new bottleneck NF. Due to the distributed nature of the solution, every upstream NFs eventually pause its computation unless the original bottleneck NF gets enough GPU access and reduces its RX queue occupancy significantly. If first NF becomes the bottleneck, then it will a signal to the source of

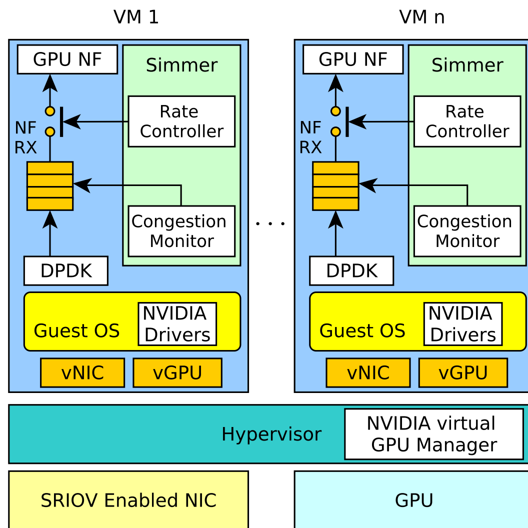


Figure 3: Architecture of *Simmer.*

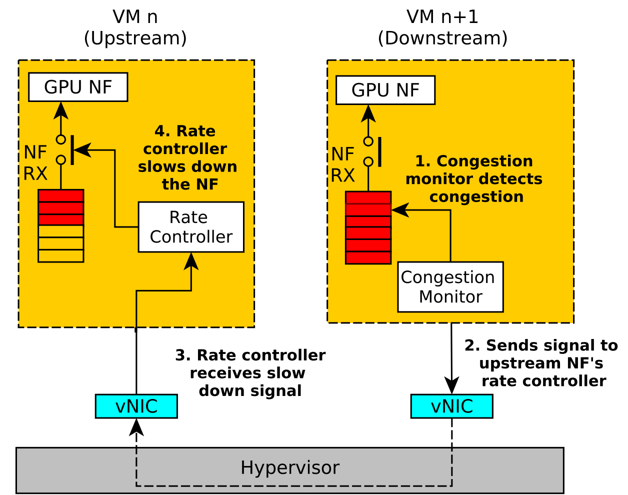


Figure 4: Rate control mechanism between two NFs.

the packets. Now it is up to the infrastructure provider to handle this signal and use any existing solutions like it can either queue the packets or spawn a new chain on another physical system and load balance the traffic. We implemented our solution as a *Simmer* module, and the subsequent section explains the architecture of *Simmer.*

3.2.2 vGPU NF Vs Process NF. A network function can be hosted within a vGPU-enabled virtual machine or implemented as a GPU-based process. In a process-based setup, each network function in an SFC is implemented as a process. In such a setup, all the processes share the same GPU in a round-robin order [13]. The GPUs schedule the multiple processes and multiple vGPUs (VM) in the same manner. At the same time, like vGPUs, no explicit scheduling-control knobs are exposed by GPUs for governing the scheduling of multiple processes. However, using vGPUs for hosting network functions instead of processes have additional advantages. Firstly, virtual machines provide better resource control than processes. Compute resources of a network function hosted on a virtual machine can be scaled up and down depending upon the requirement of the NF. Secondly, vGPUs (VM) provides better GPU resource isolation than GPU-bound processes. Each vGPU has its exclusive memory, and the memory access of one vGPU is not affected by the memory access of other vGPU. However, in the case of processes, every process has a global view of the entire GPU memory, and memory failure (like memory over-allocation or out-of-bound access) of one process can crash the other GPU-bound process.

3.3 Architecture of Simmer

Figure 3 shows the architecture of our solution setup— *Simmer. Simmer* uses single root I/O virtualization (SR-IOV) [5] enabled network interface cards and virtualization-aware NVIDIA GPUs to provision virtual machines with a virtualized network interface card (vNIC) and virtual GPU (vGPU) resources, respectively. Each virtual machine hosts a single network function (GPU NF). *Simmer* uses DPDK API [10] for bypassing the Operating System (OS) kernel network stack of the guest Operating System for efficient packet delivery. The NVIDIA virtual GPU manager sits inside the hypervisor and facilitates the physical GPU virtualization. The virtual machine's guest OS requires NVIDIA drivers to enable access to the virtual GPUs.

The *Simmer* module is present inside each virtual machine. *Simmer* has two main components– Congestion monitor and Rate controller, as shown in figure 4.

3.3.1 Congestion Monitor. : The task of the Congestion monitor is to keep track of the occupancy of the NF receive (NF RX) queue of its corresponding network function (virtual machine). There is a threshold qt associated with the RX queue of each network function. The NF RX queue threshold qt denotes the percentage occupancy of the queue. From here onwards, the RX queue always refers to the NF RX queue shown in the figure 4. The Congestion monitor periodically checks the queue and sends a *slow-down* signal to the rate controller of the upstream NF when queue occupancy crosses qt.

3.3.2 Rate controller. : The Rate controller controls the packet flow between the RX queue and the network function. It acts like an ON/OFF switch to start and stop the packet flow between the RX queue and NF. When a rate controller receives a *slow-down* signal from a downstream NF, it temporarily cuts the flow of packets from the RX queue to the NF. However, when it receives a *fasten-up* signal, it switches on the flow of packets.

3.4 Simmer Implementation

In our Simmer, GPU-based NFs are part of the GPU kernel. The GPU kernel launches the kernel when it receives a fixed number of packets, i.e., a batch of packets. The GPU kernel receives the batch of packets from the RX queue, and DPDK enqueues incoming packets in the RX queue.

At boot, *Simmer* initializes the modules along with three user specified variables: *batch_size*, *flow_status*, and *qt*. *batch_size* is the number of the packets required for a GPU kernel launch. *flow_status* defines the status of packet flow between the RX queue and NF. *qt* is the RX queue threshold, and its value determines when to trigger signals for *slow-down (pause)* or *fasten-up (resume)*. The flow of packets in the framework is as follows: DPDK enqueues packets in the RX queue of its respective NF on packet reception. If *flow_status* is non-zero and the RX queue occupancy crosses *batch_size*, the rate controller sends packets to the NF. However, if *flow_status* is zero, the rate controller pauses the packet flow from the RX queue to NF. When NF receives *batch_size* packets, it launches the GPU kernel if previous GPU kernels have completed their execution on the GPU. Once the RX queue occupancy crosses the threshold value (qt), the congestion monitor triggers the *slow-down* signal to adjacent upstream NF. On the other hand, if the queue occupancy goes below the threshold(qt), the congestion monitor triggers the *fasten-up* signal to adjacent upstream NF.

On *slow-down* signal reception, the rate controller of the upstream NF sets the *flow_status* to zero; hence it pauses the flow of packets to NF, resulting in halting further kernel launches. Alternatively, suppose the upstream NF receives a *fasten-up* signal, the NF sets the *flow_status* to the non-zero value resulting in the resumption of the kernel launches. Every time NF refrains from launching the kernel, it excuses itself from being scheduled on the physical GPU resulting in other NFs (especially bottleneck NFs) using the extra time slot to improve the chain's throughput.

We implemented the signal using the UDP protocol and sent it to other NFs over the network. To keep it fast and straightforward, we avoided TCP. Even if the same signal is sent multiple times for a single event, the upstream NF response stays stable due to binary action taken by the flow control.

4 EXPERIMENTATION

Our experimental setup consists of multiple VMs hosted on VMware ESXi-7.0.2 hypervisor [39]. The hypervisor runs on a Dell machine with Intel(R) Xeon(R) Gold 5218R CPU having 40 cores with 2.10GHz frequency and 768 GB of RAM. NVIDIA's A100 GPU [6] with 6912 cores and 40 GB of GPU memory is attached to the system, and it supports a maximum of ten vGPUs or seven MIG instances. Each VM has 30 GB RAM and eight vCPUs. All the VMs except one (traffic generator) are attached to a vGPU instance (when MIG mode is disabled), and the hypervisor uses a best-effort scheduler for vGPU scheduling. However, when MIG mode is enabled, they use the MIG GPU instance. Each VM uses single root I/O virtualization (SR-IOV) enabled virtual NIC (vNIC) backed by 25 Gbps Intel Ethernet NIC. Furthermore, each VM runs Ubuntu 20.04 as guest OS and uses DPDK 18.11 for network IO.

4.1 NFs in service function chain

We implement three NFs viz. NF-1, NF-2, and NF-3 and use them in our evaluation. All three NFs have different compute requirements. NF-2 and NF-3 are IO and compute-intensive NFs, whereas NF-1 is only IO intensive. Figure 5 shows the individual NF performance in a hardware-assisted virtualized GPU environment over A-100 GPU in two configurations: when MIG is enabled (MIG instances) and when MIG is disabled (vGPU instances). Each NF (VM) gets exclusive access to the physical GPU in vGPU and MIG configurations. MIG configuration uses A100-7-40C profile (table 3), and vGPU configuration uses A100-40C profile (table 4). Hence, only one NF (VM) is deployed on the physical GPU in either configuration to measure the maximum individual throughput each NF can offer without any interference. As we can observe from figure 5, the NFs on the MIG are either equal or better than their corresponding performance on the vGPU instance. We believe virtualization overhead in the vGPU instances such as vGPU scheduling may be the reason for the lower throughput of the NFs over vGPU.

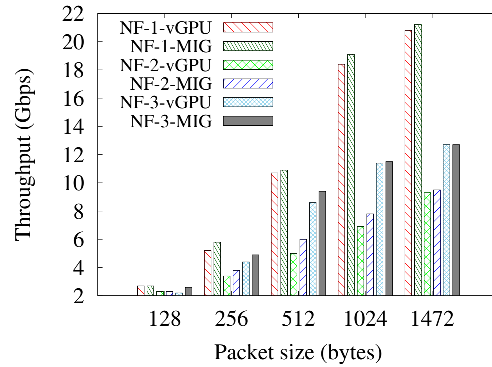


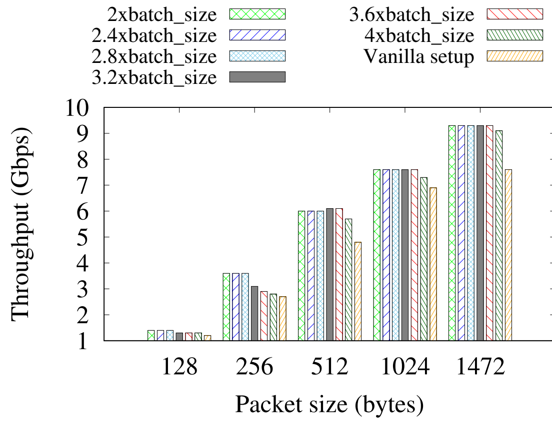
Figure 5: NF's individual performance when a one VM occupies whole GPU.

Table 3: MIG instances supported by NVIDIA A-100

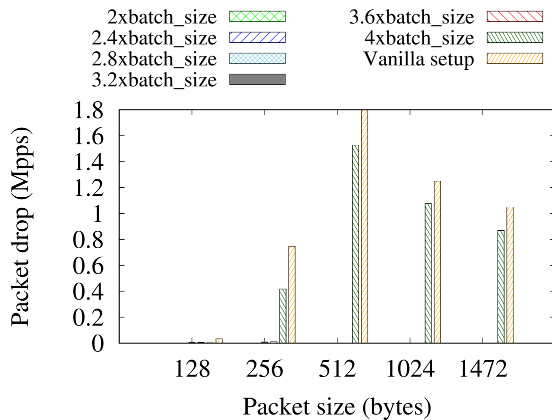
MIG profile	memory per MIG instance	No. of MIG instances exposed
A100-1-5C	5 GB	7
A100-2-10C	10 GB	3
A100-3-20C	20 GB	2
A100-4-20C	20 GB	1
A100-7-40C	40 GB	1

Table 4: vGPU profiles supported by NVIDIA A-100

vGPU profile	memory per vGPU	number of vGPUs exposed
A100-4C	4 GB	10
A100-5C	5 GB	8
A100-8C	8 GB	5
A100-10C	10 GB	4
A100-20C	20 GB	2
A100-40C	40 GB	1

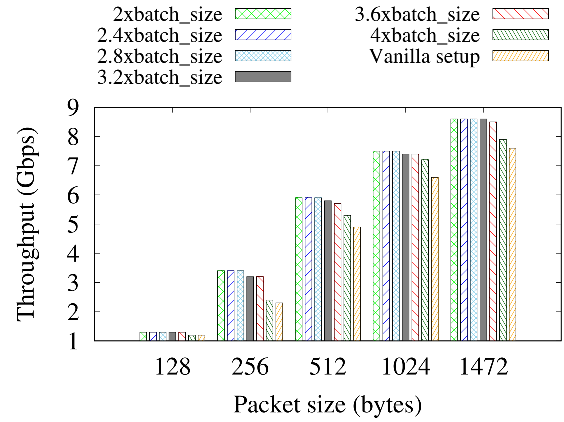


(a) Throughput

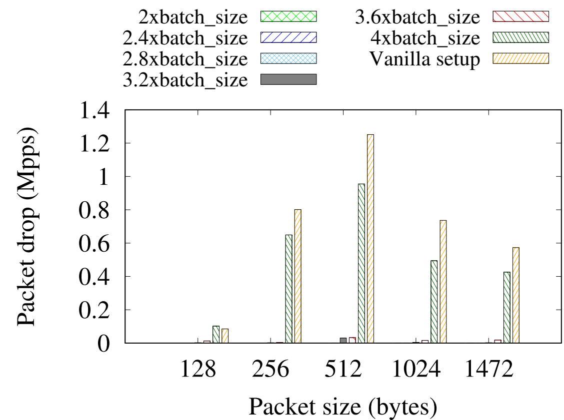


(b) Packet drops

Figure 6: Chain of three heterogeneous NF: *Simmer* with different queue threshold values (*qt*) vs vanilla setup



(a) Throughput



(b) Packet drops

Figure 7: Chain of three homogeneous NF: *Simmer* with different queue threshold values (*qt*) vs vanilla setup

4.2 Queue threshold (qt) selection

Simmer modifies the meta-information to control the scheduling of the VMs (NF) on the physical GPU. It triggers a signal when queue occupancy crosses a predefined threshold in the RX queue. We perform a set of experiments to understand the impact of threshold value on the chain's throughput. We vary the threshold value in multiple of *batch_size* because GPU is processing *batch_size* packets at a time (section 3.4). We varied the threshold value from twice the *batch_size* (2×*batch_size*) to quadruple the *batch_size* (4×*batch_size*). From figure 6, we observed that for a heterogeneous chain of three NF, packet drop reduces to zero (figure 6b) when the threshold value stays in the range of 2× to 2.8× of the *batch_size*, and system throughput stays highest in this threshold range among the rest of the threshold values (figure 6a). We kept the minimum threshold value at 2× of the *batch_size* due to a possible scenario explained as follows. A lower value means NF triggers the *slow-down* signal with packets fewer than 2× *batch_size*. A possibility arises that GPU (bottleneck NF) is already processing *batch_size* (1×) packets out of these packets and is about to complete. Due to the *slow-down* signal, the flow of incoming packets ceases, and the bottleneck NF is left with less than *batch_size* packets and cannot relaunch the kernel. The bottleneck NF sent the *slow-down* signal to ask for a more significant share of the GPU, and upstream NF obliged by pausing their kernel launch. However, bottleneck NF cannot launch the kernel due to the unavailability of enough packets; hence it becomes counterproductive.

We performed similar experiments for a chain of three homogeneous NFs (chain of three NF-2) to understand the impact of varying threshold values on the performance when three NFs of similar computing requirements are chained. Figure 7 shows the homogeneous chain's performance when the threshold is varied. The behavior of *Simmer* for a homogeneous chain is very similar to a heterogeneous chain concerning a threshold value. However, a homogeneous chain attains a throughput improvement of 13% compared to a heterogeneous chain that gets 29% throughput improvement when *Simmer* is employed. Heterogeneous chain with *Simmer* performs comparatively better as rate-proportional scheduling is possible due to the heterogeneous nature of NFs. In contrast, default round-robin scheduling provides rate-proportional scheduling in a homogeneous NF. Most of the packet drops in a homogeneous NF chain happen at the first NF of the chain. Hence GPU resource is not

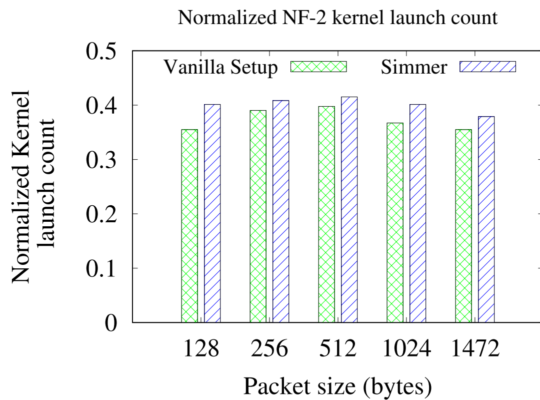


Figure 8: Improvement in bottleneck kernel launch: *Simmer* vs Vanilla setup

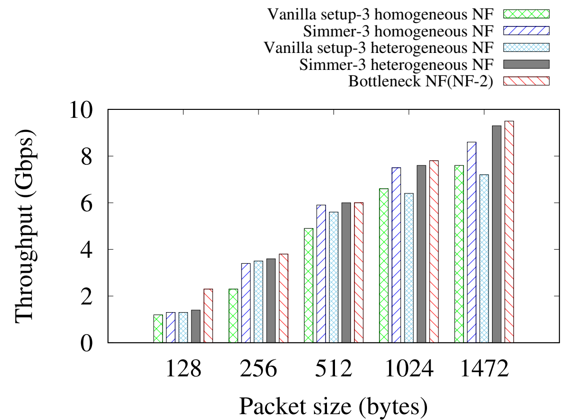


Figure 9: Static chaining of three NFs: Throughput comparison of *Simmer* vs Vanilla setup

wasted. We observed that *Simmer* also reduces memory contention. Without *Simmer*, many incoming packets are received and kept in memory by DPDK and later dropped due to the full NF RX queue. *Simmer* reduces this drop. Hence a homogeneous NF chain with *Simmer* employed benefits from contention reduction due to lower packets IO in memory. We also performed the same experiments by forming a homogeneous NF chain consisting of three NF-3 and three NF-1 and found similar performance for both homogeneous chains. In summary, *Simmer* reduces memory contention and provides rate-proportional scheduling. We are using $2.8\times batch_size$ as a queue threshold (qt) for the rest of the experiments in the paper because, for this value, both homogeneous and heterogeneous chain shows zero packet drops.

4.3 Effectiveness of rate proportional scheduling

Simmer increases the GPU share of the bottleneck NFs present in an SFC. The best way to measure this is by calculating each NF time on the GPU and then normalizing it by the total time of all the NFs on the GPU. If *Simmer* increases this normalized time for the bottleneck NF, it proves *Simmer* ability to increase GPU share for the bottleneck NF.

CUDA provides the API to measure the kernel execution time, and its precision is unquestionable in bare metal execution or passthrough mode. However, as per our evaluation, in the vGPU context, the same API can not measure kernel execution time with precision yet (as the technology matures, precision may improve). Hence, we use a normalized count of kernel launches as a workaround. To measure the *Simmer* efficacy, we normalize the bottleneck NF kernel launch count with all of the NF's combined kernel launch count and compare the values when *Simmer* is employed vs. when it is not. In our experiment, SFC consists of three NFs: NF-1(Router), NF-2(IPsec), and NF-3 (NIDS), in the order where NF-2 is the bottleneck NF for the chain. Figure 8 shows the number of NF-2 kernel launches normalized by the combined kernel launch count of all the NF. From the figure 8, we can observe that NF-2 normalized kernel launches increase when simmer is employed compared to vanilla setup, which proves the efficacy of *Simmer*.

4.4 Performance of *Simmer* in a static SFC

A static SFC is a chain where each incoming packet must go through all the network functions in a predefined order. In reality, static chain deployments are rare, but some cases do exist; hence it is imperative to test the framework for such setups. We performed experiments with two static SFCs: homogeneous SFC and heterogeneous SFC. A homogeneous SFC consists of three identical NFs (three NF-2), whereas a heterogeneous SFC consists of the NF-1, NF-2, and NF-3 in the strict chain order. Being the most compute-heavy among the rest of the NFs, NF-2 is a bottleneck NF for the chain.

From the figure 9, we can observe that the throughput of the chain when three NFs (homogeneous or heterogeneous) are chained improves when *Simmer* is employed compared to when it is not. For the heterogeneous chain, throughput improvement can be observed as high as 29% (figure 9). On analysis, we found that the *Simmer* improves the throughput because it reduces the packet drop as shown in the figures 7b and 6b. Dropped packets consume compute, and network resources in upstream NFs but do not contribute to real traffic (and throughput). *Simmer* allows bottleneck NFs to gain more compute opportunities on the physical GPU, reducing packet drops and resource wastage, thus improving throughput. The following section will analyze the performance of the *Simmer* for dynamic SFC, which is a more widely deployed model.

4.5 Performance of *Simmer* in a dynamic SFC

In a Dynamic SFC, packets may be processed on any NF in any order as per predefined policy. A set of NFs can form multiple SFC with varying chain lengths. To analyze the performance of the *Simmer*, we formed a dynamic SFC over vGPU such that different chain process packets based on destination IP address. Furthermore, we analyzed the performance of *Simmer* on both homogeneous dynamic SFC and heterogeneous dynamic SFC. We configured our setup to process two separate traffic with equal flows (same incoming rate) in two different SFCs. For homogeneous dynamic SFC, a chain of three identical NFs (NF-2) processes traffic-1, and a chain

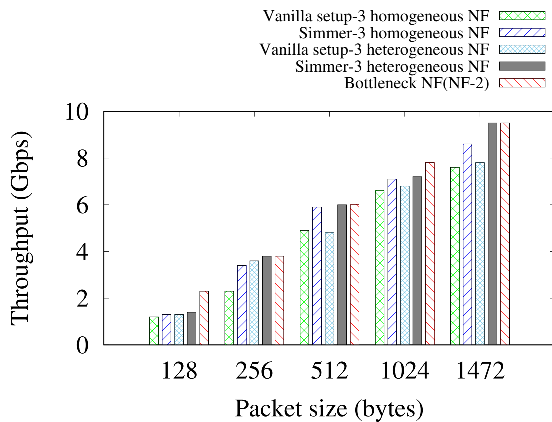


Figure 10: Dynamic chaining of three NFs: Throughput comparison of *Simmer* vs vanilla setup

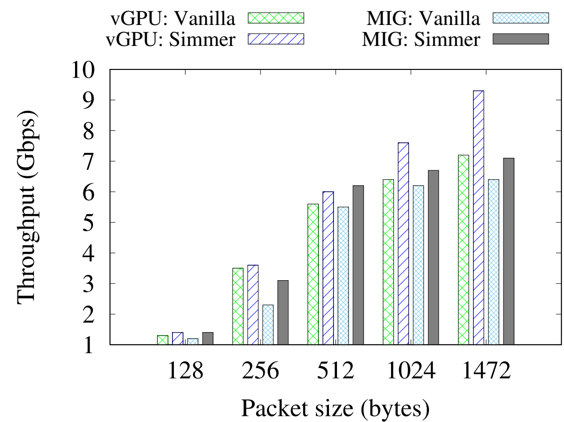


Figure 11: vGPU vs MIG instances: Throughput of three heterogeneous NF chain

Table 5: MIG profile assigned to each NF

NF	MIG profile	compute (SM)	memory (GB)
NF-1	1-5c	1	5
NF-2	3-20c	3	20
NF-3	2-10c	2	10

of two identical NFs (NF-2) processes traffic-2. For heterogeneous dynamic SFC, a chain of NF-1, NF-2, and NF-3 process traffic-1, whereas a chain of NF-2 and NF3processes traffic-2. NF-2 is the bottleneck NF for either chain in the setup.

Though software-defined networking (SDN) [30, 38] is considered a standard to achieve dynamic routing nowadays, it is relatively complex and requires complex configuration setups. Furthermore, SDN is beneficial in large networks where static routing may lead to manual errors in routing tables. However, our experimental setup consists of only three nodes (VMs) and does not require complex routing. Hence we employ static routing to construct a table, and each packet consults the table to route packets through a set of NFs to form a dynamic SFC. Our static routing table consists of *packet destination IP*, *current NF*, and *next NF*. *packet destination IP* is the IP and matched against the destination IP in the IP header of each packet. *current NF* is the NF where the packet is processed just now and looking for the following NF for further processing. Every packet is matched against both *packet destination IP* and *current NF* to determine where the packet should head for further processing, i.e., the *next NF*. From the figure 10, we can observe that the *Simmer* improves the throughput for both homogeneous and heterogeneous dynamic NFV chaining. We observed an improvement of 13% for a homogeneous dynamic SFC, whereas, for a heterogeneous dynamic SFC, we observed an improvement of 23%. Similar to static SFC, reduction in packet drop due to employment of the *Simmer* is the main reason behind the performance improvement.

4.6 NFV chaining: vGPU vs MIG instances

NVIDIA recently introduced MIG instances [8] for spatially sharing the GPU among virtual machines in a virtualized environment. As MIG instances (MIG VMs) have exclusive rights over a set of GPU cores (space sharing), their performance might differ from vGPU instances (time-sharing of GPUs). We analyzed and compared the MIG against vGPU to answer which suits better to deploy SFC. To deploy NFs over MIG instances, we partitioned the GPU into three static MIG instances and assigned it to VMs (NFs). We statically chained three NFs: NF-1, NF-2, and NF-3. Furthermore, we assigned MIG profiles to each NFs as shown in the table 5 to proportionate the GPU cores as per NF compute requirement. A-100 GPU that we have used in our experiments has 40 GB memory, 6912 GPU (CUDA) cores, and eight SMs. Each SM consists of 864 GPU cores. However, MIG allows only seven SMs for workload executions.

From the figure 11, we can observe that *Simmer* improves the performance of the NFV chaining over MIG instances as well as vGPU instances. Compared to vGPU instances, the *Simmer* performance in the MIG instance does not come from better scheduling of the NFs as there is no scheduler for the MIG. Instead, *Simmer* reduces the packet drop rate, which improves the utilization of the cores. Due to the reduction in packet drops, most of the processed packets contribute to the chain throughput.

However, SFC over vGPU performs better than MIG instances due to sharing of GPU cores. In vGPU setups, if any NF does not require computing momentarily, other NFs can utilize those cores, which can not happen in MIG setups. Furthermore, *Simmer* efficient NF scheduling utilizes the cores better, thus further boosting the throughput.

5 RELATED WORK

Initial works concerning NFV deployment on GPU (including PacketShader [17], SSLShader [22], Kargus [21], APUnet [14], etc.) focuses on single NF deployment. PacketShader [17] solves the challenges concerning kernel overhead for packet IO and shows that GPU can provide 10Gbps router throughput. SSLshader [22] and Kargus [21] employ GPU to accelerate cryptographic computation and intrusion detection system respectively. Kargus [21] is unique in the way that it divides task between CPU and GPU and execute the task on both in parallel. GPUNFV [40] focuses on stateful packet processing on the GPU. GEN [45] supports RTC-based SFC on the GPU, i.e., there is no isolation: one NF can easily corrupt the memory of other NFs. Yang Hu and Tao li [19] proposed a graph-based traffic allocation scheme to divide the work

between CPU and GPU. These works use a GPU to accelerate a single NF on either bare metal GPU or passthrough mode GPU. Whereas *Simmer* is designed for vGPU-based NFV chaining, i.e., it uses a single GPU for a chain of multiple NFs. In terms of NFV chaining, G-net [43] is one of the well-known works on the GPU in cloud setups. It uses HYPERQ-enabled NVIDIA GPU for accelerating NFV chains. However, NFs in G-net [43] share the GPU in passthrough mode and do not provide resource isolation. In contrast, *Simmer* uses vGPU, which provides resource isolation among NFs. Both G-net and *Simmer* faced different challenges to solve the respective problem of performance enhancement due to the different nature of the GPU and the property associated with it. NFVnice [26] provides rate proportional sharing of NFs, similar to *Simmer*. However NFVnice scheduling framework is applicable for NFs on the CPU, and the same techniques can not be used with *Simmer* because NFVnice schedules NFs by providing suggestions to the Linux scheduler from userspace. Since the Linux scheduler is not proprietary, its inner working is well known and can be modified and influenced via system APIs. Recent work on NFV deployment on vGPU shows that NFs underutilize passthrough GPUs, and infrastructure providers can significantly benefit from deploying such workloads on vGPU [4]. However, this work does not consider NF chaining on the same physical GPU, which faced challenges of unfair scheduling due to the work-agnostic nature of the proprietary scheduler.

6 CONCLUSION

The efficient resource management among virtual network functions in a service function chain is vital for getting a higher throughput and lower packet drops. Scheduling multiple VNFs on modern virtualization-aware GPU is challenging due to their proprietary hardware/software stack, no support for incorporating custom scheduling policies, and lack of preemption control. Also, the default virtual GPU scheduling policies are agnostic to the processing demands of VNFs. Towards overcoming these challenges, we present *Simmer* a solution for efficient scheduling of VNFs (hosted inside VMs) over physical GPU. *Simmer* schedules the NFs based on their packet processing requirements. We demonstrate by thorough experiments that *Simmer* reduces the packet drops to zero and improves the overall chain throughput up to 29%.

7 FUTURE WORK

We believe *Simmer* is the first step toward using vGPUs for NF chaining in the cloud. To further build the trust with the industry, we would like to extend our work *Simmer* for multiple systems having diverse GPUs, including hardware-assisted vGPUs, MIG instances, and passthrough-mode GPUs. We want to analyze the performance of *Simmer* for such a system in the future.

REFERENCES

[1] Jacob T Adriaens, Katherine Compton, and Nam Sung Kim et al. 2012. The case for GPGPU spatial multitasking. In *IEEE International Symposium on High-Performance Comp Architecture*. IEEE.

[2] Alfred V. Aho and Margaret J. Corasick. 1975. Efficient String Matching: An Aid to Bibliographic Search. *Commun. ACM* 18 (1975).

[3] Roberto Bonafiglia, Ivano Cerrato, and Francesco Ciaccia et al. 2015. Assessing the performance of virtualization technologies for NFV: A preliminary benchmarking. In *Fourth European Workshop on Software Defined Networks*. IEEE.

[4] Avinash Chaurasia, Uday Kurkure, and Hari Sivaraman et al. 2020. Network functions in virtualized GPU environment. In *International Conference on High Performance Computing Simulation (HPCS)*.

[5] Jianglu Chen, Jian Li, and Fei Hu. 2013. SR-IOV Based Virtual Network Sharing. In *Proceedings of the Second International Conference on Innovative Computing and Cloud Computing*. 213–218.

[6] NVIDIA Corporation. [n. d.]. *NVIDIA A100 Tensor Core GPU*. https://www.nvidia.com/en-in/data-center/a100/ [accessed 26-Nov-2021].

[7] NVIDIA Corporation. [n. d.]. *NVIDIA GPUs For Virtualization*. https://www.nvidia.com/en-in/data-center/graphics-cards-for-virtualization// [accessed 18-Nov-2021].

[8] NVIDIA Corporation. [n. d.]. *NVIDIA Multi-Instance GPU*. https://www.nvidia.com/en-in/technologies/multi-instance-gpu/ [accessed 13-Nov-2021].

[9] Richard Cziva, Simon Jouet, and Kyle JS White et al. 2015. Container-based network function virtualization for software-defined networks. In *2015 IEEE symposium on computers and communication (ISCC)*. IEEE.

[10] DPDK. 2019. Part 1: Architecture Overview. https://doc.dpdk.org/guides/prog_guide/overview.html. Accessed: 2019-03-08.

[11] José Duato, Antonio J Pena, and Federico Silla et al. 2010. rCUDA: Reducing the number of GPU-based accelerators in high performance clusters. In *International Conference on High Performance Computing & Simulation*. IEEE.

[12] James Fung and Steve Mann. 2008. Using graphics devices in reverse: GPU-based Image Processing and Computer Vision. In *IEEE International Conference on Multimedia and Expo*.

[13] Anshuj Garg, Purushottam Kulkarni, and Uday Kurkure et al. 2019. Empirical Analysis of Hardware-Assisted GPU Virtualization. In *26th IEEE International Conference on High Performance Computing, Data, and Analytics (HiPC)*.

[14] Younghwan Go, Muhammad Jamshed, and YoungGyoun Moon et al. 2017. APUNet: Revitalizing GPU As Packet Processing Accelerator. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation (NSDI)*. USENIX Association.

[15] P. Gupta, S. Lin, and N. McKeown. 1998. Routing lookups in hardware at memory access speeds. In *proceedings of IEEE Conference on Computer Communications (INFOCOM)*.

[16] Joel Halpern, Carlos Pignataro, et al. 2015. Service function chaining (sfc) architecture. In *RFC 7665*.

[17] Sangjin Han, Keon Jang, and KyoungSoo Park et al. 2010. PacketShader: a GPU-accelerated software router. *ACM SIGCOMM Computer Communication Review* 40 (2010).

[18] Cheol-Ho Hong, Ivor Spence, and Dimitrios S Nikolopoulos. 2017. GPU virtualization and scheduling methods: A comprehensive survey. *ACM Computing Surveys (CSUR)* 50 (2017).

[19] Yang Hu and Tao Li. 2018. Enabling Efficient Network Service Function Chain Deployment on Heterogeneous Server Platform. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*.

[20] Jinho Hwang, K K_ Ramakrishnan, and Timothy Wood. 2015. NetVM: High performance and flexible networking using virtualization on commodity platforms. *IEEE Transactions on Network and Service Management* 12 (2015).

[21] Muhammad Asim Jamshed, Jihyung Lee, and Sangwoo Moon et al. 2012. Kargus: A Highly-scalable Software-based Intrusion Detection System. In *Proceedings of the ACM Conference on Computer and Communications Security*. ACM.

[22] Keon Jang, Sangjin Han, and Seungyeop Han et al. 2011. SSLShader: Cheap SSL Acceleration with Commodity Processors.. In *NSDI*.

[23] Anuj Kalia, Dong Zhou, and Michael Kaminsky et al. 2015. Raising the bar for using GPUs in software packet processing. In *12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15)*.

[24] Murat Karakus and Arjan Durresi. 2017. A survey: Control plane scalability issues and approaches in Software-Defined Networking (SDN). *Computer Networks* 112 (2017).

[25] S. Kent. 2005. *IP Encapsulating Security Payload (ESP)*. RFC.

[26] Sameer G Kulkarni, Wei Zhang, and Jinho Hwang et al. 2020. Nfvnice: Dynamic backpressure and scheduling for nfv service chains. *IEEE/ACM Transactions on Networking* 28 (2020).

[27] Xiaoyao Li, Xiuxiu Wang, and Fangming Liu et al. 2018. DHL: Enabling flexible software network functions with FPGA acceleration. In *38th IEEE International Conference on Distributed Computing Systems (ICDCS)*. IEEE.

[28] Joao Martins, Mohamed Ahmed, and Costin Raiciu et al. 2014. ClickOS and the art of network function virtualization. In *11th {USENIX} symposium on networked systems design and implementation ({NSDI} 14)*.

[29] Gabriel S Niemiec, Luis MS Batista, and Alberto E Schaeffer-Filho et al. 2019. A survey on FPGA support for the feasible execution of virtualized network functions. *IEEE Communications Surveys & Tutorials* 22 (2019).

[30] Zafar Ayyub Qazi, Cheng-Chun Tu, and Luis Chiang et al. 2013. SIMPLE-fying middlebox policy enforcement using SDN. In *Proceedings of the ACM SIGCOM*.

[31] Harald Scheidl. 2018. GPU Image Processing using OpenCL. https://towardsdatascience.com/get-started-with-gpu-image-processing-15e34b787480. [accessed: 29-Sep-2021].

[32] Justine Sherry, Shaddi Hasan, and Colin Scott et al. 2012. Making middleboxes someone else's problem: Network processing as a cloud service. *ACM SIGCOMM Computer Communication Review* 42 (2012).

[33] Lin Shi, Hao Chen, and Jianhua Sun et al. 2011. vCUDA: GPU-accelerated high-performance computing in virtual machines. *IEEE Trans. Comput.* 61 (2011).

[34] Liran Sidki, Yehuda Ben-Shimol, and Akiva Sadovski. 2016. Fault tolerant mechanisms for SDN controllers. In *IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*.

[35] Jaewoong Sim, Aniruddha Dasgupta, and Hyesoon Kim et al. 2012. A performance analysis framework for identifying potential benefits in GPGPU applications. In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*.

[36] Weibin Sun and Robert Ricci. 2013. Fast and flexible: Parallel packet processing with GPUs and click. In *Architectures for Networking and Communications Systems*. IEEE.

[37] Yusuke Suzuki, Shinpei Kato, and Hiroshi Yamada et al. 2014. GPUvm: Why Not Virtualizing GPUs at the Hypervisor?. In *USENIX Annual Technical Conference (ATC)*. USENIX Association.

[38] Slavica Tomovic, Nedjeljko Lekic, and Igor Radusinovic et al. 2016. A new approach to dynamic routing in SDN networks. In *18th Mediterranean Electrotechnical Conference (MELECON)*.

[39] Inc. VMware. [n. d.]. *VMware ESXi: The Purpose-Built Bare Metal Hypervisor*. https://www.vmware.com/in/products/esxi-and-esx.html [accessed 26-Nov-2021].

[40] Xiaodong Yi, Jingpu Duan, and Chuan Wu. 2017. GPUNFV: A GPU-Accelerated NFV System *(APNet'17)*. Association for Computing Machinery.

[41] Andrew J Younge, John Paul Walters, and Stephen Crago et al. 2014. Evaluating GPU passthrough in Xen for high performance cloud computing. In *IEEE International Parallel & Distributed Processing Symposium Workshops*. IEEE.

[42] Hangchen Yu and Christopher J Rossbach. 2017. Full virtualization for gpus reconsidered. In *Proceedings of the Annual Workshop on Duplicating, Deconstructing, and Debunking*.

[43] Kai Zhang, Bingsheng He, and Jiayu Hu et al. 2018. G-net: Effective {GPU} sharing in {NFV} systems. In *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*.

[44] Kai Zhang, Jiayu Hu, and Bei Hua. 2015. A holistic approach to build real-time stream processing system with GPU. *J. Parallel and Distrib. Comput.* 83 (2015).

[45] Zhilong Zheng, Jun Bi, and Chen Sun et. al. 2018. GEN: A GPU-Accelerated Elastic Framework for NFV. In *Proceedings of the 2nd Asia-Pacific Workshop on Networking*. ACM.