



A GPU-assisted NFV framework for intrusion detection system

Downloaded from: <https://research.chalmers.se>, 2026-04-04 14:20 UTC

Citation for the original published paper (version of record):

Araujo, I., Natalino Da Silva, C., Cardoso, D. (2021). A GPU-assisted NFV framework for intrusion detection system. *Computer Communications*, 169: 92-98.

<http://dx.doi.org/10.1016/j.comcom.2021.01.024>

N.B. When citing this work, cite the original published paper.

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)

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;

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

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

24 However, one of the big challenges of shifting specific appliance hardware
25 for general-purpose hardware is the difficulty of reaching, using general-
26 purpose hardware, the same performance level achieved by specific appli-
27 ances. Nonetheless, general-purpose hardware opens an opportunity to ex-
28 plore different types of hardware, e.g., processing packets using graphics pro-
29 cessing units (GPUs).

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

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

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

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

58 **2. Related Work**

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

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

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

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

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

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

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

108 **3. General Purpose Graphic Processing Unit**

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

116 training of large-scale ML models.

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

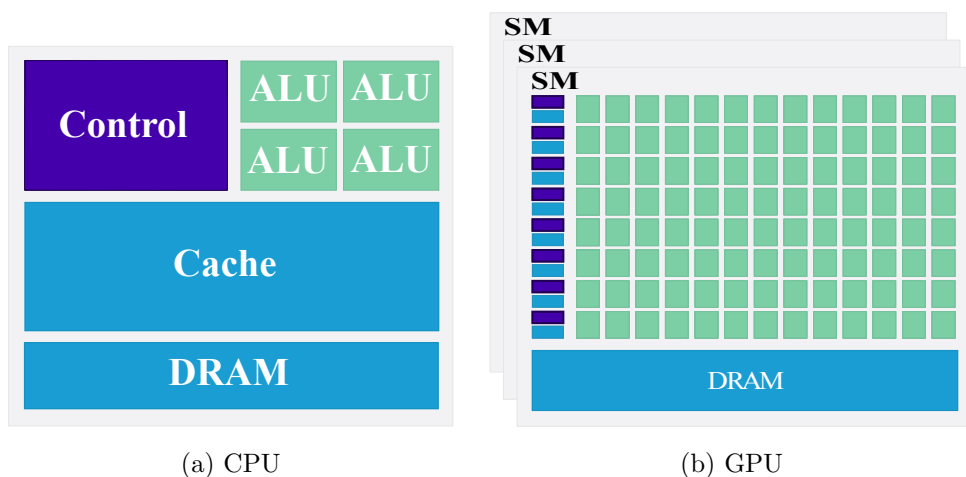


Figure 1: CPU and GPU architectures [21].

122 Figure 1 illustrates the differences between CPU and GPU architectures.
123 CPUs (Figure 1a) focus on executing several different instructions over dif-
124 ferent data and reserve a significant space of chip to the memory cache to
125 accelerate the access to the data. A large cache memory reduces the la-
126 tency of memory access. GPUs (Figure 1b), in contrast, focus on executing
127 the same instructions over multiple instances of different data and reserve
128 more space of the chip to ALUs, which increases the throughput of compu-
129 tations. The fact that GPUs have more ALUs than CPUs makes the GPU
130 ideal for large amounts of data executing the same instructions over differ-

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

134 One of the significant disadvantages of GPGPUs is the need to trans-
135 fer data between the random access memory (RAM) (which communicates
136 directly with the CPU) and the GPU memory. These data transfers intro-
137 duce an overhead before start processing on a GPU. Therefore, the benefits
138 obtained by increased throughput may not compensate for the overhead of
139 transferring the data depending on the amount of data to be processed. Con-
140 sequently, the time to process small amounts of data on GPUs will be longer
141 than on CPUs. This is particularly important for the packet processing capa-
142 bilities of vNFs, since the amount of data to be processed may define whether
143 the use of GPGPUs is beneficial or not.

144 3.1. *CUDA stream*

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

155 Figure 2 illustrates the difference between these approaches by showing

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

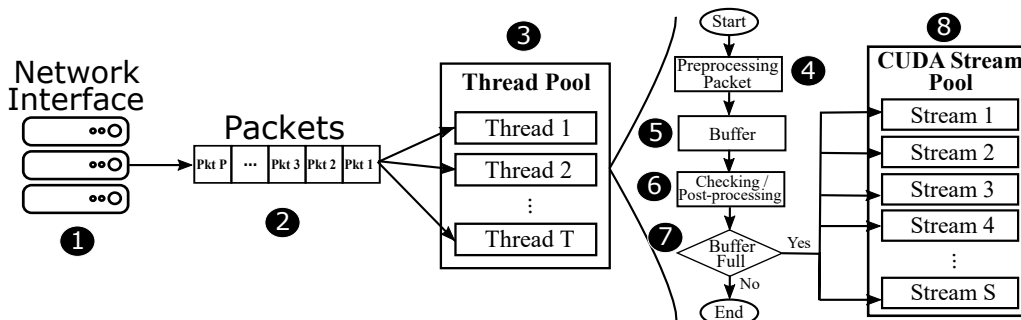


Figure 3: Workflow of the proposed GPU-assisted packet processing framework

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

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

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

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

206 5. Case Study and Performance Assessment

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

²The implementation will publicly available upon publication at: <https://git.io/fjnIx>

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

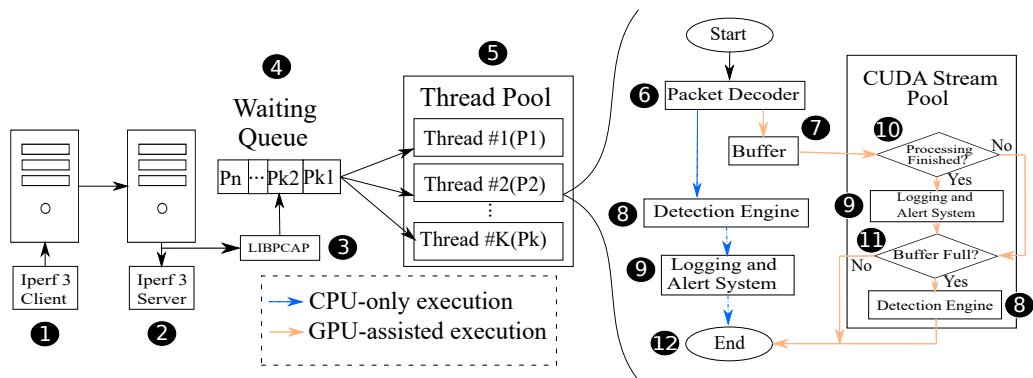


Figure 4: IDS architecture and case study execution flow for CPU-only and GPU-assisted executions.

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

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

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

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

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

241 *5.1. Setup*

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

245 The experiments were executed using two computers, one client and one
246 server. The client generates the traffic using iPerf3 and is equipped with
247 Intel(R) Core(TM) i7-6700 CPU @ 3.40GHz, 16 GB RAM, Linux CentOS
248 7, and 1 Gbps PCI Express Gigabit Ethernet Controller. The server runs
249 the developed IDS and receives the traffic. It is also equipped with Intel(R)
250 Xeon(R) CPU E5-2620 v2 @ 2.10GHz, 16 GB RAM, Linux CentOS 7, 1
251 Gbps Network Interface Card, and an NVIDIA Tesla K20c with 2496 CUDA
252 Cores and 5GB of VRAM. The two machines are connected directly by a

253 gigabit Ethernet cable. We evaluate the performance of the CPU-only and
 254 GPU-assisted implementations of the IDS over different traffic intensities and
 255 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.

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

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

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

268 5.2. Results

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

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

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

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

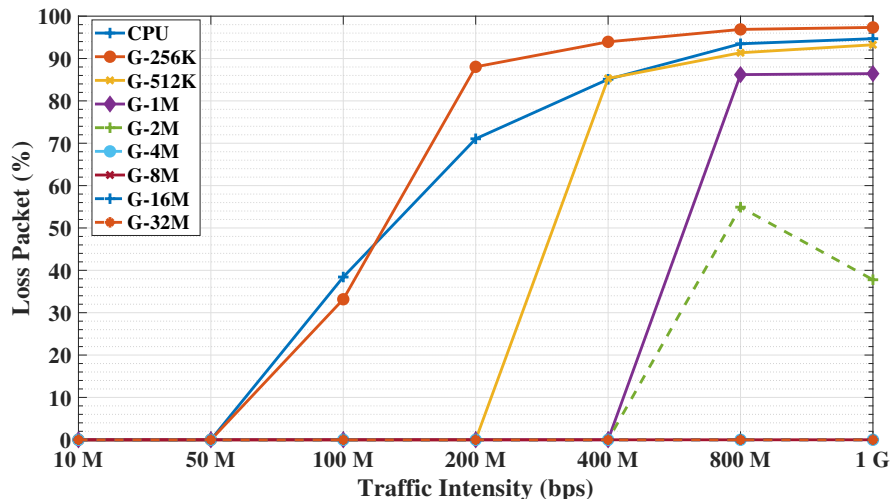


Figure 5: Packets loss of the IDS with CPU and GPU executions for different traffic intensities. The GPU results (G) were tested for different buffer size limits.

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

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

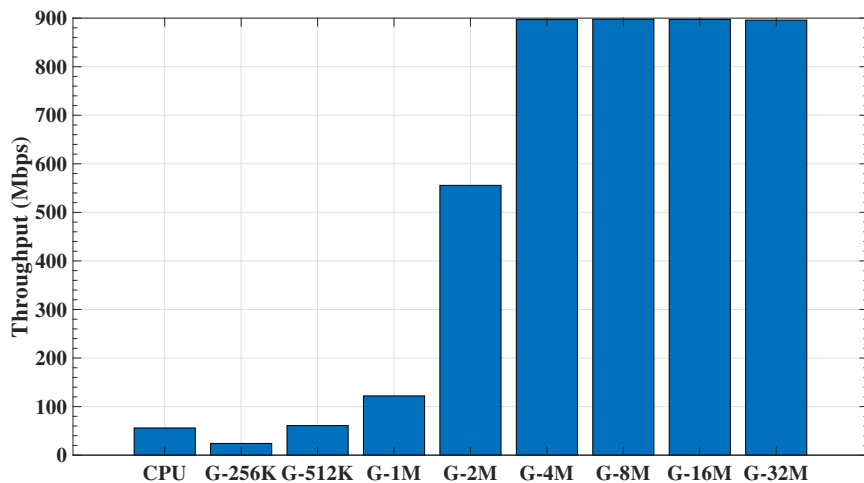


Figure 6: Throughput of the IDS with CPU and GPU executions for the 1 Gbps traffic intensity. The GPU results (G) were tested for different buffer size limits.

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

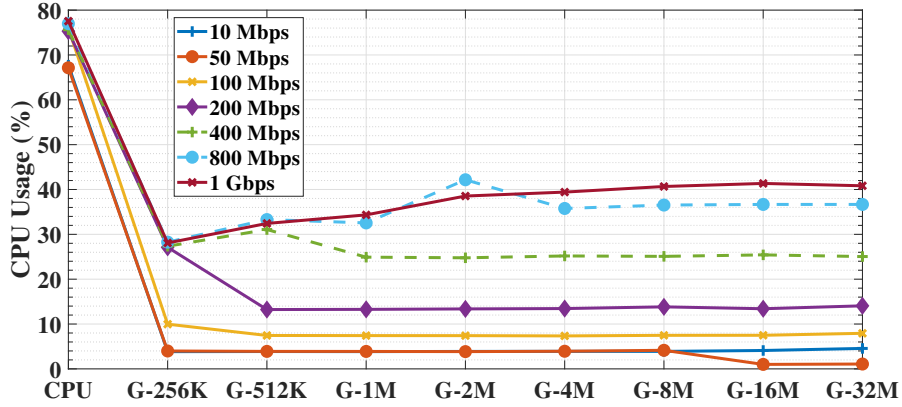


Figure 7: CPU usage for different traffic intensities using the CPU-only and GPU-assisted (G) IDS with different buffer sizes.

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

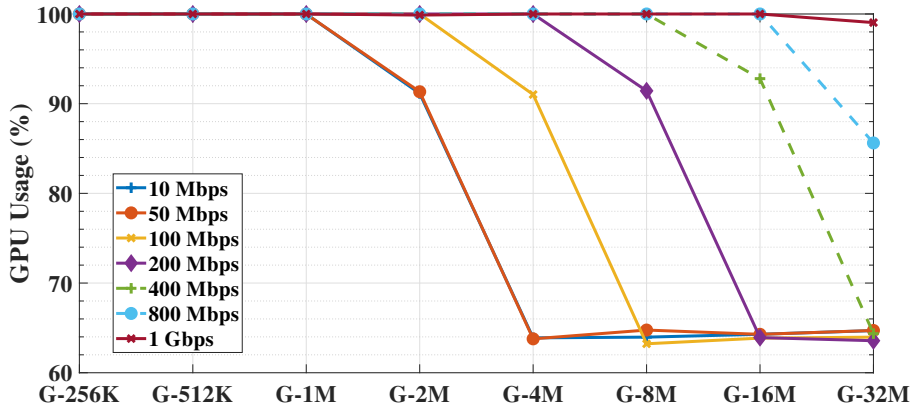


Figure 8: GPU usage over different buffer size limits of the GPU-assisted for different traffic intensities.

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

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

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

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

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

350 Table 3 shows that the GPU-assisted configuration achieved better per-
351 formance in almost all the cases, except for the 10 Mbps traffic intensity. The
352 lower performance of the GPU-assisted framework under low traffic intensi-
353 ties is due to the memory transfer overhead, which overcomes the benefits of
354 the GPU multi-processing capabilities. Moreover, applications with higher
355 delay allow for larger buffer sizes, which translates to higher speedups shown
356 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)

357 6. Conclusion

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

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

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

382 **Acknowledgment**

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

388 **References**

- 389 [1] C. VNI, Cisco Visual Networking Index: Forecast and Trends, 2017–
390 2022, Technical Report, Cisco, 2019.
- 391 [2] A. M. Alwakeel, A. K. Alnaim, E. B. Fernandez, A Survey of Network

- 392 Function Virtualization Security, in: SoutheastCon 2018, IEEE, 2018,
393 pp. 1–8. DOI: 10.1109/secon.2018.8479121.
- 394 [3] M. Pattaranantakul, R. He, Q. Song, Z. Zhang, A. Meddahi, NFV
395 Security Survey: From Use Case Driven Threat Analysis to State-of-
396 the-Art Countermeasures, IEEE Communications Surveys Tutorials 20
397 (2018) 3330–3368. DOI: 10.1109/COMST.2018.2859449.
- 398 [4] J. Kim, K. Jang, K. Lee, S. Ma, J. Shim, S. Moon, NBA (Network
399 Balancing Act): A High-Performance Packet Processing Framework for
400 Heterogeneous Processors, in: European Conference on Computer Sys-
401 tems, EuroSys 15, Association for Computing Machinery, Bordeaux,
402 France, 2015, pp. 1–14. DOI: 10.1145/2741948.2741969.
- 403 [5] Y. Hu, T. Li, Enabling Efficient Network Service Function Chain
404 Deployment on Heterogeneous Server Platform, in: 2018 IEEE In-
405 ternational Symposium on High Performance Computer Architecture
406 (HPCA), pp. 27–39. DOI: 10.1109/HPCA.2018.00013.
- 407 [6] C. Lin, J. Li, C. Liu, S. Chang, Perfect Hashing Based Parallel Algo-
408 rithms for Multiple String Matching on Graphic Processing Units, IEEE
409 Transactions on Parallel and Distributed Systems 28 (2017) 2639–2650.
410 DOI: 10.1109/TPDS.2017.2674664.
- 411 [7] T. Suzuki, S. Kim, J. Kani, K. Suzuki, A. Otaka, T. Hanawa, Paral-
412 lelization of cipher algorithm on cpu/gpu for real-time software-defined
413 access network, in: 2015 Asia-Pacific Signal and Information Processing

- 414 Association Annual Summit and Conference (APSIPA), pp. 484–487.
415 DOI: 10.1109/APSIPA.2015.7415318.
- 416 [8] W. Bul’ajoul, A. James, M. Pannu, Improving network intrusion de-
417 tection system performance through quality of service configuration and
418 parallel technology, *Journal of Computer and System Sciences* 81 (2015)
419 981 – 999. DOI:10.1016/j.jcss.2014.12.012.
- 420 [9] W. Bulajoul, A. James, S. Shaikh, A new architecture for network
421 intrusion detection and prevention, *IEEE Access* 7 (2019) 18558–18573.
422 DOI: 10.1109/access.2019.2895898.
- 423 [10] H. Jiang, G. Zhang, G. Xie, K. Salamatian, L. Mathy, Scalable high-
424 performance parallel design for Network Intrusion Detection Systems on
425 many-core processors, in: *Architectures for Networking and Communi-
426 cations Systems*, pp. 137–146. DOI:10.1109/ANCS.2013.6665196.
- 427 [11] D. Kuvaiskii, S. Chakrabarti, M. Vij, Snort Intrusion Detection System
428 with Intel Software Guard Extension (Intel SGX), *arXiv e-prints* (2018).
429 DOI:arxiv:1802.00508.
- 430 [12] G. Vasiliadis, M. Polychronakis, S. Ioannidis, MIDeA: A Multi-Parallel
431 Intrusion Detection Architecture, in: *Proceedings of the 18th ACM
432 Conference on Computer and Communications Security, CCS 11*, Asso-
433 ciation for Computing Machinery, New York, NY, USA, 2011, p. 297308.
434 DOI:10.1145/2046707.2046741.
- 435 [13] M. A. Jamshed, J. Lee, S. Moon, I. Yun, D. Kim, S. Lee, Y. Yi, K. Park,
436 Kargus: A Highly-Scalable Software-Based Intrusion Detection System,

- 437 in: Proceedings of the 2012 ACM Conference on Computer and Com-
438 munications Security, CCS 12, Association for Computing Machinery,
439 New York, NY, USA, 2012, p. 317328. DOI:10.1145/2382196.2382232.
- 440 [14] Y. Go, M. A. Jamshed, Y. Moon, C. Hwang, K. Park, APUNet: Re-
441 vitalizing GPU as Packet Processing Accelerator, in: 14th USENIX
442 Symposium on Networked Systems Design and Implementation (NSDI
443 17), USENIX Association, Boston, MA, 2017, pp. 83–96.
- 444 [15] C. Stylianopoulos, L. Johansson, O. Olsson, M. Almgren, CLort: High
445 Throughput and Low Energy Network Intrusion Detection on IoT De-
446 vices with Embedded GPUs, in: N. Gruschka (Ed.), Secure IT Systems,
447 Springer International Publishing, 2018, pp. 187–202. DOI: 10.1007/978-
448 3-030-03638-6_12.
- 449 [16] G. Vasiliadis, S. Antonatos, M. Polychronakis, E. P. Markatos, S. Ioan-
450 nidis, Gnort: High Performance Network Intrusion Detection Using
451 Graphics Processors, in: Recent Advances in Intrusion Detection,
452 Springer Berlin Heidelberg, Berlin, Heidelberg, 2008, pp. 116–134. DOI:
453 10.1007/978-3-540-87403-4_7.
- 454 [17] Z. Zheng, J. Bi, H. Wang, C. Sun, H. Yu, H. Hu, K. Gao, J. Wu, Grus:
455 Enabling Latency SLOs for GPU-Accelerated NFV Systems, in: 2018
456 IEEE 26th International Conference on Network Protocols (ICNP), pp.
457 154–164. DOI: 10.1109/ICNP.2018.00025.
- 458 [18] X. Yi, J. Duan, C. Wu, GPUNFV: A GPU-Accelerated NFV System,
459 in: Proceedings of the First Asia-Pacific Workshop on Networking, AP-

- 460 Net17, Association for Computing Machinery, Hong Kong, China, 2017,
461 pp. 85–91. DOI: 10.1145/3106989.3106990.
- 462 [19] P. Du, R. Weber, P. Luszczek, S. Tomov, G. Peterson, J. Dongarra,
463 From CUDA to OpenCL: Towards a performance-portable solution for
464 multi-platform GPU programming, *Parallel Computing* 38 (2012) 391
465 – 407. DOI: 10.1016/j.parco.2011.10.002.
- 466 [20] W.-S. Lai, C.-C. Wu, L.-F. Lai, M.-C. Sie, Two-Phase PFAC Algorithm
467 for Multiple Patterns Matching on CUDA GPUs, *Electronics* 8 (2019)
468 270. DOI: 10.3390/electronics8030270.
- 469 [21] A. Herten, GPU-based Online Track Reconstruction for PANDA and
470 Application to the Analysis of $D \rightarrow K\pi\pi$, Dr., Ruhr-Universitt Bochum,
471 Bochum, 2015. DOI: FZJ-2015-05760.
- 472 [22] R. Farber, Chapter 7 - Techniques to Increase Parallelism, in: *CUDA*
473 *Application Design and Development*, Morgan Kaufmann, Boston, 2011,
474 pp. 157 – 177. DOI: 10.1016/B978-0-12-388426-8.00007-0.
- 475 [23] A. R. Baker, J. Esler, Chapter 2 - Introducing Snort 2.6, in: *Snort*
476 *Intrusion Detection and Prevention Toolkit*, Syngress, Rockland, 2007,
477 pp. 31–67. DOI: 10.1016/B978-159749099-3/50007-0.