Noname manuscript No.
(will be inserted by the editor)

High-speed TCP flow record extraction using GPUs

Paula Roquero · Javier Ramos · Victor Moreno · Iván González · Javier Aracil

Friday 10th July, 2015 at 15:41

Abstract Traffic analysis is an essential part of capacity planning, quality of service assurance and reinforcement of security in current telecommunication networks. Traffic volume increases with network speed and the analysis of large traffic traces is computationally intensive. The paper presents, for the first time ever, a flow extraction software that allows to obtain complex TCP-aware flow records at 4.4 Millions of packets per second in a single GPU. Such TCP flow records include number of retransmissions and duplicates per flow, whose calculation is very challenging to obtain at high-speed. Our software significantly increases the processing performance of the recently proposed high-speed sniffers based on commodity hardware and demonstrates the advantages of applying massively parallel processing devices for traffic analysis.

1 Introduction and problem statement

A great variety of services are being offered on top of the Internet Protocol (IP), which is inherently best effort. This calls for network monitoring and traffic analysis in order to ensure quality of service and perform capacity planning. Besides, traffic analysis plays a fundamental role in network security.

In the past, network traffic volume was small enough to be managed with a sniffer device. Nowadays, the traffic volume is huge and the sniffers have evolved to sophisticated systems that not only perform packet capture at line rate but also take care of storing and processing the captured packets. In this light, flow record extraction is nowadays a common functionality in network monitoring systems. Note that a network flow is a sequence of packets sharing the same five-tuple, namely: IP source and destination addresses, source and destination ports and IP protocol. Collecting and inspecting traffic at flow level is essential as data is more aggregated and the resulting dataset is smaller than the packet trace itself. Then, once the time interval, hosts, protocols or ports of interest have been identified, the

Departamento de Tecnología Electrónica y de las Comunicaciones, E.P.S.
Universidad Autónoma de Madrid. 28049 Madrid, Spain
javier.aracil@uam.es +34 914972272

corresponding packets from the trace can be extracted and analyzed. Furthermore, there are *flow collectors* deployed in network management centers that collect flow records from different network segments and perform flow record correlation for Quality of Service (QoS) purposes, or issue alarms whenever an anomalous condition happens. We propose a GPU-enabled system can act as a flow record generator for such systems.

The most common flow record standards are Netflow [1] and IPFIX [2]. Each flow record typically includes the source and destination IP addresses (possibly with MAC addresses as well), flow size (both in bytes and packets), duration, and other relevant parameters such as the number of TCP RST (reset) flags detected. Such parameters have different computational requirements. For example, the flow size can be easily calculated by aggregating all the packet sizes in a flow, sequentially as they appear in the trace, even if they are out of order. However, the processing requirements are very stringent when it comes to compare fields from many different packets in the trace.

In this paper, we focus on the TCP performance parameters which are more challenging in terms of processing than other communication protocols. Precisely, this is the case for packet duplicates and TCP retransmissions. The concept of TCP retransmissions is well-known: whereby a TCP sender retransmits unacknowledged packets after a timeout period expires. In turn, a packet duplicate may appear in a trace because the same packet inbound to the Virtual Local Area Network (VLAN) is eventually transmitted outbound from the VLAN. If the whole VLAN is captured then a packet copy will be generated, for example, when a Switched Port Analyzer (SPAN) port of the whole VLAN is set up. However, chances are that the packet is not a byte-per-byte duplicate, but the same packet with its TTL field decremented by one. This is the case of traffic sniffing at both ends of an intermediate router, namely with a layer 3 hop in between. We call the former a "switching" duplicate and the latter a "routing" duplicate [3]. We only deal with switching duplicates in this paper.

We note that the percentage of packets retransmitted per TCP connection is a relevant statistic because the more retransmissions the worse the quality of service, specially for bulk data transfers. On the other hand, it is extremely important to detect duplicates in the trace. If not, severe bias may be introduced in typical traffic statistics such as flow size and duration.

The detection of both TCP retransmissions and duplicates is a very demanding task in terms of processing, as many different packets must be compared to one another. Actually, packets may arrive out of order and the potential duplicate or retransmission may be located totally out-of-sequence. To complicate matters, this task requires a ping-pong buffer that temporarily stores the packets (or the packet fields of interest) from a given flow in order to compare them, the larger the buffer the more the accuracy. Fortunately, such comparison task is well suited for parallelization, as every packet has to be compared with its neighbors in the same TCP connection.

On the other hand, the use of GPUs facilitates the adoption of ad-hoc hardware for traffic capture and analysis. The research community has paid attention to the utilization of flexible and cost-aware solutions based on commodity hardware [4], in contrast to FPGA-based approaches [5] and other commercial solutions [6]. The advantages of using commodity hardware are twofold. First, the amount of investment involved in the purchase of specialized hardware exceeds in several orders of magnitude the price of commodity hardware-based solutions. Second, it provides more flexibility to adapt to any network operation and management task as well as to make the network maintenance easier. As an example of this, we highlight the special interest that software routers have recently awakened [7, 8, 9]. Moreover, the utilization of commodity hardware presents other advantages such as benefiting from energy-saving policies already implemented in PCs, better availability for hardware component updates and flexibility in the implementation of novel measurement techniques.

Precisely, there have been many efforts to improve the packet capturing capability to 10 and 40 Gbps. However, the issue of how to process the traffic, namely how to extract the statistics of interest from the traffic trace, has not deserved the same attention. If the traffic volume is large, it turns out that the processing bottleneck is significant, specially for statistics that involve the comparison of fields from many different packets in the trace, as noted before.

We note that the packet capturing capabilities are usually based in Receive Side Scaling techniques that basically divert the incoming traffic through separate hardware queues, which are subsequently handled by different CPU cores in parallel [10]. By distributing the traffic between different queues, the throughput per queue decreases, thus alleviating the load per core in the packet capture. As a result, as much as 14.7 Millions of Packets per Second (Mpps) can be captured, which is the case for a fully utilized 10 Gbps unidirectional link with minimal-size packets (64 bytes).

As attractive and cost-effective commodity hardware solutions may be, we note that *the fundamental limitation lies in the number of cores*. In fact, most available solutions [11, 12, 13] consume up to 12 cores for a line rate of 10 Gbps and typically 8 cores just for the packet capture, all of them with a very high utilization. Most importantly, we note *that this is the number of cores occupied per network interface*. Typically, there are several active interfaces per probe because several network segments have to be measured concurrently and some packet tracking analysis between them may be performed, i.e. cores are highly utilized for capturing and there is little processing capacity left for processing packets. Furthermore, not only the number of cores involved is important for the traffic processing but also the availability of memory and hard disk. Concerning memory, we note that the processing cores consume memory for packet capturing, because large buffers are needed to absorb the peaks. In terms of hard disk, we note that the drives are typically loaded due to packet storing at high-speed.

In this paper, we study how to obtain TCP flow records by means of massive parallel programming in GPUs, which is a packet processing task typically handled by CPU cores not devoted to packet capturing. We focus on the traffic parameters within a flow that are computationally hard to obtain, such as retransmissions and duplicates. The main advantage is that *the GPU increases the processing density of the commodity hardware*, namely it does not fully utilize additional cores. Furthermore, the throughput obtained is around 4.4 Mpps in a single GPU. This suffices for a real traffic scenario of 10 Gbps with an average packet size of 500 bytes, which translates into an approximate rate of 2 Mpps. Furthermore, the GPU also increases the memory density of the commodity hardware due to the internal memory, which can be used to absorb peaks at higher rates.

Our findings show that the GPU processing power matches that of a four-core CPU system working in parallel assuming linear scalability from the single-core solution as basis. In a highly dense commodity hardware system such cores can be re-used for other packet or flow-record processing tasks, such as to run anomaly detection routines. Overall, the commodity hardware system processing capabilities are greatly enhanced and the resulting GPU-enabled system becomes a real workstation for traffic processing at very high speed, beyond mere traffic capturing and storing.

The paper is structured as follows. First, we introduce the state of the art and explain what we mean by duplicate and retransmission, in order to understand this important part of our parallel processing algorithmics. Then, the parallel algorithm in-depth description, along with implementation details, follows. Afterwards, we present the performance evaluation, both in terms of accuracy and throughput. Finally, we present the conclusions and future work. Nevertheless, before we proceed with the technical agenda, let us briefly present some introductory material about GPUs, for the sake of completeness.

2 State of the art

2.1 GPU architecture and programming model

A GPU (Graphics Processing Unit) is a hardware for graphic rendering which can be found nowadays almost on every PC and also in some smartphones or tablets. Due to its massively parallel architecture, the GPUs can run trillions of instructions per second for both graphical and non-graphical applications. A GPU that is used for non-graphical applications is commonly known as GPGPU (General-Purpose Graphics Processing Unit). As it is done in the literature, we will use the the term GPU to refer to GPGPU. The performance reached by GPUs makes this hardware amenable for High Performance Computing (HPC) clusters [14]. In fact, some supercomputer vendors have included GPUs inside their parallel computer blades. An example can be the SGI UV and the Cray XK7 supercomputer, which both include NVIDIA GPUs. Additionally, GPUs have been also used in other research articles about network traffic classification [15, 16] mainly focusing on pattern matching and security.

There are many different GPU architectures and models, NVIDIA and AMD being the most popular. Much research and testing have been performed to evaluate which technology gives a higher performance [17]. Given that NVIDIA's CUDA language provides, in general, a higher abstraction level than other GPU-oriented languages, we have opted to use NVIDIA and its CUDA programming technology.

Typically, GPU devices are external to the CPU. CPU and GPU connect and communicate through PCIe (Peripheral Component Interconnect Express), which entails memory copies from the host to the GPU. This fact can make a GPU very inefficient if the data copy takes much time compared to the processing time. The NVIDIA's GPU architecture consists of a large number of SP cores (Streaming Processors), grouped into SMs (Streaming Multiprocessors). The SPs are small processors able to perform integer operations and simple-precision floating-point operations. The SM also contains double-floating point units, several registers, a level 1 cache and a shared memory. Each SM shares these resources among its SP cores. In a similar way, every SM shares a L2 cache and the global memory between the others SMs. In the NVIDIA's Fermi architecture we can find up to 16 SMs each with 32 SP cores [18]. In the newer Kepler architecture, it is possible to find up to 15 SMs each with 192 SP cores and 64 DPUs (double-precision units) [19]. Figure 1 shows an overall design of a NVIDIA GPU architecture.

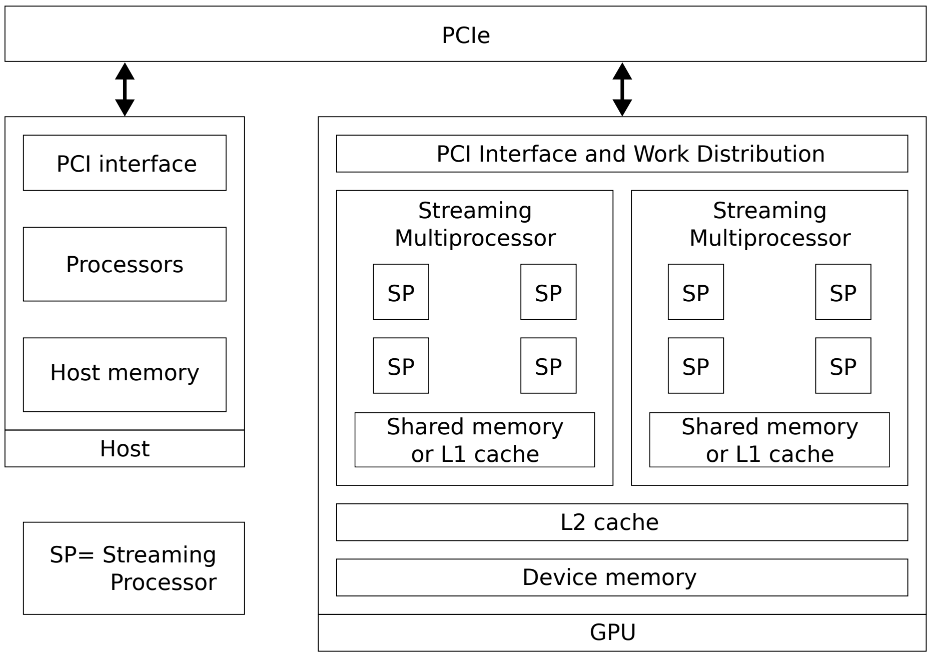

Fig. 1 NVIDIA GPU General Architecture

The CUDA programing model enables the use parallel functions that are executed on the GPU, which are called CUDA kernels. Each kernel can be executed in parallel with other kernels if the device has the required resources available. Such kernel is launched on a grid, that is composed by a set of blocks (which can be defined as 1, 2 or 3-dimensional). Each block is composed by a set of threads (that can be also defined as 1, 2 or 3-dimensional). In turn, each thread runs on a SP processor and each block is executed on a SM. Due to the architecture previously explained, different threads within the same block can share memory very efficiently (without having to access global memory).

To obtain the best performance, the programmer must ensure that the thread execution time may not diverge in excess, as this would create serialization of execution between threads of the same block. The programmer must take into account the total number of threads and its distribution between blocks. Furthermore, the programmer should also consider the amount of shared memory used by each thread and other possible architectural considerations. Figure 2 depicts how the CUDA programming model is organized. More information about CUDA programming model can be found in [20].

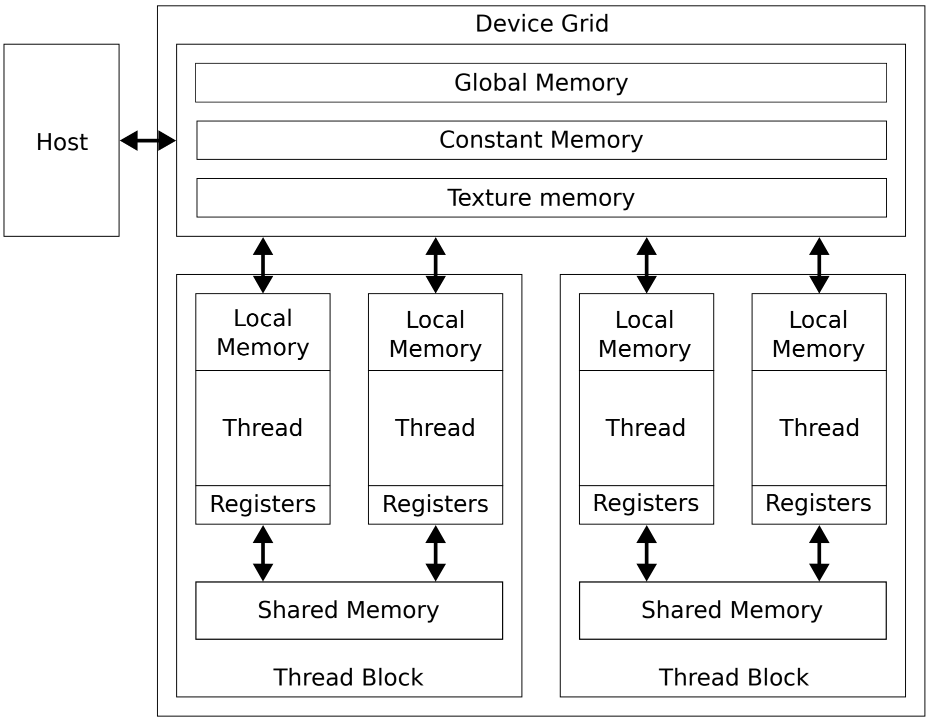

Fig. 2 CUDA Programing Model

2.2 GPU-based network processing

As it turns out, no previous work was found that deals with flow record extraction using GPUs, but there are a few references on processing network traffic with GPUs.

Interestingly, we note that [11] investigated the use of GPUs for high-speed packet routing. The GPU is used to implement high-speed routing and pattern matching for anomaly detection, with excellent results. Even though the paper does not provide flow extraction (all the processing is performed at the packet level only) it demonstrated that the packets can be swiftly relayed from the CPU to the GPU at very high-speed (40 Gbps with small 64 bytes size packets).

On the other hand, in [21] authors show how to use a GPU for packet filtering using the Berkeley Packet Filter (BPF). A performance comparison with a CPU was carried out and the CPU turned out to be actually faster. In this case, the data copy to the GPU did not pay off for the speedup achieved by the filtering process.

Other authors have focused on flow record extraction using FPGAs [22, 23, 24, 25]. This is a completely different technology that allows to obtain an impressive line rate but at the expense of a much larger development time and cost. Interestingly, none of the authors provide statistics for the number of TCP retransmissions and duplicates in the flow record. This is because FPGAs lack memory space, which is essential to detect retransmissions and duplicates.

NetGPU is a framework designed to assist in traffic analysis using GPUs with CUDA. Both the design and implementation are described in [26] and the code is available under free software license in Google Code [27]. This framework provides capabilities to read packets from several sources using PacketFeeders. Then, packets are distributed by an *analyzer* routine to several *analysis* processes, which are in charge of processing the packet in order to extract the desired statistics in the GPU. Then, the framework user implements the analysis routine to process the packets.

The framework defines buffers of fixed size for the packets. Some of these buffers can be joined before the data is sent to the GPU for processing. Actually, the framework documentation does not specify if there is a limit to the number of packets that can be accumulated. On the other hand, we note that even though this framework is interesting for packet processing it is not maintained at the present time and no performance figures are reported.

Since we employ hash tables to match packets with their corresponding retransmission or duplicate in the GPU, let us briefly review the optimized hash library *cudpp* by [28] and discuss the limitations for our current work. Such hash table has a high performance and achieves the insertion rate of 5 million key-value pairs in 35.7 ms and the access time to all such pairs is 15.3 ms. To achieve this performance the hash table is implemented as a mix of sparse perfect hashing and *cuckoo* hashing, which makes use of the faster shared memory in the GPU to speed up its creation.

The high performance of this hash table implementation is mainly due to the use of a 32-bit key and value, namely:

(i) Both the key and value can be written to memory in the same atomic access.
(ii) Small-size keys and values allows to employ the fast shared memory, which is also very scarce, to speed up the creation of the table by means of *cuckoo* hashing.

However, we do require larger keys and values for the detection of retransmissions and duplicates. More specifically, the key must be 128 bits long and the value uses 192 bits, as will be discussed in the implementation section. This prevents the adoption of the *cudpp* hash table. Furthermore, the variant of the algorithm described in the paper [28] would not yield the desired performance increase. We note that the larger key and value impede the use of the GPU's shared memory, and this is key to obtain fast *cuckoo* hashing.

2.3 Packet duplicates and TCP retransmissions

As noted in the previously, the estimation of retransmissions and duplicates within a TCP flow is demanding in terms of processing requirements. In this section we analyze how to estimate the count of retransmissions and duplicates in a flow. We note that there is tradeoff between performance and accuracy, namely it is not possible to detect all retransmissions and duplicates and keep line-rate operation at the same time.

We consider that a TCP segment containing data is a retransmission if the following TCP header fields are found in a previously arrived segment in the traffic stream: source and destination IP addresses, source and destination TCP ports and TCP sequence number. In what follows, such array will be called a *five-tuple*.

The above definition entails that only five-tuples have to be compared to detect retransmissions, thus saving memory space in the GPU, which is scarce compared to the host memory. However, we do note that some retransmissions will not be detected (false negatives). Furthermore, some segments can be mistakenly detected as a retransmission (false positives). We will carefully review what are the false negative and false positive cases and come up with an algorithm that minimizes the occurrence of false positives. In the next section, we will perform a trace-driven analysis to quantify the overall accuracy of our algorithms.

In what follows, we refer to TCP segments by their transmission order. Based on the analysis of the TCP protocol we present a taxonomy of the of the situations where false negative may arise:

(i) When an already sent TCP segment suffers fragmentation at the TCP level, only the first fragment is detected as a retransmission because the other fragments have different sequence numbers.

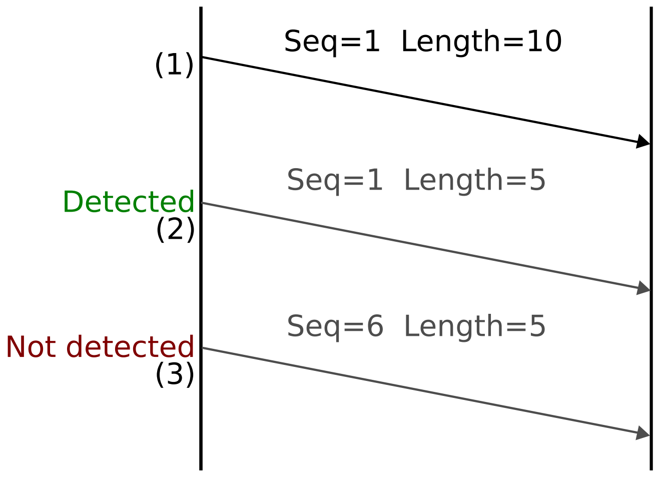

Fig. 3 TCP segmentation

In Figure 3 segment 1 is divided into two segments, 2 and 3. Then, segment 2 is detected as a retransmission but not segment 3 as it features a different sequence number that was not seen before in the traffic stream.

(ii) When two or more already sent segments are joined together only the first segment is considered a retransmission.
In Figure 4 segments 1 and 2 are combined into segment 3, but segment 2 is not detected as a retransmission because its sequence number has not appeared before in the traffic stream.

(iii) When TCP Keep-alive segments (segments without data) are captured the first one is not considered as a retransmission because its sequence number was not seen before in the traffic stream. An example is shown in Figure 5.

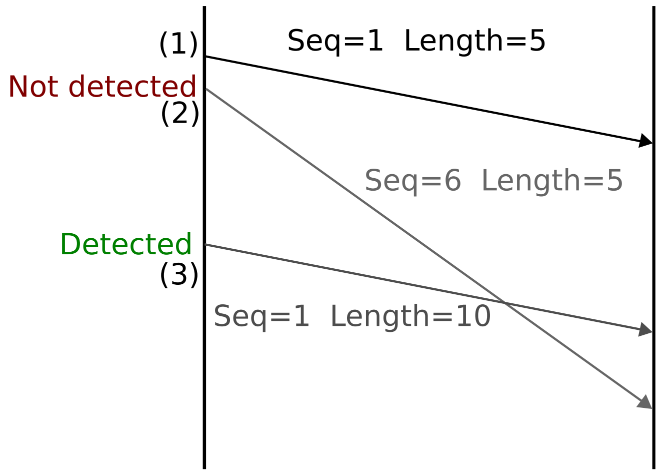

Fig. 4 Joined segments

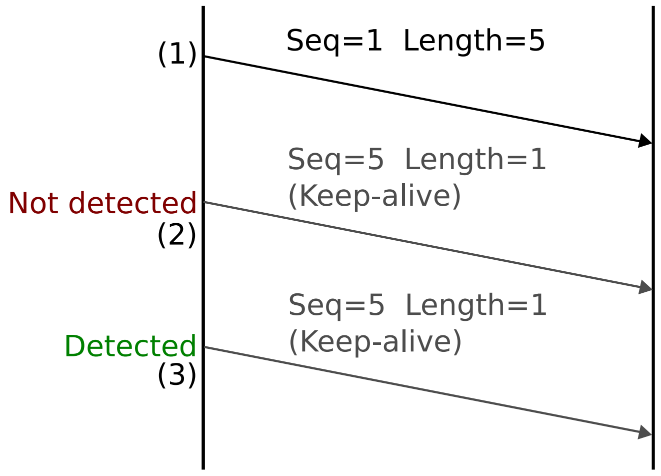

Fig. 5 TCP Keep-alive

(iv) Segments with common data.
In Figure 6, a segment with sequence number equal to 11 and length equal to 5 was lost between segments 1 and 3. Afterwards, the lost segment was retransmitted (4) with more data, and it should be considered a retransmission because of the overlapped retransmitted data. However, it cannot be detected because the sequence number was not seen before in the traffic stream.

The latter cases constitute a small percentage of all the retransmissions, as will be shown in the performance evaluation section. On the other hand, chances are

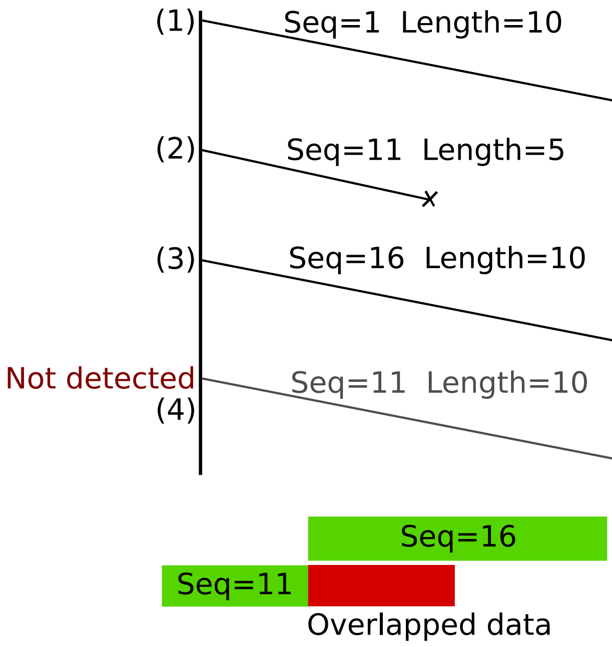

Fig. 6 Common data

that a non-retransmitted segment is tagged as retransmission, namely a false positive, if the connection is reused (*reincarnation*) and some segments are transmitted with the same sequence number than others in the previous connection.

Even though these cases are not frequent, we further reduce the probability of a false positive by setting a maximum time between retransmissions, beyond which the potential retransmission will be discarded. As a result, our algorithm does not require knowledge of the ISN (Initial Sequence Number), which is normally used to detect reincarnations. In any case, we will provide a throughout performance evaluation in Section 4, that includes an estimation of the percentage of false negatives and positives.

Concerning duplicates, we note that the five-tuple must be equal between two segments, just like a retransmission, but additionally the IP packet identifier (IPID) must be equal. As explained in the introduction section, we only take into account the *switching* duplicates [3].

For each flow exported, we obtain other parameters than the number of retransmissions and duplicates, such as a counter for the number of SYN, FIN and RST flags and also for the number of segments that announce a TCP window size equal to zero but do not have the RST flag set. This is useful to detect congestion in the receiver side (receiver's window is exhausted). Note that the RST segments are not included in such counter because typically a RST segment announces zero window size to stop the transmitter on purpose. Namely, the RST flag does not indicate congestion at the receiver.

Lastly, the flow size in bytes and duration are obtained by means of the sequence numbers and timestamps of the SYN and FIN segments. This is a usual technique which has also been used in other works [29]. Note that if two different connections, possibly from a reincarnation, have the same source and destination IP address and TCP ports, then it is not possible to associate each segment to each particular connection. To prevent biased data we do not provide this counter if more than one SYN or FIN per connection are detected. In the datasets we have considered, this only happens in 0.84% of the existing flows.

3 Algorithmic design and implementation

This section describes the operation and implementation of the proposed algorithm to: detect retransmissions and duplicates; count SYN, FIN and RST flags, and zero-window announcements; and set the timestamp and sequence number of SYN and FIN segments. For simplicity, the serial-CPU version is described first. Then, the multi-thread version and the real-time packet capture and processing version will be presented. Note that both versions rely on GPU computing for duplicate and retransmission detection, but they differ on the way work is distribute on the CPU side.

3.1 GPU + serial-CPU implementation

The algorithm takes as input the traffic source, the obtains the TCP flow data and, finally, dumps the results to a file. This process is shown in Figure 7.

We distinguish the following steps:

1. *Fetching the packets*: The data for each TCP segment is read from a traffic source and stored in the *Host segment list*.
2. *Copying to the GPU*: The data for each TCP segment is copied from the *Host segment list* to the *GPU segment list*. These two buffers are identical.
3. *Inserting into the segment hash table*: The data from the TCP segments is inserted into the *GPU segment hash table*. Then, the number of retransmissions for each segment is calculated.
4. *Inserting into the flow hash table*: The segment data in the *GPU segment hash table* is inserted in the *GPU flow hash table* to obtain the number of segments and retransmissions for each connection.
5. *Copying to the host*: The data for each flow is copied from the *GPU flow hash table* to the *Host flow hash table*. Both buffers are identical.
6. *Copying to sort buffer*: The *Host flow hash table* is read in order to copy the occupied buckets to one of the two *Host sorting buffers*.
7. *Sorting and writing*: When a *Host sorting buffer* is full the flows are sorted by its four-tuple and the full buffer is written to a *Result file*.
8. *Combining results*: A secondary process reads all the ordered *Result files* and generates a *Combined results* file. A *Min heap* is used so that only one flow from each file needs to be in memory at any time.

To minimize the use of GPU memory only certain fields of the TCP segment are copied to the *Host segment list*, namely:

– *Source and destination IPs.*

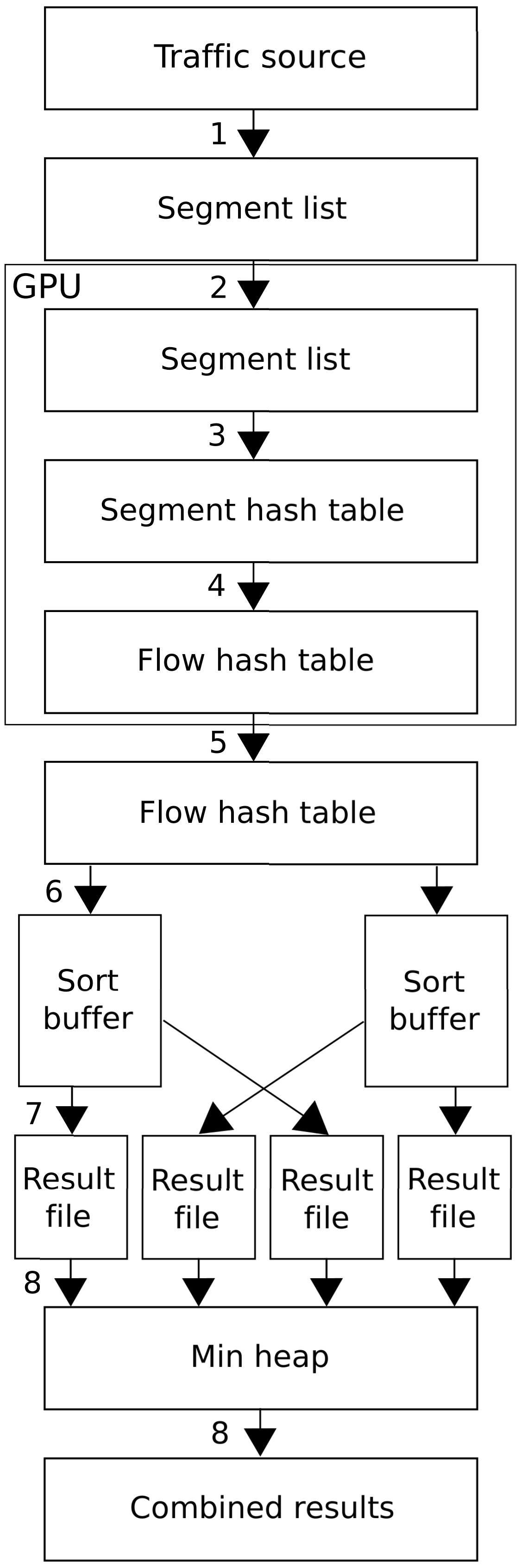

Fig. 7 Data flow

– *Source and destination ports.*
– *Sequence number.*
– *Timestamp.*
– *Payload size.*
– *IP identifier.*
– *Internal flags about the type of segment.*
– *TCP flags.*

Among those fields, only source and destination IPs and TCP ports plus TCP sequence number are used to build the key assigned to each segment. The hash distribution policy uses this key value to distribute the data among the table's entries using the MurmurHash2 hash function. The rest of the fields will be stored on each table entry.

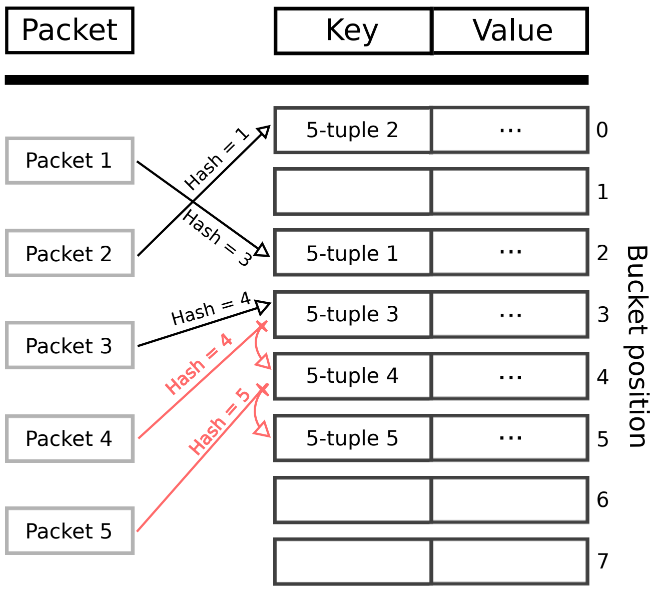

Fig. 8 Hash table structure

From the point of view of the GPU, the first stage of the algorithm performs the insertion of TCP segments into a hash table (Figure 8) implemented inside the GPU using the five-tuple as the key. Then, the segments are read from the source file in blocks with a size that decreases the number of collisions, and copied to the global GPU memory. After that, the kernel to find retransmissions and duplicates is launched in the GPU (*FindRetransmissionsKernel*). This kernel uses one warp per block (32 threads) and a block per segment. The reason for this kernel configuration is to avoid deadlocks caused when part of the threads in a SIMT processor are locked in a branch that must be unlocked by the remaining threads in the other branch. This also presents the benefit of doing coalescent accesses to memory.

On the other hand, each block executes the steps shown in Listing 1.

Listing 1 Segment insertion algorithm

```
for seg in segment_list:
  if seg == any of the previous n segments:
    seg.duplicate = true;
  seg_hash = hashFun(seg.5-tuple);
  seg_bucket = seg_hash % seg_hash_table.size;
  while seg is not inserted:
    if seg_bucket.used == false:
      seg_bucket.used = true;
      seg_bucket.5-tuple = seg.5-tuple;
      if seg.duplicate == true:
        seg_bucket.dup_count = 1;
      else:
        seg_bucket.dup_count = 0;
        seg_bucket.original_found = 1;
      seg_bucket.flag_count = seg.flag_count;
    else if seg.5-tuple == seg_bucket.5-tuple:
      if seg.duplicate == true:
        seg_bucket.dup_count += 1;
      else if seg_bucket.original_found == 1:
        seg_bucket.retx_count += 1;
      else:
        seg_bucket.original_found = 1;
      seg_bucket.flag_count += seg.flag_count;
  else:
    seg_bucket = (seg_bucket + 1) % seg_hash_table.size;
```

Due to the fact that the segments to be analyzed are split into more than one memory block, it is necessary to design a mechanism to detect retransmissions and duplicates in two adjacent memory blocks. To detect retransmissions, once a new block is loaded into the hash table, only the segments from the next block with a five-tuple already present in the hash table are inserted. Then, results are saved, the hash table is emptied and the remaining segments in the block are inserted normally. This process is shown in Figure 9. To detect duplicates, a buffer is used to save the last n segments of a block. Then the first n segments of the next block are compared to the previous stored segments in the buffer.

We note that such techniques eliminate border effects that happen whenever a duplicate or retransmission is present at the beginning of a memory block and the corresponding original packets are located at the end of the previous block. However, it does not provide cure against false negatives produced if the original packet is in the previous block but not at the end.

A different hash table is used for flow-related information, using source and destination IPs and TCP ports as key to the table. The values stored in this table are:

- *Number of segments in this flow.*
- *Number of retransmission in this flow.*
- *Number of duplicates in this flow.*
- *TCP flag counters.*
- *Timestamp.*
- *SYN and FIN sequence numbers.*
- *SYN and FIN timestamps.*
- *Payload size of the FIN segment.*

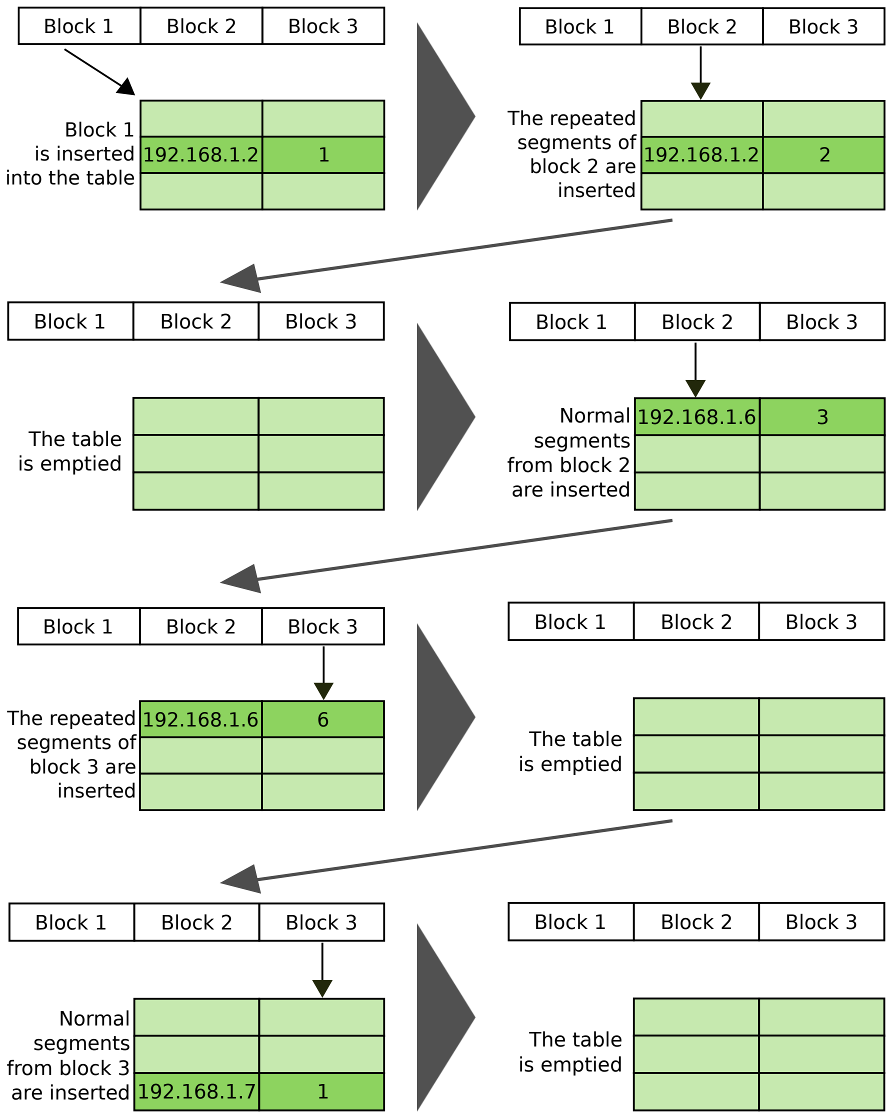
Block 1
Block 2
Block 3
Block 1 is inserted into the table
192.168.1.2
1
The repeated segments of block 2 are inserted
192.168.1.2
2
The table is emptied
Normal segments from block 2 are inserted
192.168.1.6
3
The repeated segments of block 3 are inserted
192.168.1.6
6
The table is emptied
Normal segments from block 3 are inserted
192.168.1.7
1
The table is emptied

Fig. 9 Insertion

The counters of segments, retransmissions, duplicates, flags (SYN, RST, FIN) and zero-window announcements are increased with each segment belonging to the flow. Such counters are stored in a single 4-byte variable, using only one byte per counter. This is done to reduce the hash table bucket size and thus improve GPU memory usage. In the performance evaluation section we will discuss the benefits of this technique in terms of accuracy. To calculate the duration and length of each flow, the sequence numbers and timestamps of the SYN and FIN segments must be stored. The FIN data size is also stored to calculate the total flow size. The drawback of this approach is that when several SYN or FIN flags are found in a flow, the length and duration cannot be calculated accurately, so the data is set as invalid, as noted before.

Another important design consideration is that the GPU memory is not large enough to combine all the segments from long-lived flows, so a CPU post-processing is necessary to combine all the data from a flow. The flow data is copied from GPU to host, where the data is accumulated until a buffer is full. By using two different buffers we note that results can be copied in one buffer while the other buffer is being sorted.

Once the data is back to the CPU, it traverses a min heap, which is implemented to get the smallest quadruples among the sorted files and aggregate the flow data when the quadruples are equal—adding the number of packets, retransmissions, duplicates and flow registers. The CPU thread uses then the sequence numbers and timestamps of the SYN and FIN segments to calculate the length and duration of each flow. The final result is a file with all the connection data sorted by quadruple starting with the smallest one. The whole process is shown in Figure 7.

3.2 GPU + multi-CPU implementation

Now that the basic algorithm has been explained as a serial process in the host, the real implementation using *pthreads* will be described. The use of *pthreads* increases the performance mainly because the GPU is calculating retransmissions while the host reads more packets from file. The program uses 4 threads synchronized with mutexes. Such threads read from file, insert segments, copy results to a buffer and write sorted results to disk. The threads are synchronized to protect the memory in each pipeline stage that appears in Figure 7. Note that, although there are 4 concurrent threads, the average number of used cores is 1.5.

Figure 10 shows the threads involved in the execution, linking the tasks with the steps described in section 3.1. The description follows, from left to right in the figure:

i The first thread executes steps 1 and 2 of the data flow. This thread reads segments from *the trace file or network*, filters them and stores the necessary fields in the *host segment list.* Once the GPU has processed all the segments from the previous block, the segments are copied from host memory to the *GPU segment list.* The copy is asynchronous, so new segments can be read concurrently with the copy.

ii The second thread executes steps 3, 4, and 5 of the data flow. This thread waits until the data in the *GPU segment list* is made available by the first thread. When the data is available the repeated segments are inserted into the *GPU segment hash table* and the data is joined for each flow in the *GPU flow hash table.* Once the segments in the *GPU segment hash table* are not longer needed, the table is emptied and the remaining segments in the *GPU segment list* are inserted. A mutex that prevents the beginning of the copy from the first thread is unlocked once the data in the *GPU segment list* is no longer needed. Before continuing, the thread waits in a mutex until the previous flow data in the *host flow hash table* has been copied to the *host sort buffer*. Then, the data is copied from the *GPU flow hash table* to the *host flow hash table* and other

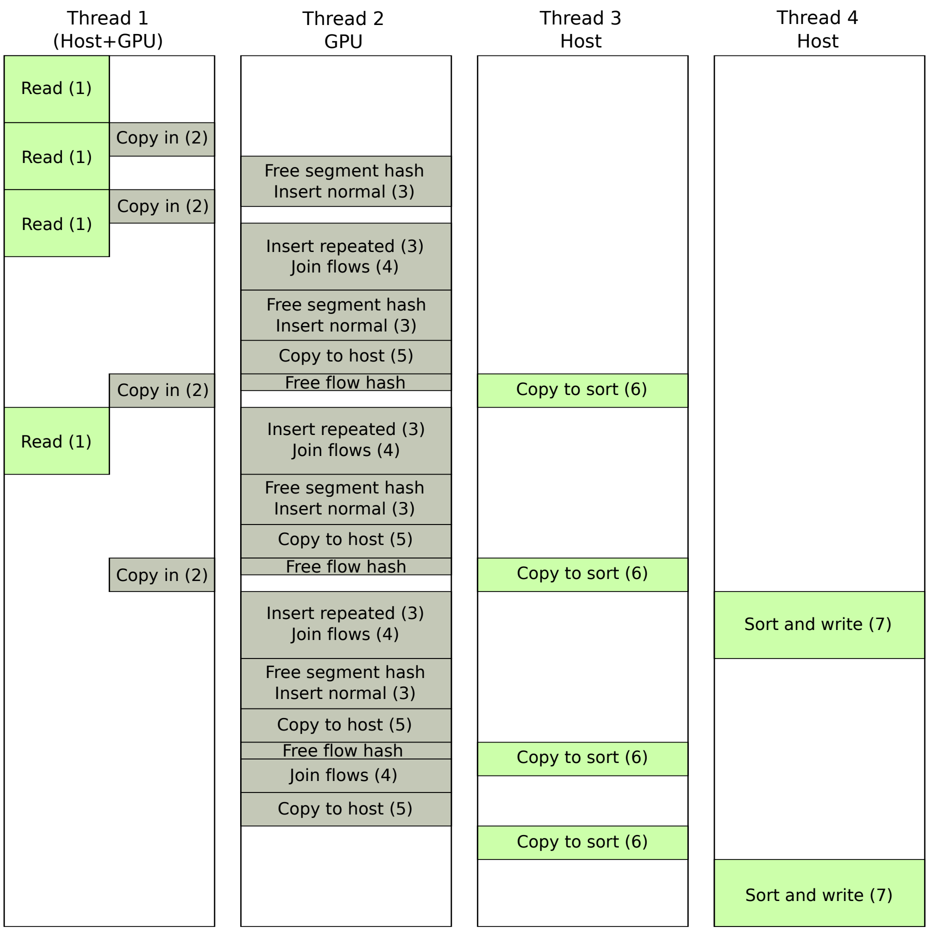

Fig. 10 Time diagram showing *pthreads*

mutex is unlocked so the third thread can start copying the current data to the *host sort buffer*. Finally, the *GPU connection hash table* is emptied.

iii The third thread executes step 6 of the data flow. This thread waits in a mutex until there are results available in the *host connection hash table*, then copies the data to the *host sort buffer* and unlocks the mutex that allows copying more results from the GPU to the host. When the *host sort buffer* has enough data, a mutex is unlocked so the fourth thread can sort the results. Meanwhile this thread can copy the results to other different *host sort buffer* so the sorting does not become a bottleneck.

iv The fourth thread executes step 7 of the data flow. This thread waits in a mutex until the *host sort buffer* is full, sorts the results and writes them to the *result file*. Then another mutex is unlocked so the *host sort buffer* can be used again to store results.

3.3 Obtaining the data from the network

To process packets directly from network no changes are necessary. The HP-CAP custom driver, capable of capturing and storing traffic at 10 Gbps has been used [30] for this purpose. This driver provides simple framework that allows instantiating user-level network processing applications that can benefit from its high-throughput design. Traffic consumers can attach to the driver and analyze or store the traffic concurrently. Note that the speed of the system is that of the slowest consumer. Furthermore, the driver includes several internal buffers to shape traffic peaks for the consumers. However, after some tests to evaluate the performance, it was discovered that the driver's buffer was filled during the copy. This issue was due to the data copy to the GPU, which was synchronous (the CPU was blocked during the copy). To solve this problem, asynchronous functions to copy data to the GPU were used. As a result, packets can be copied to the GPU at any time while being received, thus avoiding blocking.

4 Performance evaluation

We evaluated the software in a computer with a 12 core *Intel Xeon X5650*, 70 GB of memory, a *nVidia Tesla C2075* and an Intel 82599 10Gb Ethernet card. We used this computer because it was the only one with access to production data and not running a critical system. The GPU is similar to the *nVidia Tesla M2070*, which is approved for use in our production servers.

First, we evaluate the accuracy of the algorithms, by taking several traffic traces as inputs and comparing results with a ground-truth serial program which was has been proven not to generate neither false positives nor false negatives for the retransmission and duplicate detection. Note that this program is oriented to obtain as accurate as possible, being its performance damaged in return. The flows that showed discrepancy between our GPU and ground-truth analysis were analyzed by means of *Wireshark* tool to find the divergence. Once the accuracy level was established we focused on the overall throughput.

4.1 Datasets

Table 1 Capture file info

Trace	Packets	TCP Segments	Data Segments	Switching duplicates	Retx
A	188,423	161,913	89,081	41,426	1,691
B	88,423	61,990	35,360	0	1,686
C	100,000	99,923	53,721	41,426	5
D	200,000,000	167,000,000	109,000,000	-	346,060
E	310,000,000	258,000,000	168,000,000	-	536,393

Regarding datasets, in Table 1 the traces used for evaluation along with its relevant information are presented. In the case *C* file, the dataset has been used for testing accuracy in duplicate detection. On the other hand, *B* file has been used for testing accuracy in retransmission detection. Dataset *A* consists of the merge of the two previous traces and has been used to test the accuracy of both duplicates and retransmission detection. The large dataset *D* has been used to carry out random sampling of connections in order to determine the root cause of inaccuracies in retransmissions' detection, as will be explained later. Finally, *E* dataset has been used to measure the time spent in each phase of the pipeline.

4.2 Duplicates/retransmissions classfication accuracy

Regarding detection of duplicates, we note that neither false positives nor negatives were observed. We recall that a VLAN SPAN port produces duplicates, which will be separated by the transit time through the router, that is typically very small. Then, original and duplicate packets are close together in the trace, thus falling either into the same memory block or in the border between subsequent blocks. As a result, duplicates can be easily detected by the GPU. The same does not apply to retransmissions, which are the focus of the next section.

Concerning TCP retransmissions, Table 2 shows the ratio of false negatives, that is, flows with true retransmissions not detected by the GPU . for each of the dataset files. Table 3 shows the same parameter, false negatives, regarding the total number of packets. Importantly, flows without ISN were not considered due to limitations on the benchmarking program as this data is required to detect retransmissions. This is the case for flows with missing SYN packets, possibly due to a capture error. We note that a TCP connection requires both SYNs from the client and server in order to be established.

Table 2 False negatives (flows)

	A	*B*	*C*	*D*
Flows with retransmissions	531	528	3	304,229
Flows with retransmissions (ground-truth)	604	601	3	346,060
Flows with false negatives	73	73	0	41,832
Percentage of false negatives	12.09%	12.15%	0%	12.09%

Conversely, Table 4 shows the ratio of false positive flows, that is, flows with false retransmissions detected by the GPU. Similarly, Table 5 shows the ratio of false positives regarding the total number of packets for each dataset.

First, we note that duplicates are not mistakenly confused with retransmissions and the other way around, because the dataset with duplicates and retransmissions shows the same figures than the datasets with only retransmissions and duplicates respectively. Second, there are very few false positives (less than 0.001%) and a significant ratio of false negatives. Consequently, we turn our attention to the evaluation of false negatives. We argue that retransmitted packets are not detected as they are placed in different consecutive memory blocks sent to the GPU. Thus,

Table 3 False negatives (packets)

	A	*B*	*C*	*D*
Number of retransmissions	1,691	1,686	5	1,699,364
Number of retransmissions (ground-truth)	1,775	1,770	5	1,786,758
Number of false negatives	84	84	0	87,414
Percentage of false negatives	4.73%	4.75%	0%	4.89%

Table 4 False positives (flows)

	A	*B*	*C*	*D*
Flows with retransmissions	531	528	3	304,229
Flows with retransmissions (ground-truth)	531	528	3	346,060
Flows with false positives	0	0	0	1
Percentage of false positives	0%	0%	0%	$2.89 * 10^{-6}$%

Table 5 False positives (packets)

	A	*B*	*C*	*D*
Number of retransmissions	1,691	1,686	5	1,699,364
Number of retransmissions (ground-truth)	1,691	1,686	5	1,786,758
Number of false positives	0	0	0	20
Percentage of false positives	0%	0%	0%	$1.12 * 10^{-5}$%

the retransmissted packet is transferred to the GPU when the original packet is no longer present. Figure 11 shows the percentage of retransmissions not detected (regarding the total number of packets) for the *D* dataset versus the size of the memory chunk transferred to the GPU.

The figure shows that the detection ratio increases (conversely, the false negatives' ratio decreases) as the memory block size increases, reaching a lower bound of around 5%.

4.3 Why some retransmissions are not detected?

In Section 2.3 we noted that some of the retransmissions could not be detected due to the detection algorithm adopted, which only takes into account the TCP segment's five-tuple. We denote such retransmissions by *structural* retransmissions, since they cannot be detected whatsoever. However, there are other retransmis-

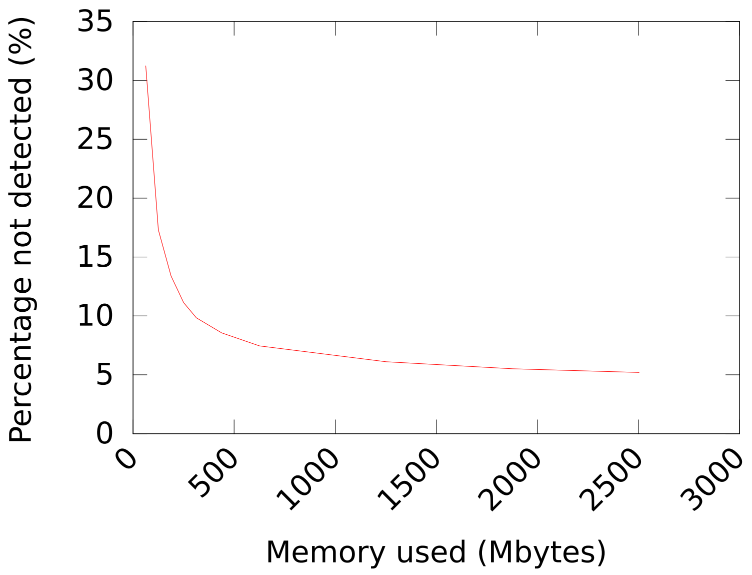

Fig. 11 Ratio of undetected retransmissions versus memory block size

sions which are not structural and could not be detected either, given the limited size of the buffer.

Recall that the packet trace is relayed to the GPU in memory blocks that fit into the GPU internal memory. If the original packet and retransmission do not fall within the same or next block then the retransmission cannot be detected. We denote such retransmissions by *split* retransmissions, as the connection they belong to is split into two different memory blocks.

Let $\mathcal{A}$ refer to the event that a given retransmission is not detected (i.e. a false negative) and let $\mathcal{A}_{struct}$ and $\mathcal{A}_{split}$ refer to the event that a given retransmission is not detected because it is either structural or split, respectively. Then,

$$\mathcal{A} = \mathcal{A}_{struct} \cup \mathcal{A}_{split} \tag{1}$$

$$\mathcal{A}_{struct} \cap \mathcal{A}_{split} \neq \emptyset \tag{2}$$

and we wish to have an estimation of $\mathbb{P}\left(\mathcal{A}_{struct}\right)$ and $\mathbb{P}\left(\mathcal{A}_{split}\right)$, noting again that structural retransmissions cannot be detected whatsoever. On the contrary, split retransmissions could be detected provided that more memory space in the GPU was available or, alternatively, that the traffic trace is demultiplexed (for example, by source IP subnetwork) in different GPUs working in parallel. We also assume that the $\mathbb{P}\left(\mathcal{A}_{struct} \cap \mathcal{A}_{split}\right)$ is very small as it corresponds to the probability of a structural retransmission that is also split into two different memory blocks.

In order to isolate split retransmissions from structural retransmissions we performed a random sampling of flows present in the D dataset as follows. Let N be the total number of flows in a dataset. Then, we randomly sampled a number of flows equal to n *such that their packets fit within a GPU memory buffer.*

Note that this ensures that undetected retransmissions (false negatives) are *all due to structural retransmissions.* Therefore, such sample is valid to obtain the probability $\mathbb{P}(\mathcal{A}_{struct})$ by means of the estimate $\hat{p}$, which is equal to the ratio of retransmissions in the sample.

We note that the confidence intervals for such proportion are given by the Cochran approximation as follows:

$$\hat{p} \pm \lambda_\alpha \sqrt{\frac{\hat{p}(1-\hat{p})}{n-1}\frac{N-n}{N}} - \frac{1}{2n} \tag{3}$$

whereby λ_α is the corresponding percentile of the standard Gaussian distribution with significance level α.

A random sample of 100,000 flows with 481,000 retransmissions was used for the analysis. It was found that 4.31% (confidence interval [4.24, 4.38]) of the retransmissions were structural. Note that this is close to the 5.14% ratio obtained for the 4 GB memory block case and it shows that the probability of split retransmissions has a lower bound of around 0.82%, and, as we deem $\mathbb{P}(\mathcal{A}_{struct} \cap \mathcal{A}_{split})$ small it must be close to $\mathbb{P}(\mathcal{A}_{split})$.

Furthermore, in a typical use case we are only interested in flows with a significant number of retransmissions (for example, connections with more than 5 retransmissions and 5% of the packets being retransmitted). In this case, only 0.31% of the flows have this problem. The statistical analysis shows that in 0.3% (confidence interval [0.25, 0.34]) of the flows, the false negatives were caused by structural retransmissions. Interestingly, as the number of per-flow retransmissions grows so does the likelihood of placing them into the same GPU memory block, which dramatically decreases the probability of split retransmissions.

4.4 Flow records accuracy

In the previous section, we have discussed that memory size is key to improve accuracy, as the larger the memory size the better chances to fit the original packet and retransmission in the same memory block. However, memory size is fixed in the GPU board and cannot be increased arbitrarily. That is why we follow the approach of compressing both packet and flow records in order to fit as many packets as possible in the memory block. Thus, a single integer is used to store four flow registers in order to reduce the hash table bucket size and increase the accuracy of retransmissions' detection.

However, we note that atomic operations, which must be used to update counters throughout the GPU code, only work with integers (32 bits) and some of our registers are 8 bit long, such as the counter of TCP flags per connection. As a result, we have to account for a possible overflow, which cannot be prevented. In this section we evaluate the impact in flow records' accuracy. The file D was used to compare the differences in the number of flags and zero window announcements detected by the custom serial and GPU programs. Table 6 shows the number of records that present differences in these two fields. We note that the accuracy is remarkable in this case reaching values lower than 0.01%.

Table 6 Flow records presenting differences in TCP Flags and zero window announcements (serial versus GPU)

	SYN	FIN	RST	Zero window
Total records	54,054	53,971	53,900	54,103
Records with differences	27	19	122	22
Percentage	0.0005%	0.000352%	0.0026%	0.0004%

4.5 Throughput

For the sake of evaluation, a very large capture file of 1.4 TB was used to minimize the effects of initialization in the execution time. This file presented a total of 5,242 million TCP packets and 162 million flows with an average duration of 43.72 seconds.

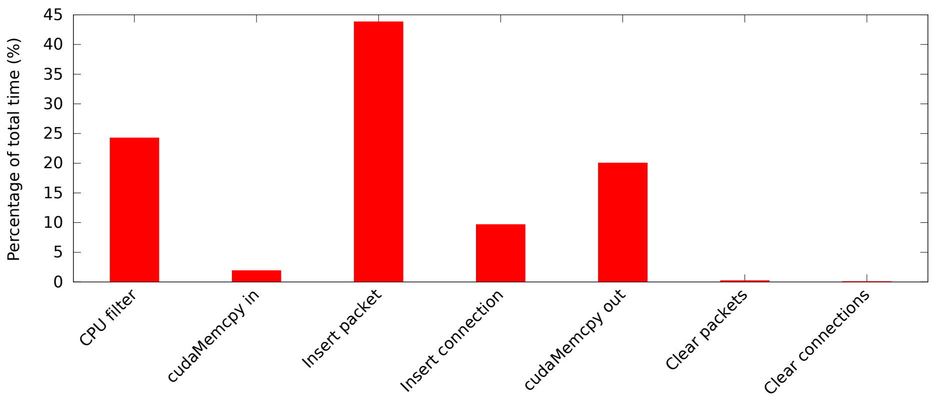

Fig. 12 Percentages of total time

We note that the throughput may be bounded by the hard disk read speed and we actually get a 1.1 Mpps if we consider the whole disk and GPU system. This figure increases to 4.4 Mpps if we remove the disk read latency from the calculation. Figure 12 shows the percentage of time spent in the different execution stages. Such stages are:

1. *CPU filter*: Filter the relevant packets.
2. *cudaMemcpy in*: Copy the relevant packets from host to GPU.
3. *Insert packet*: Insert packets in the packet hash table.
4. *Insert flow*: Insert packets in the flow hash table.
5. *cudaMemcpy out*: Copy connections from GPU to host.
6. *Clear packets*: Initialize packet hash table.
7. *Clear flows*: Initialize flow hash table.

We observe that the packet insertion task takes the longest time. Note that each packet has to be inserted into the packet hash table and, when using a GPU with 6 GB of memory, as many as 56 million of packets insertion are due. We note that the random access to memory is not well suited for a GPU. The "Packet to flow" task also involves a hash table, but the size is smaller because there are less connections than segments, so the performance penalty is not that significant. The time taken by the two *memcpy*s was variable even though pinned memory was used. The experiments were performed with the pcap file E , resulting in 12 executions of each part of the algorithm. This helps reduce the effect of the variability. It is worth noting that the packet filtering takes a lot of time because it must be executed for each packet. The filtering phase takes place in the CPU because it was found that the GPU was slower and yielded a worse accuracy. Lastly, the two *clear*s empty the packet and connection hash tables for the next batch.

4.5.1 Comparison with CPU

Fo the sake of performance comparison, the GPU algorithms were rewritten to run in the CPU. The resulting program runs in a single thread and the number of threads necessary to achieve the same performance as the GPU is extrapolated from the run time in a single CPU core. This is the best case scenario for the CPU solution, as the time lost in thread synchronization is not taken into account.

In Figures 13 and 14 we show that the GPU + CPU solution uses, on average, 1.5 processor cores and 3.7 GB of memory in the host with the default options, while the CPU solution makes use of 4 processor cores and 7.16 GB of memory to achieve the same performance and accuracy. As noted in the introduction, this is a significant saving in terms of cores, which is the bottleneck for traffic capturing and analysis at 10 Gbps in commodity hardware.

4.5.2 Capturing from network interface card

We have noted that the hard disk read speed is a limiting factor for the overall system throughput. However, the offline processing of a stored packet traces is not a real use case for the GPU, which will be typically working with live traffic from a 10 Gbps network interface card.

In this experiment, we actually assess that the measured throughput of 4.4 Mpps can be achieved when reading packets from the network interface card, i.e. a use case closer to the operational working environment. To do so, the same host used for the previous tests was connected to another host in charge of traffic generation (packet trace replay) at 10 Gbps through a 10 Gbps Ethernet link. The previously described traffic traces were replayed at different speeds while measuring packet loss. Using this setup, we achieved a 4 Mpps limit which is consistent with the analysis presented in the previous sections. We verified that the throughput was limited by the GPU and not by the driver, which is able to capture traffic at 10 Gbps.

5 Conclusions and future work

In this paper we have presented, for the first time ever, a GPU-based traffic capture and analysis system which is able to provide TCP flow records, including the challenging task of detecting per flow number of duplicates and retransmissions.

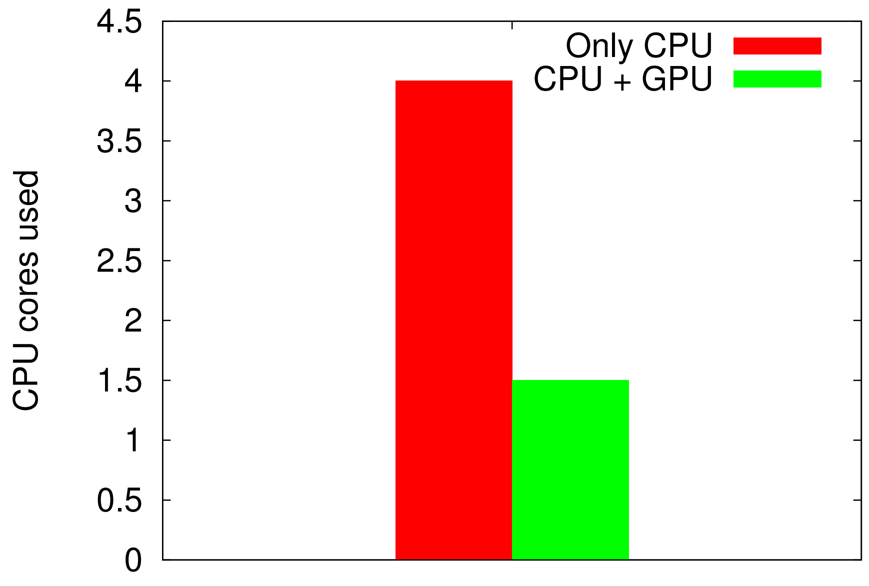

Fig. 13 Number of CPU cores used in both versions

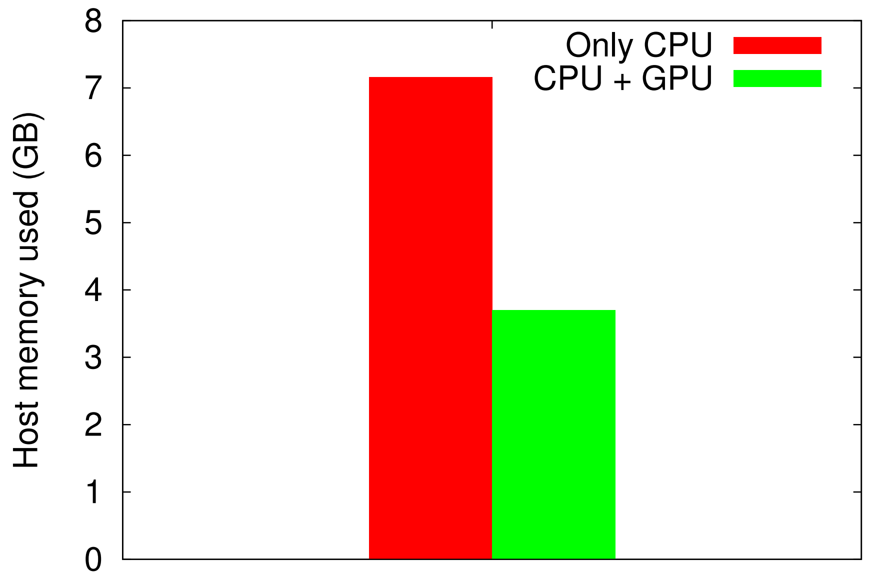

Fig. 14 Memory used in both versions

The fundamental breakthroughs to obtain a significant accuracy and speed have been studied, namely the limited memory size in the GPU board, which impedes to compare every packet with the previous ones in the trace. Despite of such constraints, the throughput for a single GPU system reaches a remarkable 4 Mpps figure. To put this figure into perspective we note that a fully saturated 10 Gbps link with 64-bytes packets produces 14.7 Mpps. However, a more realistic case with average packet size of 500 bytes, again in a fully saturated link, produces around 2 Mpps. We conclude that even though 10 Gbps line rate is not achieved, the system is fast enough to cope with a typical 10 Gbps link in real operational conditions, which, in addition to the larger packet size, is not saturated.

However, as important as throughput may be, this is not the most salient advantage of the proposed system. As it turns out, the major constraint for traffic capturing and analysis at 10 Gbps in commodity hardware is the number of CPU cores available in the system. By carefully analyzing the state of the art we have found that the most commodity hardware systems rely on Receive Side Scaling (RSS) to demultiplex traffic at the network interface card into several queues, each of which is attached to a fully dedicated CPU core. The use of GPUs alleviates the load in terms number of cores occupied in the traffic analysis. We also note that capturing the traffic only does not suffice for network monitoring: it is the analysis that matters.

As future work, we plan to achieve line rate by demultiplexing the incoming traffic stream into several GPUs, but not on a flow-per-flow basis, which demands a separate and possibly large hash table. Instead, we plan to use simple demultiplexing rules, based on the packet header structure. We also plan to test the system using more powerful GPUs, comparing the merits of both approaches and the possibility of combining them to produce a system capable of dealing with more throughput.

References

1. Benoit Claise. Rfc 3954: Cisco systems netflow services export version 9. Technical report, October 2004.
2. G. Sadasivan, N. Brownlee, B. Claise, and J. Quittek. Rfc 5470: Architecture for ip flow information export. Technical report, March 2009.
3. Inaki Ucar, Daniel Morato, Eduardo Magana, and Mikel Izal. Duplicate detection methodology for ip network traffic analysis. In *Measurements and Networking Proceedings (M&N), 2013 IEEE International Workshop on*, pages 161–166, Naples, Italy, 2013. IEEE.
4. Lothar Braun, Alexander Didebulidze, Nils Kammenhuber, and Georg Carle. Comparing and improving current packet capturing solutions based on commodity hardware. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, pages 206–217. ACM, 2010.
5. Florian Braun, John Lockwood, and Marcel Waldvogel. Protocol wrappers for layered network packet processing in reconfigurable hardware. *Micro, IEEE*, 22(1):66–74, 2002.
6. Cisco. Network analysis module (nam) products, 2014. URL `http://www.cisco.com/go/nam`.

7. Katerina Argyraki, Salman Baset, Byung-Gon Chun, Kevin Fall, Gianluca Iannaccone, Allan Knies, Eddie Kohler, Maziar Manesh, Sergiu Nedevschi, and Sylvia Ratnasamy. Can software routers scale? In *Proceedings of the ACM workshop on Programmable routers for extensible services of tomorrow*, pages 21–26, Seattle, USA, 2008. ACM.
8. Mihai Dobrescu, Norbert Egi, Katerina Argyraki, Byung-Gon Chun, Kevin Fall, Gianluca Iannaccone, Allan Knies, Maziar Manesh, and Sylvia Ratnasamy. Routebricks: Exploiting parallelism to scale software routers. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 15–28, Big Sky, USA, 2009. ACM.
9. Sangjin Han, Keon Jang, KyoungSoo Park, and Sue Moon. Building a single-box 100 gbps software router. In *Local and Metropolitan Area Networks (LANMAN), 2010 17th IEEE Workshop on*, pages 1–4, New Jersey, USA, 2010. IEEE.
10. J.L. García-Dorado, F. Mata, J. Ramos, P.M. Santiago del Río, V. Moreno, and J. Aracil. *High-Performance Network Monitoring Systems Using Commodity Hardware*, volume 7192 of *Computer Communications and Networks series*. Springer-Verlag, 2013.
11. Sangjin Han, Keon Jang, KyoungSoo Park, and Sue Moon. Packetshader: a gpu-accelerated software router. *ACM SIGCOMM Computer Communication Review*, 41(4):195–206, 2011.
12. Nicola Bonelli, Andrea Di Pietro, Stefano Giordano, and Gregorio Procissi. *On Multi–gigabit Packet Capturing with Multi–core Commodity Hardware*, volume 7192 of *Lecture Notes in Computer Science*. Springer-Verlag, 2012.
13. Luigi Rizzo. Netmap: a novel framework for fast packet i/o. In *USENIX Annual Technical Conference*, pages 101–112, 2012.
14. Volodymyr V Kindratenko, Jeremy J Enos, Guochun Shi, Michael T Showerman, Galen W Arnold, John E Stone, James C Phillips, and Wen-mei Hwu. Gpu clusters for high-performance computing. In *Cluster Computing and Workshops, 2009. CLUSTER'09. IEEE International Conference on*, pages 1–8. IEEE, 2009.
15. Lei Wang, Shuhui Chen, Yong Tang, and Jinshu Su. Gregex: Gpu based high speed regular expression matching engine. In *Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS), 2011 Fifth International Conference on*, pages 366–370. IEEE, 2011.
16. Giorgos Vasiliadis, Spiros Antonatos, Michalis Polychronakis, EvangelosP. Markatos, and Sotiris Ioannidis. Gnort: High performance network intrusion detection using graphics processors. In *Recent Advances in Intrusion Detection*, pages 116–134. Springer, 2008.
17. Ying Zhang, Lu Peng, Bin Li, Jih-Kwon Peir, and Jianmin Chen. Architecture comparisons between nvidia and ati gpus: Computation parallelism and data communications. In *Workload Characterization (IISWC), 2011 IEEE International Symposium on*, pages 205–215. IEEE, 2011.
18. NVIDIA Corporation. Nvidia's next generation cuda compute architecture: Fermi. `http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf`, 2009. [Online; accesed 15-June-2015].
19. NVIDIA Corporation. Nvidia's next generation cuda compute architecture: Kepler tm gk110. `http://www.nvidia.com/content/PDF/kepler/`

NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf, 2012. [Online; accesed 15-June-2015].

20. David B Kirk and W Hwu Wen-mei. *Programming Massively Parallel Processors: A Hands-on Approach.* Newnes, 2012.
21. Wenji Wu and Phil DeMar. Network traffic monitoring and analysis with gpus, 2013. URL http://sc13.supercomputing.org/sites/default/files/PostersArchive/tech_posters/post161s2-file3.pdf. [Online; accesed 15-June-2015].
22. M. Zadnik, L. Polsk, O. Lengal, M. Elich, and P. Kramolis. Flowmon for network monitoring. In *Networking Studies V: Selected Technical Reports*, pages 135–153, 2011.
23. Sherif Yusuf, Wayne Luk, Morris Sloman, Naranker Dulay, Emil C Lupu, and Geoffrey Brown. Reconfigurable architecture for network flow analysis. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 16(1):57—-65, 2008.
24. P. Rajeswari and N. Nagarajan. An fpga-based hardware architecture for network flow analysis. *European Journal of Scientific Research*, 83(3):338–337, 2012.
25. Marco Forconesi, Gustavo Sutter, Sergio Lopez-Buedo, and Javier Aracil. Accurate and flexible flow-based monitoring for high-speed networks. In *Field Programmable Logic and Applications (FPL), 2013 23rd International Conference on*, pages 1–4. IEEE, 2013.
26. Marc Suñé Clos. A framework for network traffic analysis using gpus. https://upcommons.upc.edu/pfc/bitstream/2099.1/8800/1/Thesis.pdf, 2010. [Online; accesed 15-June-2015].
27. Marc Suñé Clos. Netgpu project. http://code.google.com/p/netgpu, 2010. [Online; accesed 15-June-2015].
28. Dan A Alcantara, Andrei Sharf, Fatemeh Abbasinejad, Shubhabrata Sengupta, Michael Mitzenmacher, John D Owens, and Nina Amenta. Real-time parallel hashing on the gpu. In *ACM Transactions on Graphics (TOG)*, volume 28, pages 154:1–154:9, New York, NY, USA, 2009. ACM, ACM.
29. A. Papadogiannakis, M. Polychronakis, and EP. Markatos. Scap: stream-oriented network traffic capture and analysis for high-speed networks. In *Proceedings of the 2013 conference on Internet measurement conference*, pages 441–454, 2013.
30. V. VMoreno, P.M. Santiago del Rio, J. Ramos, D. Muelas, J.L. Garcia-Dorado, Gomez-Arribas F.J., and J. Aracil. Multi-granular, multi-purpose and multi-Gb/s monitoring on off-the-shelf systems. *International Journal of Network Management*, 24(4):221–234, 2014.