A GPU-assisted NFV framework for intrusion detection system

A GPU-Assisted NFV Framework for Intrusion Detection System

Igor Araujo[a,], Carlos Natalino[b], Diego Cardoso[a]

[a]*Institute of Technology, Federal University of Pará, Belém, Pará, Brazil*
[b]*Department of Electrical Engineering, Chalmers University of Technology, Gothenburg, Sweden*

Abstract

The network function virtualization (NFV) paradigm advocates the replacement of specific-purpose hardware supporting packet processing by general-purpose ones, reducing costs and bringing more flexibility and agility to the network operation. However, this shift can degrade the network performance due to the non-optimal packet processing capabilities of the general-purpose hardware. Meanwhile, graphics processing units (GPUs) have been deployed in many data centers (DCs) due to their broad use in, e.g., machine learning (ML). These GPUs can be leveraged to accelerate the packet processing capability of virtual network functions (vNFs), but the delay introduced can be an issue for some applications. Our work proposes a framework for packet processing acceleration using GPUs to support vNF execution. We validate the proposed framework with a case study, analyzing the benefits of using GPU to support the execution of an intrusion detection system (IDS) as a vNF and evaluating the traffic intensities where using our framework brings the most benefits. Results show that the throughput of the system increases

Email address: igoraraujo@ufpa.br (Igor Araujo)

Accepted by Computer Communications *DOI:10.1016/j.comcom.2021.01.024*

from 50 Mbps to 1 Gbps by employing our framework while reducing the central process unit (CPU) resource usage by almost 40%. The results indicate that GPUs are a good candidate for accelerating packet processing in vNFs.[1]

Keywords: NFV, CUDA, GPGPU, IDS

1. Introduction

The Cisco Visual Networking Index forecasts growth in global IP traffic, reaching 396 exabytes per month by 2022. It is nearly triple the 122 exabytes recorded in 2017 [1]. The network function virtualization (NFV) has been proposed as a new paradigm to help operators meet these increasing traffic requirements while reducing cost and improving flexibility and agility to their network operations.

NFV implements virtual network functions (vNFs) by decoupling hardware appliances from the functions running on them (firewalls, gateways, and others). In other words, instead of using functions that have hardware and software closely integrated, vNF uses technologies such as virtualization or containerization to run functions over general-purpose hardware [2].

The main benefits of NFV are *(i)* reduced capital expenditure (CAPEX) and operational expenditure (OPEX): the general-purpose equipment can be used across a broad set of applications, also contributing to a more flexible network architecture; *(ii)* shorter time to market: the new functions will be implemented by software, i.e., no longer by a specific hardware appliance;

[1]The complete implementation of the software components reported in this work will be published in open-source format upon paper acceptance.

(iii) reduced complexity of deployment and management: NFV can be orchestrated and managed according to the operator objectives; *(iv)* dynamic and elastic scaling of services: the vNF's resources can be provisioned following the traffic demand; *(v)* efficient usage and management: the NFV allows network functions from different vendors to run in a consolidated hardware platform and manage them in a centralized manner [3].

However, one of the big challenges of shifting specific appliance hardware for general-purpose hardware is the difficulty of reaching, using general-purpose hardware, the same performance level achieved by specific appliances. Nonetheless, general-purpose hardware opens an opportunity to explore different types of hardware, e.g., processing packets using graphics processing units (GPUs).

Recently, GPUs have been applied in many domains other than the video rendering initially intended for them due to their highly parallelized processing capability. This capability, known as general-purpose graphic processing unit (GPGPU), has enabled many recent advances in machine learning (ML). The successful application of GPUs in different areas has driven cloud computing providers to deploy them in data centers (DCs). In networking, GPUs have been applied to IPv4/v6 forwarder [4], IPsec gateway [5], deep packet inspection [6], and cipher algorithm [7]. However, some aspects, such as the delay introduced by GPUs to the packet processing, have not been studied so far.

In this paper, we propose a quasi-real-time framework for the use of GPUs to support vNF execution. The framework defines the key packet processing components that should be developed/adapted to take advantage

of the highly parallelized capabilities of GPUs while efficiently using CPU resources. We present a case study where an open-source intrusion detection system (IDS) (a common and important network function) is modified using the framework. The benefits of the GPU-assisted vNF are assessed in realistic traffic scenarios. Results show that by carefully configuring the execution parameters of the vNF, it is possible to improve the throughput from 50 Mbps to 1 Gbps while reducing the CPU usage by 40%. The results also indicate that the use of GPUs provide significant packet processing speedup and is recommended for most traffic intensities and delay requirements.

The remainder of this paper is organized as follows. Section 2 describes related work on NFV, GPU, and IDS. Section 3 introduces the background concepts to the use of GPGPU. Section 4 presents the details of the proposed GPU-assisted packet processing framework. Section 5 presents the details of the IDS implementation and the experimental setup, results and discussions. Finally, the paper is concluded in Section 6.

2. Related Work

Maintaining a high-performance IDS is critical in high throughput networks due to the increasing complexity of the DPI task and the increase of attack patterns [8]. This performance is directly defined by the hardware used, so different architectures [9] and technologies have been evaluated in order to improve the performance, like TILERAGX36 [10], Intel Software Guard Extension [11], and GPU [12, 13]. Recent works show that using GPU increases the throughput and reduces the CAPEX [14]. The GPUs also have a better performance-per-watt rate and reduced energy consumption [15].

Vasiliadis et al. [16, 12] propose architectures to improve the performance of IDS using GPU. The GPU execution management proposed overlaps one execution at the GPU of a buffer of packets, with the transfer between memories (GPU and CPU) from another buffer of packets. However, they do not perform simultaneous execution of packet batches on the GPU. [12] uses two GPUs to perform these simultaneous executions. Our framework allows the management of simultaneous executions on a single GPU. Also, it enables better usage of GPU resources with fewer CPU resources.

Zheng et al. [17] propose a framework to enforce latency Service Level Objective in GPU-accelerated NFV systems, owing to the fact that by consolidating multiple network functions in one host, current GPU-accelerated NFV systems suffer from significant latency variation for each network function. The authors present three design principles to guarantee latency in this scenario. Our framework uses two of those principles: *(i)* dynamic batch size setting and *(ii)* maximizing concurrency while minimizing interference for task execution.

Lin et al. [6] propose two means of parallel string matching algorithms that adopt perfect hashing to compact a state transition table and reduce memory usage. The authors use the parallel failureless aho-corasick algorithm (PFAC) that relies on a multi-GPU approach to process more than one buffer concurrently, i.e., one buffer for each GPU. Our framework has a single GPU approach with concurrent buffer processing.

Yi et al. [18] present the GPUNFV, a GPU-based NFV system that provides microservice for stateful service chain processing with GPU acceleration. The authors perform experiments with a firewall, load balancer, and

flow monitor as vNFs in their framework. The approach uses a single GPU and processes only one batch at a time. On the contrary, our framework implements a CUDA stream pool, running multiple batches concurrently on the same GPU.

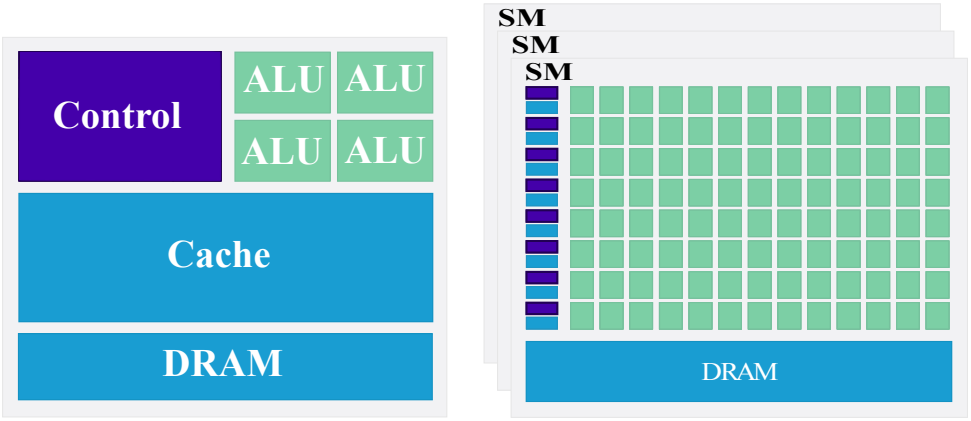
The literature shows that performance is a critical issue to the successful adoption of NFV. Moreover, IDS augments these challenges due to their criticality in maintaining a safe network. Finally, the literature demonstrates that GPUs can improve the throughput of telecommunications applications and reduce energy consumption. However, works using GPUs did not exploit the full potential of the GPU resources. They either use a multi-GPU approach to process more than one buffer at a time or waste CPU resources by locking the thread to wait for the GPU processing to finish. We propose a framework that implements a CUDA stream pool which does not lock the CPU threads to wait for a GPU response and execute more buffers concurrently. This approach translates into a more efficient usage of resources and improved throughput for the vNFs using our framework.

3. General Purpose Graphic Processing Unit

GPUs are commonly used in matrix-multiplication operations [19]. The concept of GPGPU was introduced in 2001 with support to floating-point operations in GPUs as a way to compute anything other than graphic operations. In 2006, NVIDIA released the compute unified device architecture (CUDA), enabling code execution on GPUs without requiring full and explicit conversion of the data to/from a graphical form [20]. This architecture is the main enabler of the recent advances in several areas, such as the

training of large-scale ML models.

The GPU architecture is composed of many streaming multiprocessors (SMs); each SM has many single processors. The focus of the GPU architecture is on the number of arithmetic logic units (ALUs) to increase the throughput. In contrast, central process units (CPUs) focus on a large part of the chip to memory cache, reducing the memory access latency.



(a) CPU (b) GPU

Figure 1: CPU and GPU architectures [21].

Figure 1 illustrates the differences between CPU and GPU architectures. CPUs (Figure 1a) focus on executing several different instructions over different data and reserve a significant space of chip to the memory cache to accelerate the access to the data. A large cache memory reduces the latency of memory access. GPUs (Figure 1b), in contrast, focus on executing the same instructions over multiple instances of different data and reserve more space of the chip to ALUs, which increases the throughput of computations. The fact that GPUs have more ALUs than CPUs makes the GPU ideal for large amounts of data executing the same instructions over differ-

ent values. Due to these properties, GPUs are present in almost all top 10 energy-efficient supercomputers, and many works show that GPUs present better power efficiency.

One of the significant disadvantages of GPGPUs is the need to transfer data between the random access memory (RAM) (which communicates directly with the CPU) and the GPU memory. These data transfers introduce an overhead before start processing on a GPU. Therefore, the benefits obtained by increased throughput may not compensate for the overhead of transferring the data depending on the amount of data to be processed. Consequently, the time to process small amounts of data on GPUs will be longer than on CPUs. This is particularly important for the packet processing capabilities of vNFs, since the amount of data to be processed may define whether the use of GPGPUs is beneficial or not.

3.1. CUDA stream

The management of tasks to be performed by a CUDA program, such as the GPU execution and CPU-GPU inter-memory transfer, is made by a queue of operations called stream, which follows a first-in, first-out (FIFO) pattern. If the CUDA stream in which the application execution to be scheduled is not specified, it will be allocated to a default stream. In this case, each operation will be executed in parallel with many CUDA cores but performed sequentially, thus leading to a reduced performance if the program is composed of inherently independent operations. A multiple stream approach can be used to avoid this problem by performing cross-device operations or concurrent GPU executions on a single device [22].

Figure 2 illustrates the difference between these approaches by showing

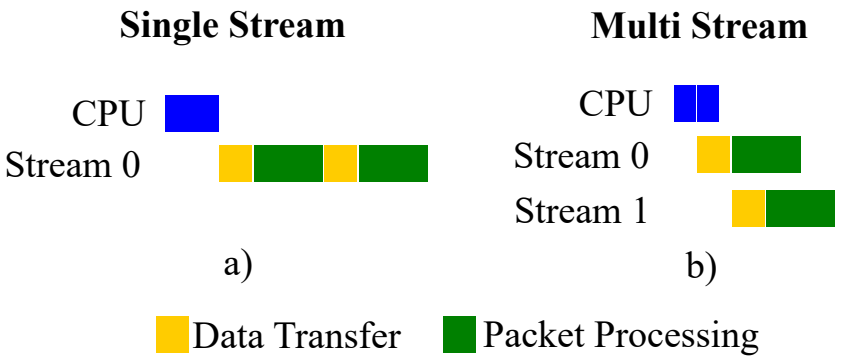


Figure 2: A CUDA application with (a) single- and (b) multi-stream approach.

an example of a CUDA application running in single- and multi-stream approaches for two tasks (i.e., packet buffers in the context of this work). For simplicity, each execution is constituted by two primary sequential operations (i.e., data transfer and packet processing). The execution in a single stream is represented in Figure 2a, where the data transfer of the first buffer is performed and followed by the GPU processing; only after the complete execution of the first buffer, the operations for the second buffer are initiated. Figure 2b illustrates the same operations, but using a multi-stream approach, where each buffer was allocated in a different stream so that the CUDA operations can run concurrently; here, as the operations can be executed independently, it is not necessary for one to finish for the other to start. In the following, we develop a framework that leverages this multi-stream approach to accelerate packet processing in vNFs.

4. GPU-Assisted Packet Processing Framework

Figure 3 shows the sequence of steps that a network packet is subject when being processed by an implementation of the framework proposed in

this work. The packets received by a network interface (1) are queued (2) and wait to be processed by an idle CPU thread from the thread pool. In order to prevent a bottleneck in the packets collected, new packets are dropped when this queue (2) reaches a size limit.

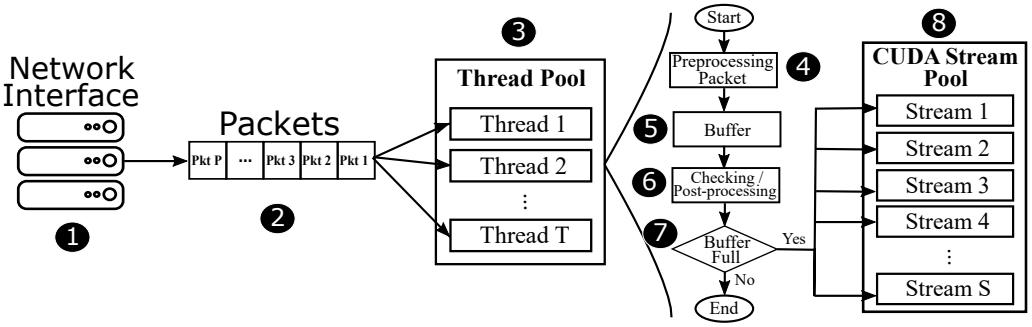


Figure 3: Workflow of the proposed GPU-assisted packet processing framework

The packets from the queue are processed by a pool of threads (3). The pool can be composed of one or many threads, enabling the processing of multiple packets from the queue at the same time. The packets processed by a CPU thread are first subject to a preprocessing step (4) that can perform operations such as packet decoding, protocol identification, handling unprintable bytes, etc. After the preprocessing, the packet is included into the buffer (5). The buffer is used to minimize the number of memory transfer requests between CPU and GPU by enabling packets to be processed by the GPU in batches.

After adding the packet into the buffer, the CPU thread checks whether any of the (previously launched) CUDA streams has finalized its processing (6). This step is necessary because our CUDA stream pool approach results in a nonblocking CPU thread. As a side effect of this approach, we need to check periodically whether the stream has finalized its processing or not. If a

CUDA stream finalized its processing, CPU thread performs the data transfer from the GPU back to the CPU memory. Once the data is transferred to the CPU memory, the CPU thread performs post-processing actions that can perform operations such as packet forwarding, logging, blocking packets, etc.

Finally, the CPU thread checks whether it should start the processing of the buffer in an idle CUDA stream (7). To do this, two conditions must be fulfilled: (i) the buffer is full or its time limit was exceeded; and (ii) there is a CUDA stream out of the pool which is currently idle. If these two conditions are fulfilled, the buffer is transferred to the GPU memory and a CUDA stream is assigned to process this buffer (8). Within the CUDA stream, each packet is processed by one CUDA core, while all the CUDA cores execute the same set of instructions over all the packets. This is a key aspect since for a vNF all the packets are always subject to the same set of operations. As mentioned before, since our framework uses CUDA streams that do not block the CPU, at this point the CPU is free and can start again the processing of the next packet in the queue.

5. Case Study and Performance Assessment

The framework proposed in Sec. 4 is validated by implementing an IDS and assessing its performance. The case study was implemented in C++[2], based on an open-source IDS called Snort [23]. To assess the benefits of using the proposed framework, we compare the performance of the framework with a IDS using only CPUs for the packet processing. Figure 4 further details the

[2]The implementation will publicly available upon publication at: `https://git.io/fjnIx`

architecture introduced in Figure 3 for the specific IDS case study, showing the workflow of the CPU-only and GPU-assisted executions, in addition to the network setup used to assess the performance of the proposed framework.

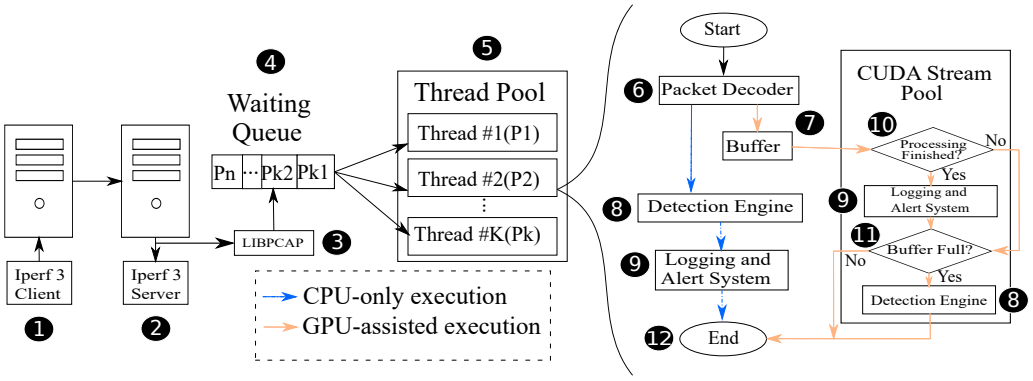


Figure 4: IDS architecture and case study execution flow for CPU-only and GPU-assisted executions.

The case study works as follows. Network traffic is generated from the client (1) to the server (2). Arriving packets are captured as a copy by the sniffer method using libpcap (3) and stored in the waiting queue (4). The waiting queue has limited capacity, and can be fully occupied if the processing capabilities of the IDS are not enough to cope with the traffic intensity. When this happens, new incoming packets cannot be accommodated in the queue, and will be dropped.

The IDS uses a CPU multi-threading approach, where one thread is used specifically to capture packets from the network, while the other threads compound a thread pool (5) that will perform the deep packet inspection (DPI). During DPI, the first operation is to decoded the packet (6), identifying its protocol.

If the IDS runs in CPU-only mode, the execution flow will start the detec-

tion engine. The detection engine (8) performs a string match algorithm, i.e., Aho-Corasick, to identify if the content from packet data is a known attack pattern. The logging and alert system triggers the appropriate measures if threats were found in the packet (9).

If the IDS runs in GPU-assisted mode, the decoded packet is added into the buffer (7). Then, the CPU thread checks whether a (previously launched) CUDA stream has finalized its processing (10). In case the processing has finished, the result is transferred from the GPU to the CPU memory, and appropriate logging and alert systems (9) are notified. Then, the CPU thread checks whether the buffer is full or not and whether there is an idle CUDA stream available (11). If both conditions are met, the CPU launches a CUDA stream to run the detection engine (8) over the current buffer. At this point, this CPU thread is free and will repeat the procedures starting from (6).

5.1. Setup

This section first presents the setup used to evaluate the performance of the GPU-assisted implementation of the IDS. Then, the performance assessment is presented.

The experiments were executed using two computers, one client and one server. The client generates the traffic using iPerf3 and is equipped with Intel(R) Core(TM) i7-6700 CPU @ 3.40GHz, 16 GB RAM, Linux CentOS 7, and 1 Gbps PCI Express Gigabit Ethernet Controller. The server runs the developed IDS and receives the traffic. It is also equipped with Intel(R) Xeon(R) CPU E5-2620 v2 @ 2.10GHz, 16 GB RAM, Linux CentOS 7, 1 Gbps Network Interface Card, and an NVIDIA Tesla K20c with 2496 CUDA Cores and 5GB of VRAM. The two machines are connected directly by a

gigabit Ethernet cable. We evaluate the performance of the CPU-only and GPU-assisted implementations of the IDS over different traffic intensities and configurations. Table 1 shows the parameters considered for the experiments.

Table 1: Parameters of the Experiments

Traffic duration	1 hour
Num. Threads[a]	4 threads
Waiting Queue limit	128 MB
Buffer Time Limit	500 ms
Num. CUDA streams	16 streams
GPU Buffer Sizes	256KB 512KB 1MB 2MB 4MB 8MB 16MB 32MB
Traffic Intensities	10Mbps 50Mbps 100Mbps 200Mbps 400Mbps 800Mbps 1Gbps

[a] Including the packet capture exclusive thread.

All the experiments have one-hour traffic duration, simulating a machine with four cores, one of them used exclusively for the packet capture and 3 to the DPI task. The waiting queue has a limit of 128 MB, i.e., if the packets overextend this limit, new incoming packets are dropped. A batch of packets is processed when either of the following conditions is met: *(i)* the buffer time reached 500 ms; or *(ii)* the buffer has reached its size limit. 16 CUDA streams have been used to better utilize the GPU resources.

In order to evaluate the packet delay incurred by the GPU buffer, eight buffer size configurations were tested in seven different network traffic intensities. Moreover, for each traffic intensity, a CPU-only execution was per-

formed, resulting in a combination of 63 experiments in total. The results of these experiments are reported next.

5.2. Results

We evaluate the performance of the CPU-only and GPU-assisted IDS over a set of metrics: *(i)* packet loss: the percentage of packets lost due to the dropping of packets resulting from a full waiting queue; *(ii)* CPU usage: the percentage of CPU used to execute the IDS; *(iii)* GPU usage: the percentage of the GPU used to execute the DPI; *(iv)* speedup: the relative performance of GPU-assisted over the CPU-only execution in terms of packet processing time.

The packet loss illustrates the capability of the IDS to process the incoming packets at the necessary rate, and its analysis can also show us the throughput achieved by the IDS. The CPU usage is another important metric that illustrates the efficiency of the IDS, directly related to its CAPEX. Moreover, a CPU-bottlenecked vNF may have unexpected behavior. The GPU usage allows us to evaluate the performance of the CUDA stream pool approach. The speedup shows how beneficial is the GPU-assisted over the CPU-only execution in terms of the delay incurred by the vNF. We then show a table with recommended configurations according to the traffic intensity and the application delay requirement.

Figure 5 shows the packet loss percentage over different traffic intensities for the CPU-only and different buffer size limits of the GPU-assisted IDS. The CPU-only implementation starts to drop packets at 100 Mbps, where it drops around 40% of the packets. With 1 Gbps traffic, the CPU-only implementation drops almost 95% of the packets. The GPU-assisted IDS

presents substantially better results if the buffer size is correctly configured. If the buffer size is 4 MB or higher, there are no packets dropped. The packet drops observed in smaller buffer sizes occur due to a higher number of CPU-GPU calls, degrading the performance of the GPU-assisted implementation at high bit rates. However, larger buffers may lead to long delays under low traffic intensities. This indicates that our framework can benefit from a dynamic adjustment of its parameters to better match the needs for a particular traffic intensity.

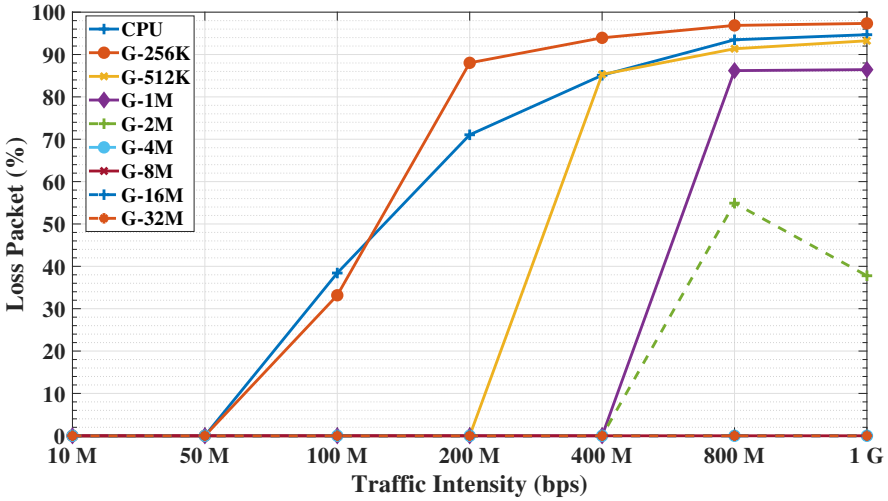


Figure 5: Packets loss of the IDS with CPU and GPU executions for different traffic intensities. The GPU results (G) were tested for different buffer size limits.

Figure 6 shows the throughput of the CPU-only and the GPU-assisted versions with different buffer sizes for 1 Gbps traffic intensity. The 1 Gbps traffic intensity is relevant because it is the maximum theoretical throughput for the network interface card used in our experiments. Due to protocol overhead and other factors, 900 Mbps is the maximum rate that we can

achieve. Our framework achieves the throughput of approximately 900 Mbps for the GPU-assisted versions with a buffer greater than or equal to 4MB. As we saw in Figure 5, these are the configurations where no packet drop is experienced, and shows that our framework is able to achieve the maximum practical throughput of our experimental setup. As opposed to the GPU-assisted IDS, the CPU-only IDS only achieves a throughput of around 50 Mbps, which means that the GPU-assisted version improves throughput by around 16 times.

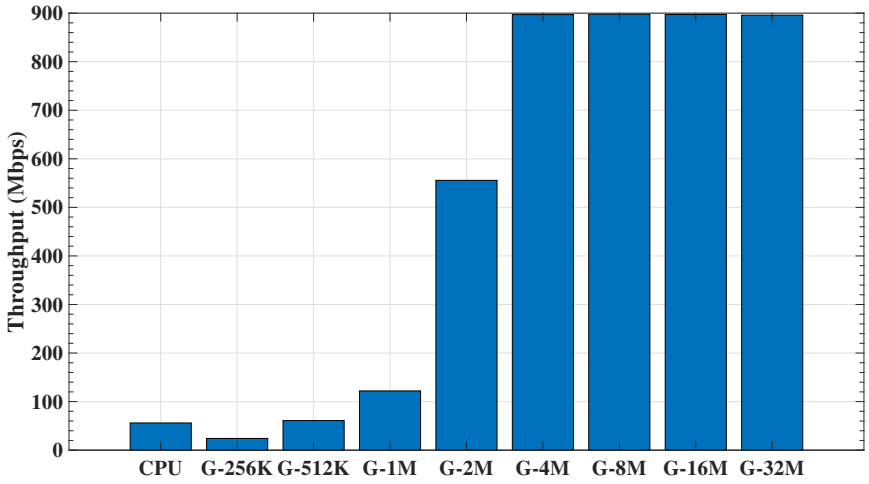


Figure 6: Throughput of the IDS with CPU and GPU executions for the 1 Gbps traffic intensity. The GPU results (G) were tested for different buffer size limits.

Figure 7 shows the CPU usage over different framework configurations for different traffic intensities. The CPU-only IDS presents a CPU usage between 65% to 78% across all traffic intensities tested. On the other hand, the GPU-assisted IDS uses only 1% to 42% of CPU resources, showing a reduction of at least 46% in the CPU usage compared with the CPU-only

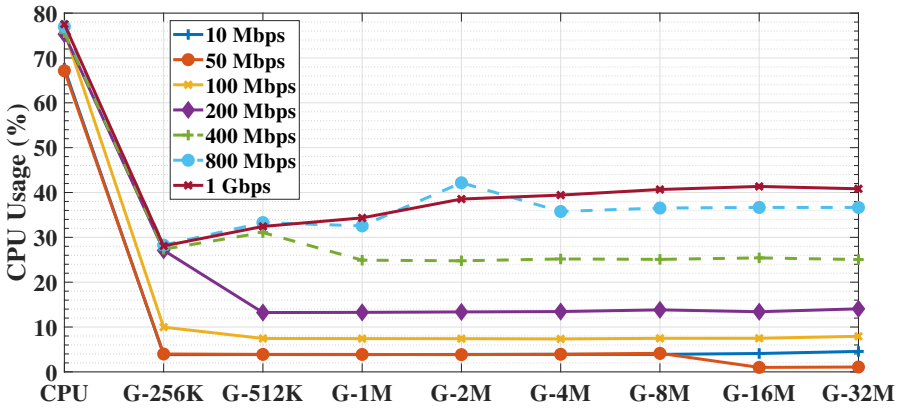


Figure 7: CPU usage for different traffic intensities using the CPU-only and GPU-assisted (G) IDS with different buffer sizes.

IDS. Another point worth mentioning is that even in low traffic intensities, i.e., 10 and 50 Mbps, where the CPU-only version has not dropped packets, the decrease in CPU usage ranges from 32% to 95%, approximately.

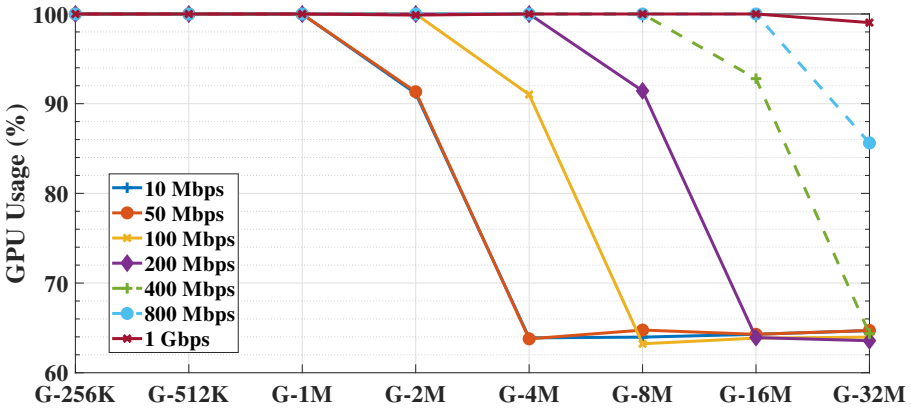


Figure 8: GPU usage over different buffer size limits of the GPU-assisted for different traffic intensities.

Figure 8 shows the GPU usage over different buffer size limits for different traffic intensities. The buffer size limit has a strong impact on GPU usage. A

smaller buffer size penalizes the GPU usage due to the more frequent CPU-GPU memory transfers, which reduces the GPU efficiency. Our approach, when appropriately configured, can use around 64% of the GPU resources. This value represents a significant improvement over previous works, which report a GPU usage of around 20% (in this case, the more we can use the GPU, the better). Moreover, with the highest traffic tested (1 Gbps), the GPU usage reaches near 100% for the biggest buffer size. This trend indicates that the framework can be potentially used for higher bit rates combined with higher-performance GPUs.

Table 2: Speedup of the Packet Processing Time (Delay) of the GPU-assisted over the CPU-only IDS. Values in bold font highlight the cases where both CPU-only and GPU-assisted IDS dropped packets.

Buffer	**Traffic Intensity (Mbps)**						
Size (B)	**10**	**50**	**100**	**200**	**400**	**800**	**1000**
256K	0.27	1.29	**2.60**	**0.40**	**0.39**	**0.43**	**0.43**
512K	0.14	1.03	551.90	811.14	**0.94**	**1.18**	**1.09**
1M	0.08	0.62	375.37	651.16	1023.40	**1.88**	**2.19**
2M	0.08	0.33	208.11	388.74	687.53	6.91	18.11
4M	0.08	0.19	107.66	206.29	380.90	794.35	930.03
8M	0.08	0.20	62.74	105.21	197.24	419.72	494.75
16M	0.08	0.20	62.86	62.16	99.90	214.41	252.58
32M	0.08	0.20	63.23	61.95	60.64	107.92	127.07

Table 2 shows the speedup of the average packet processing time (delay) of the GPU-assisted over the CPU-only IDS. In our case, a number greater than

one means that the GPU-assisted packet processing is faster than the CPU-only one. The table shows that the buffer size definition greatly impacts the speedup achieved. For instance, with 400 Mbps of traffic, the GPU-assisted IDS can achieve up to 1023 times faster packet processing when a buffer of 1 MB is used. However, using half of this buffer size (i.e., 512 KB) yields worse results than the CPU-only execution (i.e., speedup of 0.94). These results indicate that GPUs are more beneficial for higher-intensity traffic scenarios, but even with traffic as low as 50 Mbps, benefits can still be obtained.

The results shown so far indicate that GPU-assisted vNFs can achieve significant benefits over CPU-only ones. However, different applications may have different delay requirements. Table 3 illustrates how the recommended framework configuration may change depending on the traffic intensity and delay requirement. We assume applications with 50, 100, 250, and 500 ms, assuming that applications with lower delay might be served by resources closer to the edge of the network. The recommended configuration is based on the setting that provides a delay lower than required while favoring the ones with lower dropped packets.

Table 3 shows that the GPU-assisted configuration achieved better performance in almost all the cases, except for the 10 Mbps traffic intensity. The lower performance of the GPU-assisted framework under low traffic intensities is due to the memory transfer overhead, which overcomes the benefits of the GPU multi-processing capabilities. Moreover, applications with higher delay allow for larger buffer sizes, which translates to higher speedups shown in Table 2.

Table 3: Recommended Configuration Given the Traffic Intensity and Application's Delay Requirement.

Traffic Intensity	**Delay Requirement**			
	50 ms	**100 ms**	**250 ms**	**500 ms**
10 Mbps	CPU	CPU	CPU	GPU (2MB)
50 Mbps	GPU (256KB)	GPU (1MB)	GPU (2MB)	GPU (4MB)
100 Mbps	GPU (1MB)	GPU (2MB)	GPU (4MB)	GPU (8MB)
200 Mbps	GPU (2MB)	GPU (4MB)	GPU (8MB)	GPU (16MB)
400 Mbps	GPU (4MB)	GPU (8MB)	GPU (16MB)	GPU (32MB)
800 Mbps	GPU (8MB)	GPU (16MB)	GPU (32MB)	GPU (32MB)
1 Gbps	GPU (8MB)	GPU (16MB)	GPU (32MB)	GPU (32MB)

6. Conclusion

In this paper, we introduced a framework for the use of GPUs to support the packet processing tasks of vNFs. The framework was validated by a case study where we modified a state-of-the-art open-source IDS to incorporate the capabilities of the proposed framework. The results obtained from the case study show the benefits of using the proposed framework over several different performance indicators, such as throughput, delay, and resource usage. Finally, the paper presented a table that shows the recommended framework configuration for different traffic intensities and application delay requirements.

We demonstrated that the benefits of using GPUs to network packet processing depend on the traffic intensity and buffer size. There are cases where the traffic intensity or the buffer size are low, and the GPU execution will in-

cur performance degradation in comparison with CPU execution. Therefore, ensuring the best performance across different traffic scenarios is an important challenge to fully utilize the potential of GPUs for packet processing. As future work, a hybrid CPU-GPU IDS inspection can be developed, where an intelligent strategy (possibly enabled by machine learning) selects which processing entity and buffer size should be used given the current traffic properties and application requirements such as delay. On a different aspect, new technologies such as Remote Direct Memory Access (RDMA) allows the GPU to access packets directly from the Network Interface Card (NIC), not requiring the packet to be accessed by the CPU first, and then transferred to the GPU later. In the future, such technologies can be used to mitigate the performance degradation caused by CPU-GPU memory transfer.

Acknowledgment

This work was supported by CNPq, National Council for Scientific and Technological Development - Brazil. The Tesla K20c used for this research was donated by NVIDIA Corporation's GPU Grant Program. This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001

References

[1] C. VNI, Cisco Visual Networking Index: Forecast and Trends, 2017–2022, Technical Report, Cisco, 2019.

[2] A. M. Alwakeel, A. K. Alnaim, E. B. Fernandez, A Survey of Network

Function Virtualization Security, in: SoutheastCon 2018, IEEE, 2018, pp. 1–8. DOI: 10.1109/secon.2018.8479121.

[3] M. Pattaranantakul, R. He, Q. Song, Z. Zhang, A. Meddahi, NFV Security Survey: From Use Case Driven Threat Analysis to State-of-the-Art Countermeasures, IEEE Communications Surveys Tutorials 20 (2018) 3330–3368. DOI: 10.1109/COMST.2018.2859449.

[4] J. Kim, K. Jang, K. Lee, S. Ma, J. Shim, S. Moon, NBA (Network Balancing Act): A High-Performance Packet Processing Framework for Heterogeneous Processors, in: European Conference on Computer Systems, EuroSys 15, Association for Computing Machinery, Bordeaux, France, 2015, pp. 1–14. DOI: 10.1145/2741948.2741969.

[5] Y. Hu, T. Li, Enabling Efficient Network Service Function Chain Deployment on Heterogeneous Server Platform, in: 2018 IEEE International Symposium on High Performance Computer Architecture (HPCA), pp. 27–39. DOI: 10.1109/HPCA.2018.00013.

[6] C. Lin, J. Li, C. Liu, S. Chang, Perfect Hashing Based Parallel Algorithms for Multiple String Matching on Graphic Processing Units, IEEE Transactions on Parallel and Distributed Systems 28 (2017) 2639–2650. DOI: 10.1109/TPDS.2017.2674664.

[7] T. Suzuki, S. Kim, J. Kani, K. Suzuki, A. Otaka, T. Hanawa, Parallelization of cipher algorithm on cpu/gpu for real-time software-defined access network, in: 2015 Asia-Pacific Signal and Information Processing

Association Annual Summit and Conference (APSIPA), pp. 484–487. DOI: 10.1109/APSIPA.2015.7415318.

[8] W. Bul'ajoul, A. James, M. Pannu, Improving network intrusion detection system performance through quality of service configuration and parallel technology, Journal of Computer and System Sciences 81 (2015) 981 – 999. DOI:10.1016/j.jcss.2014.12.012.

[9] W. Bulajoul, A. James, S. Shaikh, A new architecture for network intrusion detection and prevention, IEEE Access 7 (2019) 18558–18573. DOI: 10.1109/access.2019.2895898.

[10] H. Jiang, G. Zhang, G. Xie, K. Salamatian, L. Mathy, Scalable high-performance parallel design for Network Intrusion Detection Systems on many-core processors, in: Architectures for Networking and Communications Systems, pp. 137–146. DOI:10.1109/ANCS.2013.6665196.

[11] D. Kuvaiskii, S. Chakrabarti, M. Vij, Snort Intrusion Detection System with Intel Software Guard Extension (Intel SGX), arXiv e-prints (2018). DOI:arxiv:1802.00508.

[12] G. Vasiliadis, M. Polychronakis, S. Ioannidis, MIDeA: A Multi-Parallel Intrusion Detection Architecture, in: Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS 11, Association for Computing Machinery, New York, NY, USA, 2011, p. 297308. DOI:10.1145/2046707.2046741.

[13] M. A. Jamshed, J. Lee, S. Moon, I. Yun, D. Kim, S. Lee, Y. Yi, K. Park, Kargus: A Highly-Scalable Software-Based Intrusion Detection System,

in: Proceedings of the 2012 ACM Conference on Computer and Communications Security, CCS 12, Association for Computing Machinery, New York, NY, USA, 2012, p. 317328. DOI:10.1145/2382196.2382232.

[14] Y. Go, M. A. Jamshed, Y. Moon, C. Hwang, K. Park, APUNet: Revitalizing GPU as Packet Processing Accelerator, in: 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17), USENIX Association, Boston, MA, 2017, pp. 83–96.

[15] C. Stylianopoulos, L. Johansson, O. Olsson, M. Almgren, CLort: High Throughput and Low Energy Network Intrusion Detection on IoT Devices with Embedded GPUs, in: N. Gruschka (Ed.), Secure IT Systems, Springer International Publishing, 2018, pp. 187–202. DOI: 10.1007/978-3-030-03638-6_12.

[16] G. Vasiliadis, S. Antonatos, M. Polychronakis, E. P. Markatos, S. Ioannidis, Gnort: High Performance Network Intrusion Detection Using Graphics Processors, in: Recent Advances in Intrusion Detection, Springer Berlin Heidelberg, Berlin, Heidelberg, 2008, pp. 116–134. DOI: 10.1007/978-3-540-87403-4_7.

[17] Z. Zheng, J. Bi, H. Wang, C. Sun, H. Yu, H. Hu, K. Gao, J. Wu, Grus: Enabling Latency SLOs for GPU-Accelerated NFV Systems, in: 2018 IEEE 26th International Conference on Network Protocols (ICNP), pp. 154–164. DOI: 10.1109/ICNP.2018.00025.

[18] X. Yi, J. Duan, C. Wu, GPUNFV: A GPU-Accelerated NFV System, in: Proceedings of the First Asia-Pacific Workshop on Networking, AP-

Net17, Association for Computing Machinery, Hong Kong, China, 2017, pp. 85–91. DOI: 10.1145/3106989.3106990.

[19] P. Du, R. Weber, P. Luszczek, S. Tomov, G. Peterson, J. Dongarra, From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming, Parallel Computing 38 (2012) 391 – 407. DOI: 10.1016/j.parco.2011.10.002.

[20] W.-S. Lai, C.-C. Wu, L.-F. Lai, M.-C. Sie, Two-Phase PFAC Algorithm for Multiple Patterns Matching on CUDA GPUs, Electronics 8 (2019) 270. DOI: 10.3390/electronics8030270.

[21] A. Herten, GPU-based Online Track Reconstruction for PANDAand Application to the Analysis of D→K$\pi\pi$, Dr., Ruhr-Universitt Bochum, Bochum, 2015. DOI: FZJ-2015-05760.

[22] R. Farber, Chapter 7 - Techniques to Increase Parallelism, in: CUDA Application Design and Development, Morgan Kaufmann, Boston, 2011, pp. 157 – 177. DOI: 10.1016/B978-0-12-388426-8.00007-0.

[23] A. R. Baker, J. Esler, Chapter 2 - Introducing Snort 2.6, in: Snort Intrusion Detection and Prevention Toolkit, Syngress, Rockland, 2007, pp. 31–67. DOI: 10.1016/B978-159749099-3/50007-0.