GANDALF
A GPU-based Architecture For Network and Data Level Flows

Deliverable 3.1

Evaluation of GANDALF

Co-financed by Greece and the European Union

1 Introduction

In this report, we experimentally evaluate GANDALF under various configurations and workloads. We experimentally show that its novel techniques for sharing memory context between network interfaces and the GPU to avoid redundant data movement, can be used effectively to drastically improve the performance of a broad range of applications, from simple packet forwarding to more resource-intensive applications, such as packet encryption, stateful traffic classification, and intrusion detection. In addition, we show that its prototype packet scheduling mechanisms can increase the utilization of the GPU and the shared PCIe bus efficiently.

2 Performance Evaluation

2.1 Hardware Setup

Our base system is equipped with two Intel Xeon E5520 Quad-core CPUs at 2.27GHz and 12 GB of RAM (6 GB per NUMA domain). Each CPU is connected to peripherals via a separate I/O hub, linked to several PCIe slots. Figure 1 shows the overall system architecture. Each I/O hub is connected to an NVIDIA GTX480 graphics card via a PCIe v2.0 x16 slot, and one Intel 82599EB with two 10 GbE ports, via a PCIe v2.0 8× slot. Each GTX480 is equipped with 480 cores, organized in 15 multiprocessors, and 1.5GB of GDDR5 memory. The system runs Linux 3.5 with CUDA v5.0 installed. After experimentation, we have found that the best placement is to have a GPU and a NIC on each NUMA node. We also place the GPU and NIC buffers in the same memory domain, as local memory accesses sustain lower latency and more bandwidth compared to remote accesses. We also modified the NIC driver to carefully place packet buffers on the respective local memory domain.

For traffic generation we use a custom packet generator built on top of `netmap` [1]. Test traffic consists of both synthetic traffic, as well as real traffic traces.

2.2 Data Transfer

We evaluate the zero-copy mechanism by taking into account the size of the transferred packets. The system can efficiently deliver all incoming packets to user space, regardless of the packet size, by mapping the NIC's DMA packet buffer. However, small data transfers to the GPU incur sig-

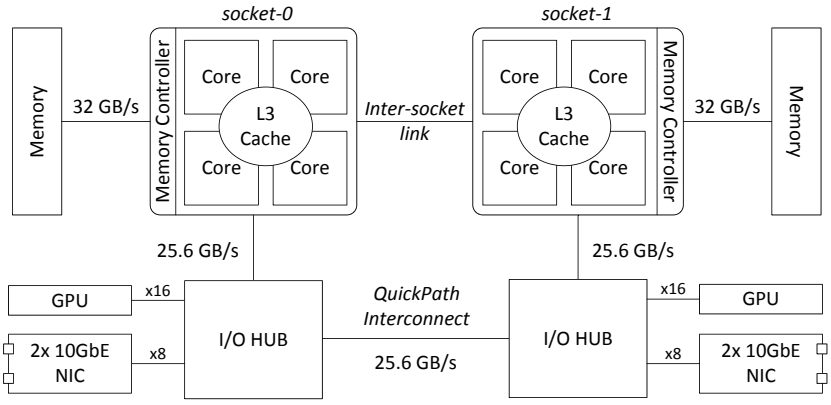


Figure 1: Hardware setup.

Table 1: Sustained PCIe throughput (Gbit/s) for transferring data to a single GPU, for different buffer sizes.

Buffer	**1KB**	**4KB**	**64KB**	**256KB**	**1MB**	**16MB**
Host to GPU	2.04	7.12	34.4	42.1	45.7	47.8
GPU to Host	2.03	6.70	21.1	23.8	24.6	24.9

nificant penalties. Table 1 shows that for transfers of less than 4KB, the PCIe throughput falls below 7 Gbit/s. With a large buffer though, the transfer rate to the GPU exceeds 45 Gbit/s, while the transfer rate from the GPU to the host decreases to about 25 Gbit/s.[1]

To overcome the low PCIe throughput, GANDALF transfers batches of network packets to the GPU, instead of one at a time. However, as packets are placed in fixed-sized slots, transferring many slots at once results in redundant data transfers when the slots are not fully occupied. As we can see in Table 2, when traffic consists of small packets, the actual PCIe throughput drops drastically. Thus, it is better to copy small network packets sequentially into another buffer, rather than transfer the corresponding slots directly. Direct transfer pays off only for packet sizes over 512 bytes (when buffer occupancy is over $512/1536 = 33.3\%$), achieving 47.8 Gbit/s for 1518-byte packets (a 2.3× speedup).

Consequently, we adopted a simple *selective offloading* scheme, whereby packets in the shared buffer are copied to another buffer sequentially (in 16-byte aligned boundaries) if the overall occupancy of the shared buffer

[1]The PCIe asymmetry in the data transfer throughput is related to the interconnection between the motherboard and the GPUs [2].

Table 2: Sustained throughput (Gbit/s) for various packet sizes, when bulk-transferring data to a single GPU.

Packet size (bytes)	64	128	256	512	1024	1518
Copy back-to-back	13.76	18.21	20.53	19.21	19.24	20.04
Zero-copy	2.06	4.03	8.07	16.13	32.26	47.83

is sparse. Otherwise, the shared buffer is transferred directly to the GPU. Occupancy is computed—without any additional overhead—by simply counting the number of bytes of the newly arrived packets every time a new interrupt is generated by the NIC.

Figure 3 shows the throughput for forwarding packets with all data transfers included, but without any GPU computations. We observe that the forwarding performance for 64-byte packets reaches 21 Gbit/s, out of the maximum 29.09 Gbit/s, while for large packets it reaches the maximum full line rate. We also observe that the GPU transfers of large packets are completely hidden on the Rx+GPU+Tx path, as they are performed in parallel using the pipeline shown in Figure 2, and thus they do not affect overall performance. Unfortunately, this is not the case for small packets (less than 128-bytes), which suffer an additional 2–9% performance hit due to memory contention.

2.3 Raw GPU Processing Throughput

Having examined data transfer costs, we now evaluate the computational performance of a single GPU—exluding all network I/O transfers—for packet decoding, connection state management, TCP stream reassembly, and some representative traffic processing applications. We plot both the performance achieved by the GPU only (Figure 4), and the performance achieved with all PCIe transfers included (Figure 5).

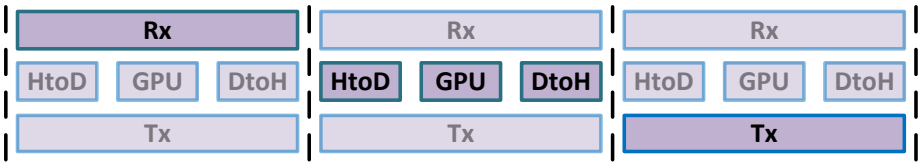


Figure 2: The I/O and processing pipeline.

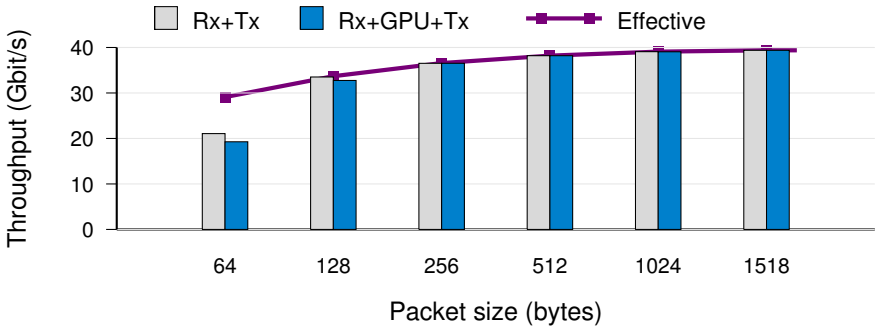


Figure 3: Data transfer throughput for different packet sizes when using two dual-port 10GbE NICs.

Table 3: Static filters and their instruction counts.

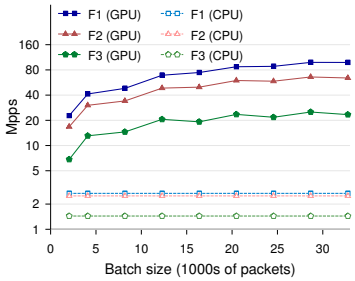
Filter	Description	Instr.
1	"ip"	4
2	"ip src net 192.168.2.0/24 and dst net 10.0.0.0/8"	10
3	"ip and tcp port (ssh or http or imap or smtp or pop3 or ftp)"	23

2.3.1 Packet Filtering

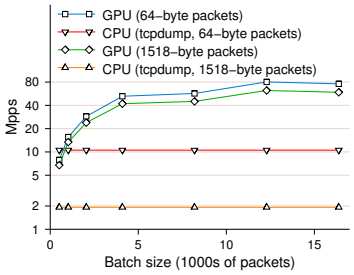
We use three sets of filters with increasing complexity, as shown in Table 3, for static filtering performance evaluation. The instruction numbers of these filters are also listed. Figures 4(a) and 5(a) show the raw GPU performance for each filter. For both CPU and GPU, the number of the produced instruction increases—almost linearly—the processing time. Still, the GPU is about 16.2–37.7 times faster than a single CPU core, and about 3.1–4.8 times faster with all PCIe transfers included, when having a sufficient number of packets that are processed in parallel.

2.3.2 Packet Decoding

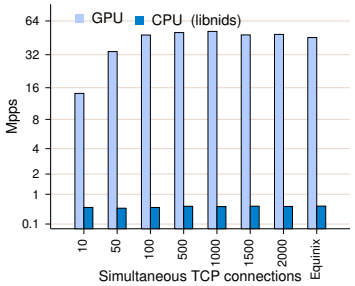
Decoding a packet according to its protocols is one of the most basic packet processing operations, and thus we use it as a base cost of our framework. Figure 5(b) shows the GPU performance, with all PCIe transfers included, for fully decoding incoming UDP packets into appropriately aligned structures (throughput is very similar for TCP). As expected, the throughput increases as the number of packets processed in parallel increases. When decoding 64-byte packets, the GPU performance with PCIe transfers included, reaches 48 Mpps, which is about 4.5 times faster than the computational throughput of the `tcpdump` decoding process



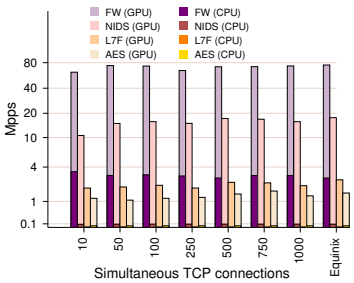
(a) BPF Filtering.



(b) Packet decode.



(c) TCP state management and reassembly



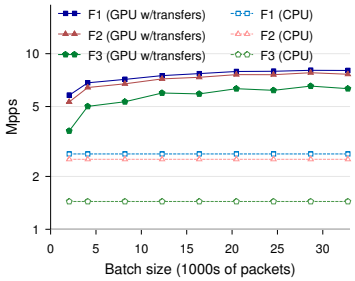
(d) Applications.

Figure 4: Average raw processing throughput, **excluding PCIe transfers**, sustained by the GPU to (a) filter network packets, (b) decode network packets, (c) maintain flow state and reassemble TCP streams, and (d) perform various network processing operations.

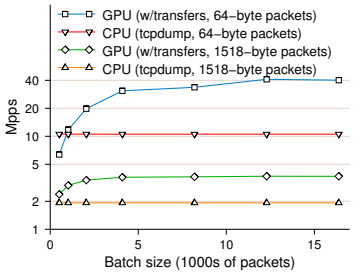
sustained by a single CPU core, when packets are read from memory. For 1518-byte packets, the GPU sustains about 3.8 Mpps and matches the performance of 1.92 CPU cores.

2.3.3 Connection State Management and TCP Stream Reassembly

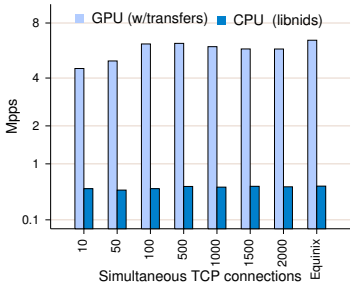
In this experiment we measure the performance of maintaining connection state on the GPU, and the performance of reassembling the packets of TCP flows into application-level streams. Figures 5(c) and 4(c) show the packets processed per second for both operations, with and without PCIe transfers accordingly. Test traffic consists of real HTTP connections with random IP addresses and TCP ports. Each connection fetches about 800KB from a server, and comprises about 870 packets (320 minimum-size ACKs, and 550 full-size data packets). We also use a trace-driven workload ("Equinix") based on a trace captured by CAIDA's *equinix-sanjose* monitor [3], in which the average and median packet length is 606.2 and



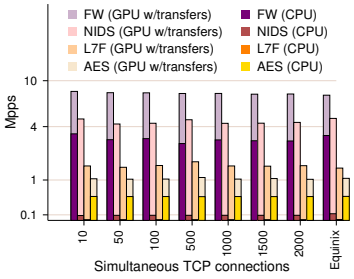
(a) BPF Filtering.



(b) Packet decode.



(c) TCP state management and reassembly



(d) Applications.

Figure 5: Average processing throughput, **including all PCIe transfers**, sustained by the GPU to (a) filter network packets, (b) decode network packets, (c) maintain flow state and reassemble TCP streams, and (d) perform various network processing operations.

81 bytes respectively.

Keeping state and reassembling streams requires several hash table lookups and updates, which result to marginal overhead for a sufficient number of simultaneous TCP connections and the Equinix trace; about 20–25% on the raw GPU performance sustained for packet decoding, that increases to 45–50% when the number of concurrent connections is low. The reason is that smaller numbers of concurrent connections result to lower parallelism. To compare with a CPU implementation, we measure the equivalent functionality of the Libnids TCP reassembly library [4], when packets are read from memory. Although Libnids implements more specific cases of the TCP stack processing, compared to GANDALF, the network traces that we used for the evaluation enforce exactly the same functionality to be exercised. We can see that the throughput of a single CPU core is about 94× lower than our GPU implementation, at about 0.55 Mpps (Figure 4(c)). Even when all PCIe data transfers are included

Table 4: Time spent (μsec) for traversing the connection table and removing expired connections.

Elements	1M buckets	8M buckets	16M buckets
0.1M	463	3,595	7,166
1M	463	3,588	7,173
2M	934	3,593	7,181
4M	1,924	3,593	7,177
8M	3,935	3,597	7,171
16M	7,991	7,430	7,173
32M	16,060	15,344	14,851

(Figure 5(c)), the GPU version achieves performance comparable to about 10 CPU cores (assuming ideal parallelization).

2.3.4 Removing Expired Connections

Removal of expired connections is very important for preventing the connection table from becoming full with stale adversarial connections, idle benign connections, or connections that failed to terminate cleanly [5]. Table 4 shows the GPU time spent for connection expiration. The time spent to traverse the table is constant when occupancy is lower than 100%, and analogous to the number of buckets; for larger values it increases due to the extra overhead of iterating the chain lists. Having a small hash table with a large load factor is better than a large but sparsely populated table. For example, the time to traverse a 1M-bucket table that contains up to 1M elements is about 20$\times$ lower than a 16M-bucket table with the same number of elements. If the occupancy is higher than 100% though, it is slightly better to use a 16M-bucket table.

2.3.5 Packet Processing Applications

In this experiment we measure the computational throughput of the GPU for the applications presented in Deliverable 2.1. The NIDS is configured to use all the `content` patterns (about 10,000 strings) of the latest Snort distribution [6], combined into a single Aho-Corasick state machine, and their corresponding `pcre` regular expressions compiled into individual DFA state machines. The application-layer filter application (L7F) uses the "best-quality" patterns (12 regular expressions for identifying common services such as HTTP and SSH) of L7-filter [7], compiled into 12 different DFA state machines. The Firewall (FW) application uses 10,000 randomly

generated rules for blocking incoming and outgoing traffic based on certain TCP/UDP port numbers and IP addresses. The test traffic consists of the HTTP-based traffic and the trace-driven Equinix workload described earlier. Note that the increased asymmetry in packet lengths and network protocols in the above traces is a stress-test workload for our data-parallel applications, given the SIMT architecture of GPUs [8].

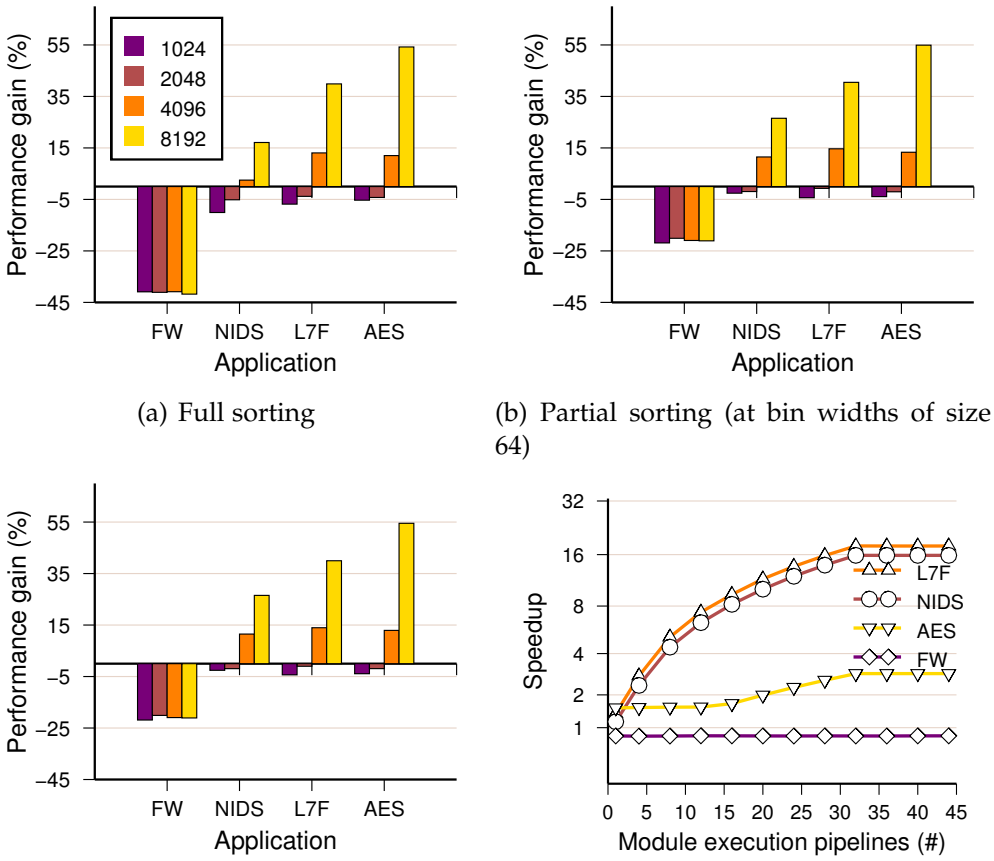
Figure 5(d) shows the GPU throughput sustained by each application, including PCIe transfers, when packets are read from host memory. FW, as expected, has the highest throughput of about 8 Mpps—about 2.3 times higher than the equivalent single-core CPU execution. The throughput for NIDS is about 4.2–5.7 Mpps, and for L7F is about 1.45–1.73 Mpps. The large difference between the two applications is due to the fact that the NIDS shares the same Aho-Corasick state machine to initially search all packets. In the common case, each packet will be matched only once against a single DFA. In contrast, the L7F requires each packet to be explicitly matched against each of the 12 different regular expression DFAs for both CPU and GPU implementations. The corresponding single-core CPU implementation of NIDS reaches about 0.1 Mpps, while L7F reaches 0.01 Mpps. We also note that both applications are explicitly forced to match all packets of all flows, even after they have been successfully classified (worst-case analysis). Finally, AES has a throughput of about 1.1 Mpps, as it is more computationally intensive. The corresponding CPU implementation using the AES-NI [9] instruction set on a single core reaches about 0.51 Mpps.[2]

2.3.6 Packet Scheduling

In this experiment we measure how the packet scheduling technique, described in Deliverable 2.1, affects the performance of different network applications. For test traffic we used the trace-driven Equinix workload.

Figures 6(a)–6(c) show the performance gain of each application for different packet batch sizes, under three grouping granularities: i) full sorting, ii) grouping packets in bins of width size equal to 64, and iii) grouping packets in bins of width size equal to 128. We observe that full sorting boosts the performance of full packet processing applications, up to 55% for computationally intensive workloads like AES. Memory-intensive applications, such as NIDS, have a lower (about 15%) benefit, that reaches about 26% when partially sorting each batch of packets, due

[2]The CPU performance of AES was measured on an Intel Xeon E5620 at 2.40GHz, because the Intel Xeon E5520 of our base system does not support AES-NI.



(a) Full sorting

(b) Partial sorting (at bin widths of size 64)

(c) Partial sorting (at bin widths of size 128)

(d) Remapping thread-module execution

Figure 6: Performance gains of packet sorting on raw GPU execution time, when executing (i) a single processing module (Figures 6(a)- 6(c)), and (ii) multiple modules simulteneously (Figure 6(d)).

to the lower sorting overheads. We also observe that gains increase as the batch size increases. With larger batch sizes, there is a greater range of packet sizes and protocols, hence more opportunities for better grouping.

In contrast, packet scheduling has a negative effect on lightweight processing (as in FW, which only processes a few bytes of each packet), because the sorting overhead is not amortized by the resulting SIMT execution. Partial sorting sustains lower overhead and is able to decrease the negative effect to about 21%. Still, as we cannot know at runtime if processing will be heavyweight or not, it is not feasible to predict if packet sorting is worth applying. As a result, quite lightweight workloads (as in

FW) will perform worse, although this lower performance will be hidden most of the time by data transfer overlap (Figure 2).

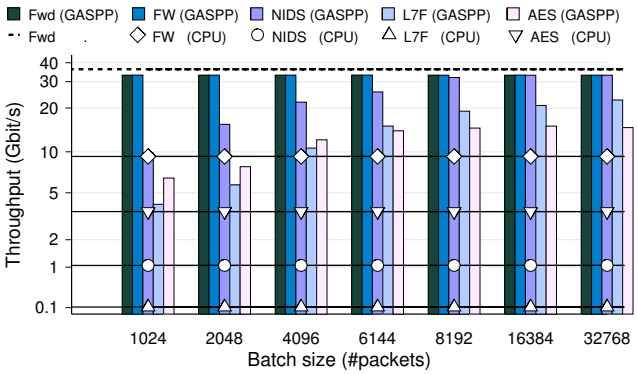
Another important aspect is how control flow divergence affects performance, e.g., when packets follow different module execution pipelines. To achieve this, we explicitly enforce different packets of the same batch to be processed by different modules. Figure 6(d) shows the achieved speedup when applying packet scheduling over the baseline case of mapping packets to thread warps without any reordering (network order). We note that although the actual work of the modules is the same every time (i.e., the same processing will be applied on each packet), it is executed by different code blocks, thus execution is forced to diverge. We see that as the number of different modules increases, our packet scheduling technique achieves a significant speedup. The speedup stabilizes after the number of modules exceeds 32, as only 32 threads (warp size) can run in a SIMT manner any given time. In general, code divergence within warps plays a significant role in GPU performance. The thread remapping achieved through our packet scheduling technique tolerates the irregular code execution at a fixed cost.

2.4 End-to-End Performance

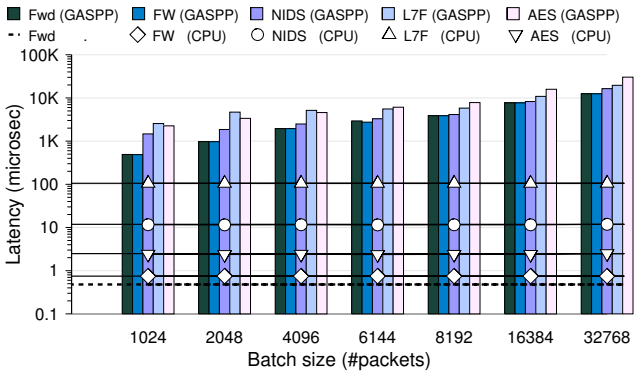
2.4.1 Individual Applications

Figure 7 shows the sustained end-to-end forwarding throughput and latency of individual GANDALF-enabled applications for different batch sizes. We use four different traffic generators, equal to the number of available 10 GbE ports in our system. To prevent synchronization effects between the generators, the test workload consists of the HTTP-based traffic described earlier. For comparison, we also evaluate the corresponding CPU-based implementations running on a single core, on top of `netmap`.

The FW application can process all traffic delivered to the GPU, even for small batch sizes. NIDS, L7F, and AES, on the other hand, require larger batch sizes. The NIDS application requires batches of 8,192 packets to reach similar performance. Equivalent performance would be achieved (assuming ideal parallelization) by 28.4 CPU cores. More computationally intensive applications, however, such as L7F and AES, cannot process all traffic. L7F reaches 19 Gbit/s a batch size of 8,192 packets, and converges to 22.6 Gbit/s for larger sizes—about 205.1 times faster than a single CPU core. AES converges to about 15.8 Gbit/s, and matches the performance



(a) Throughput.



(b) Latency.

Figure 7: Sustained traffic forwarding throughput (a) and latency (b) for GANDALF-enabled applications.

of 4.4 CPU cores with AES-NI support. As expected, latency increases linearly with the batch size, and for certain applications and large batch sizes it can reach tens of milliseconds (Figure 7(b)). Fortunately, a batch size of 8,192 packets allows for a reasonable latency for all applications, while it sufficiently utilizes the PCIe bus and the parallel capabilities of the GTX480 card (Figure 7(a)). For instance, NIDS, L7F, and FW have a latency of 3–5 ms, while AES, which suffers from an extra GPU-to-host data transfer, has a latency of 7.8 ms.

2.4.2 Consolidated Applications

Consolidating multiple applications has the benefit of distributing the overhead of data transfer, packet decoding, state management, and stream reassembly across all applications, as all these operations are performed only once. Moreover, through the use of context keys, GANDALF opti-

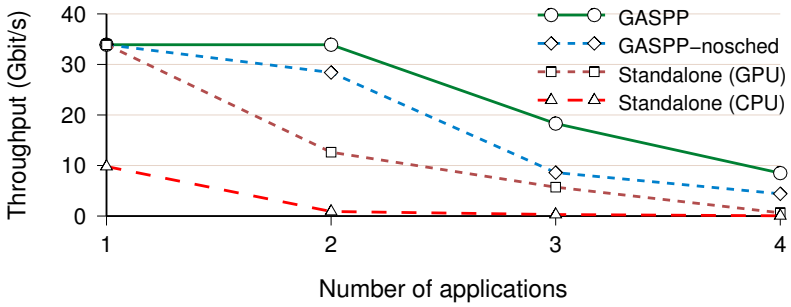


Figure 8: Sustained throughput for concurrently running applications.

mizes SIMT execution when packets of the same batch are processed by different applications. Figure 8 shows the sustained throughput when running multiple GANDALF applications. Applications are added in the following order: FW, NIDS, L7F, AES (increasing overhead). We also enforce packets of different connections to follow different application processing paths. Specifically, we use the hash of the each packet's 5-tuple for deciding the order of execution. For example, a class of packets will be processed by application 1 and then application 2, while others will be processed by application 2 and then by application 1; eventually, all packets will be processed by all registered applications. For comparison, we also plot the performance of GANDALF when packet scheduling is disabled (GANDALF-nosched), and the performance of having multiple standalone applications running on the GPU and the CPU.

We see that the throughput for GANDALF converges to the throughput of the most intensive application. When combining the first two applications, the throughput remains at 33.9 Gbit/s. When adding the L7F (x=3), performance degrades to 18.3 Gbit/s. L7F alone reaches about 20 Gbit/s (Figure 7(a)). When adding AES (x=4), performance drops to 8.5 Gbit/s, which is about 1.93× faster than GANDALF-nosched. The achieved throughput when running multiple standalone GPU-based implementations is about 16.25× lower than GANDALF, due to excessive data transfers.

3 Conclusion

We have presented the evaluation of GANDALF, a flexible, efficient, and high-performance framework for network traffic processing applications. GANDALF can achieve multi-gigabit traffic forwarding rates even for

complex and computationally intensive network operations such as stateful traffic classification, intrusion detection, and packet encryption. Especially when consolidating multiple network applications on the same system, GANDALF can achieve up to 16.2× speedup compared to standalone GPU-based implementations of the same applications.

References

[1] L. Rizzo, "netmap: A Novel Framework for Fast Packet I/O," in *Proceedings of the 2012 USENIX conference on USENIX Annual Technical Conference*, 2012.

[2] S. Han, K. Jang, K. Park, and S. Moon, "PacketShader: A GPU-accelerated Software Router," in *Proceedings of the ACM SIGCOMM 2010 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, August 2010.

[3] http://www.caida.org/data/passive/passive_2011_dataset.xml.

[4] "Libnids library," http://libnids.sourceforge.net/.

[5] M. Vutukuru, H. Balakrishnan, and V. Paxson, "Efficient and Robust TCP Stream Normalization," in *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, 2008.

[6] "Snort IDS/IPS," http://www.snort.org.

[7] http://l7-filter.sourceforge.net/.

[8] "CUDA Programming Guide," http://http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

[9] "AES-NI," https://software.intel.com/sites/default/files/article/165683/aes-wp-2012-09-22-v01.pdf.