GASPP: A GPU-Accelerated Stateful Packet Processing Framework

Giorgos Vasiliadis,* Lazaros Koromilas,* Michalis Polychronakis,† Sotiris Ioannidis*
*FORTH-ICS, †Columbia University
{gvasil,koromil,sotiris}@ics.forth.gr, mikepo@cs.columbia.edu

Abstract

Graphics processing units (GPUs) are a powerful platform for building high-speed network traffic processing applications using low-cost hardware. Existing systems tap the massively parallel architecture of GPUs to speed up certain computationally intensive tasks, such as cryptographic operations and pattern matching. However, they still suffer from significant overheads due to critical-path operations that are still being carried out on the CPU, and redundant inter-device data transfers.

In this paper we present GASPP, a programmable network traffic processing framework tailored to modern graphics processors. GASPP integrates optimized GPU-based implementations of a broad range of operations commonly used in network traffic processing applications, including the first purely GPU-based implementation of network flow tracking and TCP stream reassembly. GASPP also employs novel mechanisms for tackling control flow irregularities across SIMT threads, and sharing memory context between the network interface and the GPU. Our evaluation shows that GASPP can achieve multi-gigabit traffic forwarding rates even for computationally intensive and complex network operations such as stateful traffic classification, intrusion detection, and packet encryption. Especially when consolidating multiple network applications on the same device, GASPP achieves up to 16.2× speedup compared to standalone GPU-based implementations of the same applications.

1 Introduction

The emergence of commodity *many*-core architectures, such as multicore CPUs and modern graphics processors (GPUs) has proven to be a good solution for accelerating many network applications, and has led to their successful deployment in high-speed environments [10, 12–14, 26]. Recent trends have shown that certain network packet processing operations can be implemented efficiently on GPU architectures. Typically, such operations are either computationally intensive (e.g., encryption [14]), memory-intensive (e.g., IP routing [12]), or both (e.g., intrusion detection and prevention [13, 24, 26]). Modern GPU architectures offer high computational throughput and hide excessive memory latencies.

Unfortunately, the lack of programming abstractions and GPU-based libraries for network traffic processing—even for simple tasks such as packet decoding and filtering—increases significantly the programming effort needed to build, extend, and maintain high-performance GPU-based network applications. More complex critical-path operations, such as flow tracking and TCP stream reassembly, currently still run on the CPU, negatively offsetting any performance gains by the offloaded GPU operations. The absence of adequate OS support also increases the cost of data transfers between the host and I/O devices. For example, packets have to be transferred from the network interface to the user-space context of the application, and from there to kernel space in order to be transferred to the GPU. While programmers can explicitly optimize data movements, this increases the design complexity and code size of even simple GPU-based packet processing programs.

As a step towards tackling the above inefficiencies, we present *GASPP*, a network traffic processing framework tailored to modern graphics processors. GASPP integrates into a purely GPU-powered implementation many of the most common operations used by different types of network traffic processing applications, including the first GPU-based implementation of network flow tracking and TCP stream reassembly. By hiding complicated network processing issues while providing a rich and expressive interface that exposes only the data that matters to applications, GASPP allows developers to build complex GPU-based network traffic processing applications in a flexible and efficient way.

We have developed and integrated into GASPP novel mechanisms for sharing memory context between network interfaces and the GPU to avoid redundant data movement, and for scheduling packets in an efficient way

that increases the utilization of the GPU and the shared PCIe bus. Overall, GASPP allows applications to scale in terms of performance, and carry out on the CPU only infrequently occurring operations.

The main contributions of our work are:

- We have designed, implemented, and evaluated GASPP, a novel GPU-based framework for high-performance network traffic processing, which eases the development of applications that process data at multiple layers of the protocol stack.
- We present the first (to the best of our knowledge) purely GPU-based implementation of flow state management and TCP stream reconstruction.
- We present a novel packet scheduling technique that tackles control flow irregularities and load imbalance across GPU threads.
- We present a zero-copy mechanism that avoids redundant memory copies between the network interface and the GPU, increasing significantly the throughput of cross-device data transfers.

2 Motivation

The Need for Modularity. The rise of general-purpose computing on GPUs (GPGPU) and related frameworks, such as CUDA and OpenCL, has made the implementation of GPU-accelerated applications easier than ever. Unfortunately, the majority of GPU-assisted network applications follow a monolithic design, lacking both modularity and flexibility. As a result, building, maintaining, and extending such systems eventually becomes a real burden. In addition, the absence of libraries for network processing operations—even for simple tasks like packet decoding or filtering—increases development costs even further. GASPP integrates a broad range of operations that different types of network applications rely on, with all the advantages of a GPU-powered implementation, into a single application development platform. This allows developers to focus on core application logic, alleviating the low-level technical challenges of data transfer to and from the GPU, packet batching, asynchronous execution, synchronization issues, connection state management, and so on.

The Need for Stateful Processing. Flow tracking and TCP stream reconstruction are mandatory features of a broad range of network applications. Intrusion detection and traffic classification systems typically inspect the application-layer stream to identify patterns that span multiple packets and thwart evasion attacks [9, 28]. Existing GPU-assisted network processing applications, however, just offload to the GPU certain data-parallel tasks, and are saturated by the many computationally heavy operations that are still being carried out on the CPU, such as network flow tracking, TCP stream reassembly, and protocol parsing [13, 26].

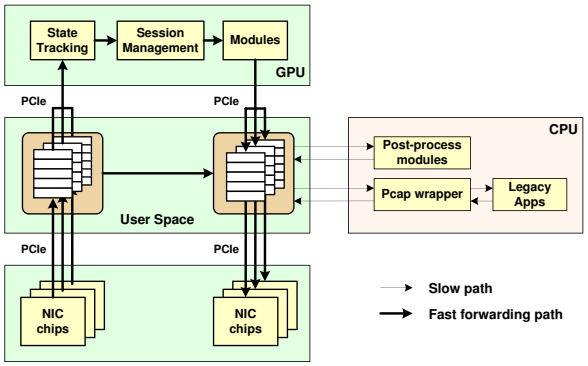


Figure 1: GASPP architecture.

The most common approach for stateful processing is to buffer incoming packets, reassemble them, and deliver "chunks" of the reassembled stream to higher-level processing elements [6, 7]. A major drawback of this approach is that it requires several data copies and significant extra memory space. In Gigabit networks, where packet intervals can be as short as 1.25 μsec (in a 10GbE network, for a MTU of 1.5KB), packet buffering requires large amounts of memory even for very short time windows. To address these challenges, the primary objectives of our GPU-based stateful processing implementation are: (i) process as many packets as possible *on-the-fly* (instead of buffering them), and (ii) ensure that packets of the same connection are processed *in-order*.

3 Design

The high-level design of GASPP is shown in Figure 1. Packets are transferred from the network interfaces to the memory space of the GPU in batches. The captured packets are then classified according to their protocol and are processed in parallel by the GPU. For stateful protocols, connection state management and TCP stream reconstruction are supported for delivering a consistent application-layer byte stream.

GASPP applications consist of *modules* that control all aspects of the traffic processing flow. Modules are represented as GPU device functions, and take as input a network packet or stream chunk. Internally, each module is executed in parallel on a batch of packets. After processing is completed, the packets are transferred back to the memory space of the host, and depending on the application, to the appropriate output network interface.

3.1 Processing Modules

A central concept of NVIDIA's CUDA [5] that has influenced the design of GASPP is the organization of GPU programs into *kernels*, which in essence are functions that are executed by groups of threads. GASPP allows

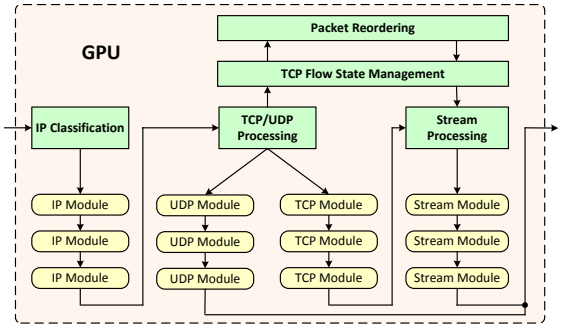


Figure 2: GPU packet processing pipeline. The pipeline is executed by a different thread for every incoming packet.

users to specify processing tasks on the incoming traffic by writing GASPP *modules*, applicable on different protocol layers, which are then mapped into GPU kernel functions. Modules can be implemented according to the following prototypes:

```
__device__ uint processEth(unsigned pktid,
    ethhdr *eth, uint cxtkey);
__device__ uint processIP(unsigned pktid,
    ethhdr *eth, iphdr *ip, uint cxtkey);
__device__ uint processUDP(unsigned pktid,
    ethhdr *eth, iphdr *ip, udphdr *udp, uint cxtkey);
__device__ uint processTCP(unsigned pktid,
    ethhdr *eth, iphdr *ip, tcphdr *tcp, uint cxtkey);
__device__ uint processStream(unsigned pktid,
    ethhdr *eth, iphdr *ip, tcphdr *tcp, uchar *chunk,
    unsigned chunklen, uint cxtkey);
```

The framework is responsible for decoding incoming packets and executing all registered `process*()` modules by passing the appropriate parameters. Packet decoding and stream reassembly is performed by the underlying system, eliminating any extra effort from the side of the developer. Each module is executed at the corresponding layer, with pointer arguments to the encapsulated protocol headers. Arguments also include a unique identifier for each packet and a user-defined key that denotes the packet's class (described in more detail in §5.3). Currently, GASPP supports the most common network protocols, such as Ethernet, IP, TCP and UDP. Other protocols can easily be handled by explicitly parsing raw packets. Modules are executed per-packet in a data-parallel fashion. If more than one modules have been registered, they are executed back-to-back in a packet processing pipeline, resulting in GPU module chains, as shown in Figure 2.

The `processStream()` modules are executed whenever a new normalized TCP chunk of data is available. These modules are responsible for keeping internally the state between consecutive chunks—or, alternatively, for storing chunks in global memory for future use—and continuing the processing from the last state of the previous chunk. For example, a pattern matching application can match the contents of the current chunk and keep the state of its matching algorithm to a global variable; on the arrival of the next chunk, the matching process will continue from the previously stored state.

As modules are simple to write, we expect that users will easily write new ones as needed using the function prototypes described above. In fact, the complete implementation of a module that simply passes packets from an input to an output interface takes only a few lines of code. More complex network applications, such as NIDS, L7 traffic classification, and packet encryption, require a few dozen lines of code, as described in §6.

3.2 API

To cover the needs of a broad range of network traffic processing applications, GASPP offers a rich GPU API with data structures and algorithms for processing network packets.

Shared Hash Table. GASPP enables applications to access the processed data through a global hash table. Data stored in an instance of the hash table is persistent across GPU kernel invocations, and is shared between the host and the device. Internally, data objects are hashed and mapped to a given bucket. To enable GPU threads to add or remove nodes from the table in parallel, we associate an atomic lock with each bucket, so that only a single thread can make changes to a given bucket at a time.

Pattern Matching. Our framework provides a GPU-based API for matching fixed strings and regular expressions. We have ported a variant of the Aho-Corasick algorithm for string searching, and use a DFA-based implementation for regular expression matching. Both implementations have linear complexity over the input data, independent of the number of patterns to be searched. To utilize efficiently the GPU memory subsystem, packet payloads are accessed 16-bytes at a time, using an `int4` variable [27].

Cipher Operations. Currently, GASPP provides AES (128-bit to 512-bit key sizes) and RSA (1024-bit and 2048-bit key sizes) functions for encryption and decryption, and supports all modes of AES (ECB, CTR, CFB and OFB). Again, packet contents are read and written 16-bytes at a time, as this substantially improves GPU performance. The encryption and decryption process happens in-place and as packet lengths may be modified, the checksums for IP and TCP/UDP packets are recomputed to be consistent. In cases where the NIC controller supports checksum computation offloading, GASPP simply forwards the altered packets to the NIC.

Network Packet Manipulation Functions. GASPP provides special functions for dropping network packets (`Drop()`), ignoring any subsequent registered user-

defined modules (`Ignore()`), passing packets to the host for further processing (`ToLinux()`), or writing their contents to a dump file (`ToDump()`). Each function updates accordingly the packet index array, which holds the offsets where each packet is stored in the packet buffer, and a separate "metadata" array.

4 Stateful Protocol Analysis

The stateful protocol analysis component of GASPP is designed with minimal complexity so as to maximize processing speed. This component is responsible for maintaining the state of TCP connections, and reconstructing the application-level byte stream by merging packet payloads and reordering out-of-order packets.

4.1 Flow Tracking

GASPP uses a connection table array stored in the global device memory of the GPU for keeping the state of TCP connections. Each record is 17-byte long. A 4-byte hash of the source and destination IP addresses and TCP ports is used to handle collisions in the flow classifier. Connection state is stored in a single-byte variable. The sequence numbers of the most recently received client and server segments are stored in two 4-byte fields, and are updated every time the next in-order segment arrives. Hash table collisions are handled using a locking chained hash table with linked lists (described in detail in §3.2). A 4-byte pointer points to the next record (if any).

The connection table can easily fill up with adversarial partially-established connections, benign connections that stay idle for a long time, or connections that failed to terminate properly. For this reason, connection records that have been idle for more than a certain timeout, set to 60 seconds by default, are periodically removed. As current GPU devices do not provide support for measuring real-world time, we resort to a separate GPU kernel that is initiated periodically according to the timeout value. Its task is to simply mark each connection record by setting the first bit of the state variable. If a connection record is already marked, it is removed from the table. A marked record is unmarked when a new packet for this connection is received before the timeout expires.

4.2 Parallelizing TCP Stream Reassembly

Maintaining the state of incoming connections is simple as long as the packets that are processed in parallel by the GPU belong to different connections. Typically, however, a batch of packets usually contains several packets of the same connection. It is thus important to ensure that the order of connection updates will be correct when processing packets of the same connection in parallel.

TCP reconstruction threads are synchronized through a separate array used for pairing threads that must process consecutive packets. When a new batch is received, each thread hashes its packet twice: once using $hash(addr\_s, addr\_d, port\_s, port\_d, seq)$, and a second time using $hash(addr\_s, addr\_d, port\_s, port\_d, seq + len)$, as shown in Figure 3. A memory barrier is used to guarantee that all threads have finished hashing their packets. Using this scheme, two packets x and y are consecutive if: $hash_x(4\text{-}tuple, seq + len) = hash_y(4\text{-}tuple, seq)$. The hash function is unidirectional to ensure that each stream direction is reconstructed separately. The SYN and SYN-ACK packets are paired by hashing the sequence and acknowledge numbers correspondingly. If both the SYN and SYN-ACK packets are present, the state of the connection is changed to `ESTABLISHED`, otherwise if only the SYN packet is present, the state is set to `SYN_RECEIVED`.

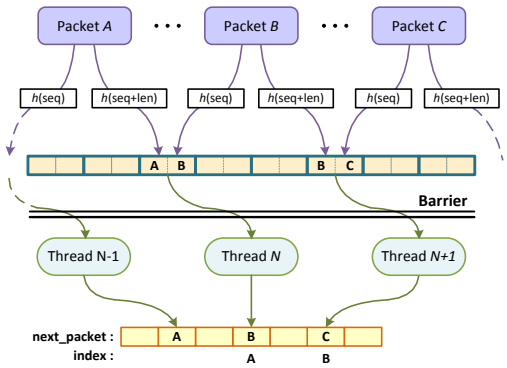


Figure 3: Ordering sequential TCP packets in parallel. The resulting *next_packet* array contains the next in-order packet, if any (i.e. *next_packet[A] = B*).

Having hashed all pairs of consecutive packets in the hash table, the next step is to create the proper packet ordering for each TCP stream using the `next_packet` array, as shown in Figure 3. Each packet is uniquely identified by an `id`, which corresponds to the index where the packet is stored in the packet index array. The `next_packet` array is set at the beginning of the current batch, and its cells contain the `id` of the next in-order packet (or `-1` if it does not exist in the current batch), e.g., if `x` is the `id` of the current packet, the `id` of the next in-order packet will be `y = next_packet[x]`. Finally, the connection table is updated with the sequence number of the last packet of each flow direction, i.e., the packet `x` that does not have a next packet in the current batch.

4.3 Packet Reordering

Although batch processing handles out-of-order packets that are included in the same batch, it does not solve the problem in the general case. A potential solution for inline applications would be to just drop out-of-sequence packets, forcing the host to retransmit them. Whenever an expected packet would be missing, subsequent pack-

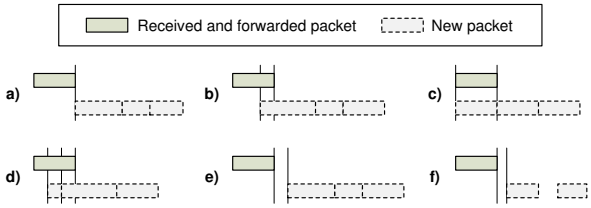


Figure 4: Subsequent packets (dashed line) may arrive in-sequence ((a)–(d)) or out of order, creating holes in the reconstructed TCP stream ((e)–(f)).

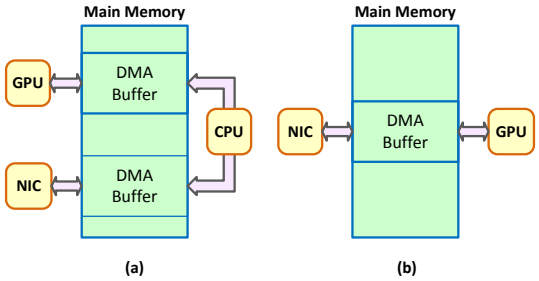


Figure 5: Normal (a) and zero-copy (b) data transfer between the NIC and the GPU.

ets would be actively dropped until the missing packet arrives. Although this approach would ensure an in-order packet flow, it has several disadvantages. First, in situations where the percentage of out-of-order packets is high, performance will degrade. Second, if the endpoints are using selective retransmission and there is a high rate of data loss in the network, connections would be rendered unusable due to excessive packet drops.

To deal with TCP sequence hole scenarios, GASPP only processes packets with sequence numbers less than or equal to the connection's current sequence number (Figure 4(a)–(d)). Received packets with no preceding packets in the current batch and with sequence numbers larger than the ones stored in the connection table imply sequence holes (Figure 4(e)–(f)), and are copied in a separate buffer in global device memory. If a thread encounters an out-of-order packet (i.e., a packet with a sequence number larger than the sequence number stored in the connection table, with no preceding packet in the current batch after the hashing calculations of §4.2), it traverses the `next_packet` array and marks as out-of-order all subsequent packets of the same flow contained in the current batch (if any). This allows the system to identify sequences of out-of-order packets, as the ones shown in the examples of Figure 4(e)–(f). The buffer size is configurable and can be up to several hundred MBs, depending on the network needs. If the buffer contains any out-of-order packets, these are processed right after a new batch of incoming packets is processed.

Although packets are copied using the very fast device-to-device copy mechanism, with a memory bandwidth of about 145 GB/s, an increased number of out-of-order packets can have a major effect on overall performance. For this reason, by default we limit the number of out-of-order packets that can be buffered to be equal to the available slots in a batch of packets. This size is enough under normal conditions, where out-of-order packets are quite rare [9], and it can be configured as needed for other environments. If the percentage of out-of-order packets exceeds this limit, our system starts to drop out-of-order packets, causing the corresponding host to retransmit them.

5 Optimizing Performance

5.1 Inter-Device Data Transfer

The problem of data transfers between the CPU and the GPU is well-known in the GPGPU community, as it results in redundant cross-device communication. The traditional approach is to exchange data using DMA between the memory regions assigned by the OS to each device. As shown in Figure 5(a), network packets are transferred to the page-locked memory of the NIC, then copied to the page-locked memory of the GPU, and from there, they are finally transferred to the GPU.

To avoid costly packet copies and context switches, GASPP uses a single buffer for efficient data sharing between the NIC and the GPU, as shown in Figure 5(b), by adjusting the `netmap` module [20]. The shared buffer is added to the internal tracking mechanism of the CUDA driver to automatically accelerate calls to functions, as it can be accessed directly by the GPU. The buffer is managed by GASPP through the specification of a policy based on time and size constraints. This enables real-time applications to process incoming packets whenever a timeout is triggered, instead of waiting for buffers to fill up over a specified threshold. Per-packet buffer allocation overheads are reduced by transferring several packets at a time. Buffers consist of fixed-size slots, with each slot corresponding to one packet in the hardware queue. Slots are reused whenever the circular hardware queue wraps around. The size of each slot is 1,536 bytes, which is consistent with the NIC's alignment requirements, and enough for the typical 1,518-byte maximum Ethernet frame size.

Although making the NIC's packet queue directly accessible to the GPU eliminates redundant copies, this does not always lead to better performance. As previous studies have shown [12, 26] (we verify their results in §7.1), contrary to NICs, current GPU implementations suffer from poor performance for small data transfers. To improve PCIe throughput, we batch several packets and transfer them at once. However, the fixed-size partitioning of the NIC's queue leads to redundant data transfers for traffic with many small packets. For example, a 64-

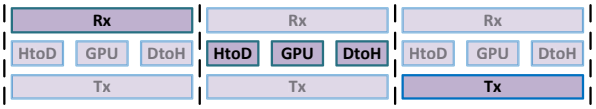


Figure 6: The I/O and processing pipeline.

byte packet consumes only 1/24th of the available space in its slot. This introduces an interesting trade-off, and as we show in §7.1, occasionally it is better to copy packets back-to-back into a second buffer and transferring it to the GPU. GASPP dynamically switches to the optimal approach by monitoring the actual utilization of the slots.

The forwarding path requires the transmission of network packets after processing is completed, and this is achieved using a triple-pipeline solution, as shown in Figure 6. Packet reception, GPU data transfers and execution, and packet transmission are executed asynchronously in a multiplexed manner.

5.2 Packet Decoding

Memory alignment is a major factor that affects the packet decoding process, as GPU execution constrains memory accesses to be aligned for all data types. For example, `int` variables should be stored to addresses that are a multiple of `sizeof(int)`. Due to the layered nature of network protocols, however, several fields of encapsulated protocols are not aligned when transferred to the memory space of the GPU. To overcome this issue, GASPP reads the packet headers from global memory, parses them using bitwise logic and shifting operations, and stores them in appropriately aligned structures. To optimize memory usage, input data is accessed in units of 16 bytes (using an `int4` variable).

5.3 Packet Scheduling

Registered modules are scheduled on the GPU, per protocol, in a serial fashion. Whenever a new batch of packets is available, it is processed in parallel using a number of threads equal to the number of packets in the batch (each thread processes a different packet). As shown in Figure 2, all registered modules for a certain protocol are executed serially on decoded packets in a lockstep way.

Network packets are processed by different threads, grouped together into logical units known as *warps* (in current NVIDIA GPU architectures, 32 threads form a warp) and mapped to SIMT units. As threads within the same warp have to execute the same instructions, load imbalance and code flow divergence within a warp can cause inefficiencies. This may occur under the following primary conditions: (i) when processing different transport-layer protocols (i.e., TCP and UDP) in the same warp, (ii) in full-packet processing applications when packet lengths within a warp differ significantly, and (iii) when different packets follow different processing paths, i.e., threads of the same warp execute different user-defined modules.

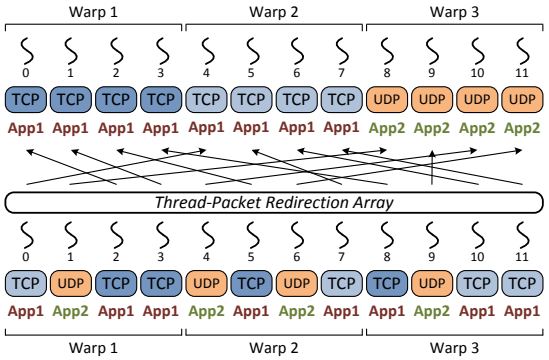


Figure 7: Packet scheduling for eliminating control flow divergences and load imbalances. Packet brightness represents packet size.

As the received traffic mix is typically very dynamic, it is essential to find an appropriate mapping between threads and network packets at runtime. It is also crucial that the overhead of the mapping process is low, so as to not jeopardize overall performance. To that end, our basic strategy is to group the packets of a batch according to their encapsulated transport-layer protocol and their length. In addition, module developers can specify *context keys* to describe packets that belong to the same class, which should follow the same module execution pipeline. A context key is a value returned by a user-defined module and is passed (as the final parameter) to the next registered module. GASPP uses these context keys to further pack packets of the same class together and map them to threads of the same warp after each module execution. This gives developers the flexibility to build complex packet processing pipelines that will be mapped efficiently to the underlying GPU architecture at runtime.

To group a batch of packets on the GPU, we have adapted a GPU-based radix sort implementation [1]. Specifically, we assign a separate weight for each packet consisting of the byte concatenation of the `ip_proto` field of its IP header, the value of the context key returned by the previously executed module, and its length. Weights are calculated on the GPU after each module execution using a separate thread for each packet, and are used by the radix sort algorithm to group the packets. Moreover, instead of copying each packet to the appropriate (i.e., sorted) position, we simply change their order in the packet index array. We also attempted to relocate packets by transposing the packet array on the GPU device memory, in order to benefit from memory coalescing [5]. Unfortunately, the overall cost of the corresponding data movements was not amortized by the resulting memory coalescing gains.

Using the above procedure, GASPP assigns dynamically to the same warp any similar-sized packets meant to be processed by the same module, as shown in Figure 7. Packets that were discarded earlier or of which the processing pipeline has been completed are grouped and mapped to warps that contain only idle threads—otherwise warps would contain both idle and active threads, degrading the utilization of the SIMT processors. To prevent packet reordering from taking place during packet forwarding, we also preserve the initial (presorted) packet index array. In §7.2 we analyze in detail how control flow divergence affects the performance of the GPU, and show how our packet scheduling mechanisms tackle the irregular code execution at a fixed cost.

6 Developing with GASPP

In this section we present simple examples of representative applications built using the GASPP framework.

L3/L4 Firewall. Firewalls operate at the network layer (port-based) or the application layer (content-based). For our purposes, we have built a GASPP module that can drop traffic based on Layer-3 and Layer-4 rules. An incoming packet is filtered if the corresponding IP addresses and port numbers are found in the hash table; otherwise the packet is forwarded.

L7 Traffic Classification. We have implemented a L7 traffic classification tool (similar to the L7-filter tool [2]) on top of GASPP. The tool dynamically loads the pattern set of the L7-filter tool, in which each application-level protocol (HTTP, SMTP, etc.) is represented by a different regular expression. At runtime, each incoming flow is matched against each regular expression independently. In order to match patterns that cross TCP segment boundaries that lie on the same batch, each thread continues the processing to the next TCP segment (obtained from the `next_packet` array). The processing of the next TCP segment continues until a final or a fail DFA-state is reached, as suggested in [25]. In addition, the DFA-state of the last TCP segment of the current batch is stored in a global variable, so that on the arrival of the next stream chunk, the matching process continues from the previously stored state. This allows the detection of regular expressions that span (potentially deliberately) not only multiple packets, but also two or more stream chunks.

Signature-based Intrusion Detection. Modern NIDS, such as Snort [7], use a large number of regular expressions to determine whether a packet stream contains an attack vector or not. To reduce the number of packets that need to be matched against a regular expression, typical NIDS take advantage of the string matching engine and use it as a first-level filtering mechanism before proceeding to regular expression matching. We have implemented the same functionality on top of GASPP, using a different module for scanning each incoming traffic stream against all the fixed strings in a signature set. Patterns that cross TCP segments are handled similarly to the L7 Traffic Classification module. Only the matching streams are further processed against the corresponding regular expressions set.

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

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

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

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

AES. Encryption is used by protocols and services, such as SSL, VPN, and IPsec, for securing communications by authenticating and encrypting the IP packets of a communication session. While stock protocol suites that are used to secure communications, such as IPsec, actually use connectionless integrity and data origin authentication, for simplicity, we only encrypt all incoming packets using the AES-CBC algorithm and a different 128-bit key for each connection.

7 Performance Evaluation

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. 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. 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.

For traffic generation we use a custom packet generator built on top of `netmap` [20]. Test traffic consists of both synthetic traffic, as well as real traffic traces.

7.1 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

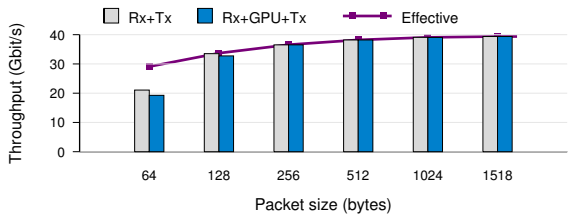


Figure 8: Data transfer throughput for different packet sizes when using two dual-port 10GbE NICs.

packet buffer. However, small data transfers to the GPU incur significant 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, GASPP 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 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 8 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 6, 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% hit due to memory contention.

[1]The PCIe asymmetry in the data transfer throughput is related to the interconnection between the motherboard and the GPUs [12].

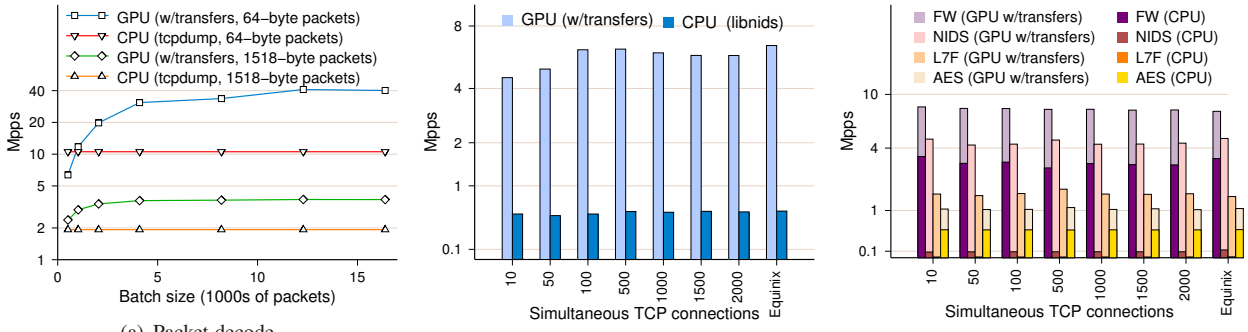
7.2 Raw GPU Processing Throughput

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.

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 9(a) shows the GPU performance for fully decoding incoming UDP packets into appropriately aligned structures, as described in §5.2 (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 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.

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. Figure 9(b) shows the packets processed per second for both operations. 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 81 bytes respectively.

Keeping state and reassembling streams requires several hashtable 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 [6], when packets are read from memory. Although Libnids implements more specific cases of the TCP stack processing, compared to GASPP, 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 0.55 Mpps, about 10× lower than the GPU version with all PCIe data transfers included.



(a) Packet decode.

(b) TCP state management and reassembly

(c) Applications.

Figure 9: Average processing throughput sustained by the GPU to (a) decode network packets, (b) maintain flow state and reassemble TCP streams, and (c) perform various network processing operations.

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

Table 3: Time spent (μsec) for traversing the connection table and removing expired connections.

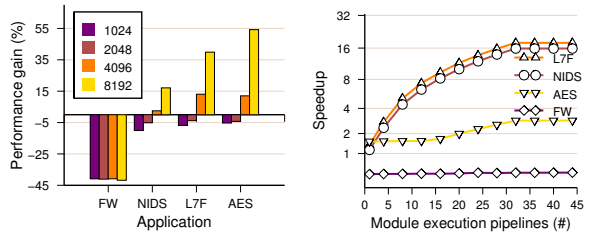
Removing Expired Connections. Removal of expired connections is very important for preventing the connection table from becoming full with stale adversarial connections, idle benign connections, or connections that failed to terminate cleanly [28]. Table 3 shows the GPU time spent for connection expiration. 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× 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.

Packet Processing Applications. In this experiment we measure the computational throughput of the GPU for the applications presented in §6. The NIDS is configured to use all the `content` patterns (about 10,000 strings) of the latest Snort distribution [7], 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 [2], 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 [5].

Figure 9(c) 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 (as we described in §6). 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 [4] instruction set on a single core reaches about 0.51 Mpps.[2]

Packet Scheduling In this experiment we measure how the packet scheduling technique, described in §5.3,

[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.



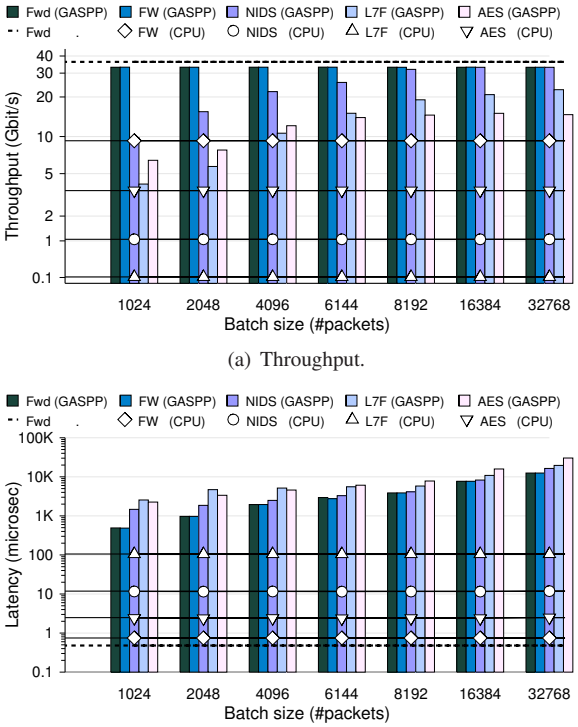
(a) Load imbalance (b) Divergence

Figure 10: Performance gains on raw GPU execution time when applying packet scheduling (the scheduling cost is included).

affects the performance of different network applications. For test traffic we used the trace-driven Equinix workload. Figure 10(a) shows the performance gain of each application for different packet batch sizes. 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 observe that packet scheduling 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. 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. As we cannot know at runtime if processing will be heavyweight or not, it is not feasible to predict if packet scheduling 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 6).

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 10(b) shows the achieved speedup when applying packet scheduling over the baseline case of mapping packets to thread warps without any reordering (network order). 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.



(a) Throughput.

(b) Latency.

Figure 11: Sustained traffic forwarding throughput (a) and latency (b) for GASPP-enabled applications.

7.3 End-to-End Performance

Individual Applications. Figure 11 shows the sustained end-to-end forwarding throughput and latency of individual GASPP-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 perfor-

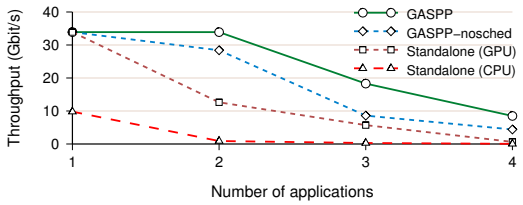


Figure 12: Sustained throughput for concurrently running applications.

mance 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 11(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 11(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.

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, GASPP optimizes SIMT execution when packets of the same batch are processed by different applications. Figure 12 shows the sustained throughput when running multiple GASPP 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 GASPP when packet scheduling is disabled (GASPP-nosched), and the performance of having multiple standalone applications running on the GPU and the CPU.

We see that the throughput for GASPP 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 11(a)). When adding AES (x=4), performance drops to 8.5 Gbit/s, which is about 1.93× faster than GASPP-nosched. The achieved throughput when running multiple standalone GPU-based implementations is about 16.25× lower than GASPP, due to excessive data transfers.

8 Limitations

Typically, a GASPP developer will prefer to port functionality that is parallelizable, and thus benefit from the GPU execution model. However, there may be parts of data processing operations that do not necessarily fit well on the GPU. In particular, middlebox functionality with complex conditional processing and frequent branching may require extra effort.

The packet scheduling mechanisms described in §5.3 help accommodate such cases by forming groups of packets that will follow the same execution path and will not affect GPU execution. Still, (i) divergent workloads that perform quite lightweight processing (e.g., which process only a few bytes from each packet, such as the FW application), or (ii) workloads where it is not easy to know which packet will follow which execution path, may not be parallelized efficiently on top of GASPP. The reason is that in these cases the cost of grouping is much higher than the resulting benefits, while GASPP cannot predict if packet scheduling is worth the case at runtime. To overcome this, GASPP allows applications to selectively pass network packets and their metadata to the host CPU for further post-processing, as shown in Figure 1. As such, for workloads that are hard to build on top of GASPP, the correct way is to implement them by offloading them to the CPU. A limitation of this approach is that any subsequent processing that might be required also has to be carried out by the CPU, as the cost of transferring the data back to the GPU would be prohibitive.

Another limitation of the current GASPP implementation is its relatively high packet processing latency. Due to the batch processing nature of GPUs, GASPP may not be suitable for protocols with hard real-time per-packet processing constraints.

9 Related Work

Click [19] is a popular modular software router that successfully demonstrates the need and the importance of modularity in software routers. Several works focus on optimizing its performance [10, 11].

SwitchBlade [8] provides a model that allows packet processing modules to be swapped in and out of reconfigurable hardware without the need to resynthesize the hardware. Orphal [18] and ServerSwitch [17] provide a common API for proprietary switching hardware, and leverages the programmability of commodity Ethernet switching chips for packet forwarding. ServerSwitch also leverages the resources of the server CPU to provide extra programmability. In order to reduce costs and enable quick functionality updates, there is an ongoing trend of migrating to consolidated software running on commodity "middlebox" servers [11, 15, 22].

GPUs provide a substantial performance boost to many network-related workloads, including intrusion

detection [13, 24, 26] cryptography [14], and IP routing [12]. Many recent works also deal with GPU resource management in the OS [16, 21]. GPUfs [23] enhances the API available to GPU code, allowing GPU software to access host files directly. Finally, software mechanisms for tackling irregularities in both control flows and memory references have been proposed [29].

10 Conclusion

We have presented the design, implementation, and evaluation of GASPP, a flexible, efficient, and high-performance framework for network traffic processing applications. GASPP explores the design space of combining the massively parallel architecture of GPUs with 10GbE network interfaces, and enables the easy integration of user-defined modules for execution at the corresponding L2–L7 network layers. GASPP has been implemented using solely commodity, inexpensive components, and our development experiences further show that GASPP is easy to program using the C/CUDA language. We have used our framework to develop representative traffic processing applications, including intrusion detection and prevention systems, packet encryption applications, and traffic classification tools.

As part of our future work, we plan to investigate further how to schedule module execution on the CPU, and how these executions will affect the overall performance of GASPP. We also plan to implement an opportunistic GPU offloading scheme, whereby packets with hard real-time processing constraints will be handled by the host CPU instead of the GPU to reduce latency.

Acknowledgments. We would like to thank our shepherd KyoungSoo Park and the anonymous reviewers for their valuable feedback. This work was supported by the General Secretariat for Research and Technology in Greece with a Research Excellence grant.

References

[1] `http://code.google.com/p/back40computing/wiki/RadixSorting`.

[2] `http://l7-filter.sourceforge.net/`.

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

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

[5] CUDA Programming Guide. `http://http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`.

[6] Libnids library. `http://libnids.sourceforge.net/`.

[7] Snort IDS/IPS. `http://www.snort.org`.

[8] ANWER, M. B., MOTIWALA, M., TARIQ, M. B., AND FEAMSTER, N. SwitchBlade: a platform for rapid deployment of network protocols on programmable hardware. In *Proceedings of the ACM SIGCOMM 2010 conference* (2010).

[9] DHARMAPURIKAR, S., AND PAXSON, V. Robust TCP stream reassembly in the presence of adversaries. In *Proceedings of the 14th conference on USENIX Security Symposium - Volume 14* (2005).

[10] DOBRESCU, M., EGI, N., ARGYRAKI, K., CHUN, B.-G., FALL, K., IANNACCONE, G., KNIES, A., MANESH, M., AND RATNASAMY, S. RouteBricks: Exploiting Parallelism to Scale Software Routers. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles* (2009).

[11] GHODSI, A., SEKAR, V., ZAHARIA, M., AND STOICA, I. Multi-resource fair queueing for packet processing. In *Proceedings of the ACM SIGCOMM 2012 conference on Applications, Technologies, Architectures, and Protocols for Computer Communication* (2012).

[12] HAN, S., JANG, K., PARK, K., AND MOON, S. 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).

[13] JAMSHED, M. A., LEE, J., MOON, S., YUN, I., KIM, D., LEE, S., YI, Y., AND PARK, K. Kargus: a highly-scalable software-based intrusion detection system. In *Proceedings of the 2012 ACM conference on Computer and Communications Security* (2012).

[14] JANG, K., HAN, S., HAN, S., PARK, K., AND MOON, S. SSLShader: Cheap SSL Acceleration with Commodity Processors. In *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation* (March 2011).

[15] JOSEPH, D., AND STOICA, I. Modeling Middleboxes. *Network, IEEE 22*, 5 (2008), 20–25.

[16] KATO, S., LAKSHMANAN, K., RAJKUMAR, R., AND ISHIKAWA, Y. TimeGraph: GPU scheduling for real-time multi-tasking environments. In *Proceedings of the 2011 USENIX Annual Technical Conference* (2011).

[17] LU, G., GUO, C., LI, Y., ZHOU, Z., YUAN, T., WU, H., XIONG, Y., GAO, R., AND ZHANG, Y. ServerSwitch: a programmable and high performance platform for data center networks. In *Proceedings of the 8th USENIX conference on Networked Systems Design and Implementation* (2011).

[18] MOGUL, J. C., YALAG, P., TOURRILHES, J., MCGEER, R., BANERJEE, S., CONNORS, T., AND SHARMA, P. API Design Challenges for Open Router Platforms on Proprietary Hardware. In *Proceedings of the ACM Workshop on Hot Topics in Networks* (2008).

[19] MORRIS, R., KOHLER, E., JANNOTTI, J., AND KAASHOEK, M. F. The Click modular router. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles* (1999).

[20] RIZZO, L. netmap: A Novel Framework for Fast Packet I/O. In *Proceedings of the 2012 USENIX conference on USENIX Annual Technical Conference* (2012).

[21] ROSSBACH, C. J., CURREY, J., SILBERSTEIN, M., RAY, B., AND WITCHEL, E. PTask: Operating System Abstractions to Manage GPUs as Compute Devices. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles* (2011).

[22] SEKAR, V., EGI, N., RATNASAMY, S., REITER, M., AND SHI, G. Design and Implementation of a Consolidated Middlebox Architecture. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation* (2012).

[23] SILBERSTEIN, M., FORD, B., KEIDAR, I., AND WITCHEL, E. GPUfs: integrating a file system with GPUs. In *Proceedings of the eighteenth international conference on Architectural Support for Programming Languages and Operating Systems* (2013).

[24] VASILIADIS, G., ANTONATOS, S., POLYCHRONAKIS, M., MARKATOS, E. P., AND IOANNIDIS, S. Gnort: High Performance Network Intrusion Detection Using Graphics Processors. In *Proceedings of the 11th International Symposium on Recent Advances in Intrusion Detection* (2008).

[25] VASILIADIS, G., POLYCHRONAKIS, M., ANTONATOS, S., MARKATOS, E. P., AND IOANNIDIS, S. Regular Expression Matching on Graphics Hardware for Intrusion Detection. In *Proceedings of the 12th International Symposium on Recent Advances in Intrusion Detection* (2009).

[26] VASILIADIS, G., POLYCHRONAKIS, M., AND IOANNIDIS, S. MIDeA: a multi-parallel intrusion detection architecture. In *Proceedings of the 18th ACM conference on Computer and Communications Security* (2011).

[27] VASILIADIS, G., POLYCHRONAKIS, M., AND IOANNIDIS, S. Parallelization and characterization of pattern matching using GPUs. In *Proceedings of the 2011 IEEE International Symposium on Workload Characterization* (2011).

[28] VUTUKURU, M., BALAKRISHNAN, H., AND PAXSON, V. Efficient and Robust TCP Stream Normalization. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy* (2008).

[29] ZHANG, E. Z., JIANG, Y., GUO, Z., TIAN, K., AND SHEN, X. On-the-fly elimination of dynamic irregularities for GPU computing. In *Proceedings of the sixteenth international conference on Architectural Support for Programming Languages and Operating Systems* (2011).