International Journal of Networked and Distributed Computing, Vol. 2, No. 4 (October 2014), 198-210

Fast Parallel Network Packet Filter System based on CUDA

Che-Lun Hung*
Department of Computer Science and Communication Engineering, Providence University, 200, sec. 7, Taiwan Boulevard, Shalu Dist.,
Taichung, 43301, Taiwan
E-mail: clhung@pu.edu.tw

Shih-Wei Guo
Department of Computer Science and Information Managemen, Providence University, 200, sec. 7, Taiwan Boulevard, Shalu Dist.,
Taichung, 43301, Taiwan
E-mail: cometlcc@gmail.com

Abstract

In recent years, with the rapid development of the network hardware and software, the network speed is enhanced to multi-gigabit. Network packet filtering is an important strategy of network security to avoid malicious attacks, and it is a computation-consuming application. Therefore, we develop two efficient GPGPU-based parallel packet classification approaches to filter packets by leveraging thousands of threads. The experiment results demonstrate that the computational efficiency of filtering packet can be significantly enhanced by using GPGPU.

Keywords: Packet filtering; Packet classification; GPGPU; CUDA; Parallel Processing.

1. Introduction

In the past few years, the development of network bandwidth and hardware technologies have grown rapidly, a variety of Internet services have been popular, such as email system, storage system, entertainment system and others. Currently, many famous corporations, such as Google, Amazon, IBM and Microsoft have released a variety of cloud services including hardware, software and platform. These services deeply rely on Internet. To maintain the robustness of Internet environments, the network security is one of the important parts indeed. Therefore, Internet security has become an important role to protect activities on Internet. For this purpose, packet analysis[1] is useful strategy to control that packet data can flow to and from a network. The criteria that use when inspecting packets are based on the Layer 3 (IPv4 and IPv6) and Layer 4 (TCP, UDP, ICMP, and ICMPv6) headers. The commonly used criteria are source and destination address, source and destination port and protocol.

Packet analysis is at the core of timely detection and typically relies on a packet filtering system. Packet filtering system drops the packets if packets match to the filter rules. A packet is compared with the filter rules in sequential order from the first to the last. Until

*Corresponding author: Che-Lun Hung. E-Mail: clhung@pu.edu.tw.

the packet matches a rule containing the keyword such as port number or IP address, the packet will be compared against all filter rules before the final action is taken. The last matched rule dictates what action to take on the packet. There is an implicit pass all at the beginning of a filtering rule set meaning that if a packet does not match any filter rule the resulting action will be passed. In general, each incoming packet can be considered independently of any other packet. Although for the IP fragments, the first fragment is related to other fragments apparently but arriving without order. It can be considered as independent for classification[2].

Packet filter system also has been applied to many network intrusion detection systems (NIDS) as the first stage. Botnet has become one major threat to Internet users in recent years. A botnet consists of a large number of bots that are networked computers compromised by malicious attackers. In general, an attacker conducts the bots to launch a variety of types of attacks such as phishing and spamming with a botnet, and then receives benefits from a variety of aspects such as economy and social security. Most of methods to detect bot's activities according to predefined patterns and signatures retrieved from well-known bots[3,4,5,6,7,8]. Although signature-based approaches are able to detect bots accurately, it is difficult to detect botnet in real time. Nowadays, detecting signature becomes more and more complex because the number of attacks is increased dramatically, and the signatures also become sophisticated. In addition, busy-hour Internet traffic will increase by a factor of 3.4 between 2013 and 2018. Therefore, to analyze the entire traffic on a high-speed link is computation-consuming problem. Consequently, random packet losses are likely to occur if the network traffic exceeds the capacity of packet filter of the botnet detection system. These botnet detection approaches has a very important stage, traffic reduction, to reduce the data set to the meaningful subset of flows to speed later stages in these approaches. To reduce the network traffic, packet filtering is useful strategy to control that packet data can flow to and from a network.

Sequential-process packet classifiers often take longer to classify a packet set captured off a giga-bit network interface than it took the set to arrive, making them infeasible for real-time traffic analysis[4]. To solve packet filter problem leaded by large amounts of traffic, one of the solutions is to increase the processing capacity of the botnet detection system. It can be either hardware[9] or software[10] solutions. However, the cost of these solutions is huge. The reasonable cost of available computing power to analyze the network traffic is critical. Therefore, parallel process is useful solution to improve the performance of the detection system.

Recently, many literatures tried to use General-Purpose Graphics Processing Unit (GPU) to solve computation intensive problem of various domains[11,12,13,14,15,16,17]. GPGPU programming has been successfully utilized in the scientific computing domains which involve a high level of numeric computation. However, other applications also could be successfully parallelized by GPGPU. The greatest benefit is that the processing units grow from many (CPU, about 2-16) to massive (GPU, about over 512). In 2006, NVIDIA proposed the Compute Unified Device Architecture (CUDA). CUDA uses a new computing architecture named Single Instruction Multiple Threads (SIMT)[18]. This architecture allows thread to execute independent and divergent instruction streams, facilitating decision-based execution that is not provide for by the more common Single Instruction Multiple Data (SIMD).

To improve the computational performance of the Botnet detection system, we propose two efficient network traffic reduction algorithms to filter packets simultaneously by using GPGPU device. By leveraging nVidia CUDA device can achieve low cost, commodity GPU co-processors to accelerate packet-filtering throughput. The proposed algorithms could be utilized in any botnet detection method that adopts traffic reduction strategy. We also implement the proposed packet classification algorithm on a variety of memory architectures on GPU to discuss the performance of proposed method. The experiment results demonstrate that the proposed method can achieve over 20 times speed up over the sequential packet classification software on single CPU. It presents that GPGPU is useful for real-time traffic analysis.

2. Related Works

According the previous literature[1], the classification can be categorized into three types: IP routing, packet demultiplexing and packet analysis. The slight differences of these types are between target environments. IP routing is utilized to forward incoming packets through the correct interface to a destination host. Packet demultiplexing is concerned that

forwarding the packets to the next hop router, or dropping the packets altogether. Packet analysis is similar in many respects to demultiplexing, and often depends on similar filtering algorithms, but may process a wider variety of packets, with a broader range of destinations. All of these three types are depending on the packet filtering. A filter is a predicate function that operates over a collection of criteria to compare each arriving packet[1,19,20]. Generally, a packet is classified by a filter which has the specific criteria. Filter criteria are Boolean valued comparisons, performed between values contained in discreet bit-ranges in the packet header and static protocol defined values.

The commonly-used and most reliable methods of classifying packet data are exhaustive search algorithms which compare packets against each and every filter in the filter set until a exact match is found[1,21]. These algorithms are generally slow, and thus not very useful. Other classes of algorithms include decision tree, decomposition and tuple space approaches. Decision tree algorithms are diverse in design, but all leverage a sequential tree like traversal of a specialized data structure in order to narrow down the number of criteria against which the packet need to be compared[1,19,22,23]. Most of demultiplexing and analysis filters are highly sequential approaches based on decision trees[19,20,24], and thus are suitable to the processing on CPUs. In contrast, decomposition algorithms can be equipped on parallel processing hardware such as FPGAs, typically splitting filter classifications into smaller sub-classifications which can be performed in parallel[1,25,26]. Tuple space algorithms are highly specialized, and exploit a variety of filter set properties in order to reduce processing time[1].

Most of demultiplexing algorithms adopt decision tree approaches because of their efficiency at pruning redundant computation on sequential processors. BPF is a well known algorithm that adopted Control Flow Graphs (CFGs) in an assembler style programmable pseudo-machine to improve performance on register-base processor[20]. Mach Packet Filter (MPF) and Dynamic Packet Filter (DPF) are extended from BPF. MPF was designed to extend and improve demultiplexing performance[27], while DPF focused on exploiting dynamic code generation in order to prune redundant instructions[24]. These filters led to the development of BPF+[19], which adopted techniques such as Predicate Assertion Propagating and Partial Redundancy Elimination, in conjunction with Just-In-Time (JIT) processing and various other optimizations, to dramatically improve processing speeds. Extensible Packet Filter (xPF)[28], Fairly Fast Packet Filter (FFPF)[29] and SWIFT[30], were developed to reduce the context switching overhead, facilitate high performance demultiplexing between multiple network monitoring application, and reduce filter update latency to support real-time filter updates, respectively.

Presently, RFC algorithm[31], which is a generalization of cross-producting[32], is the fastest classification algorithm in terms of the worst-case performance. Bitmap compression has been used in IPv4 forwarding[33,34] and IPv6 forwarding[35]. It is applied to classification to compress redundant storage in data structure[36]. However, the performance bottleneck of these methods are searching the compressed tables, and thus additional techniques have to be introduced to improve the inefficiency of calculating the number of bits set in a bitmap. Lulea[33] algorithm utilizes a summary array to pre-process the number of bits set in the bitmap, and thus it needs an extra memory access operation per trie-node to search the compressed table. The Bitmap-RFC[37] employs a built-in bit-manipulation instruction to calculate the number of bits set at runtime and apply bitmap compression to reduce its memory requirement to solve the problem of memory explosion. Thus, it is much more efficient than Lulea's in terms of time and space complexity.

However, these sequential packet classification algorithms take longer to classify a packet set captured off a giga-bit network interface than it took the set to arrive, making them infeasible for real-time traffic analysis.

Nottingham et al., [2,11] proposed a classification algorithm, by utilizing GPU co-processors to accelerate classification throughput and maximize processing efficiency in highly parallel execution context. They provided valuable articles for introducing the concept of parallel packet classification on CUDA and OpenCL platforms. However, these literatures are lack of the performance comparisons and implementations with a variety of memory architectures of GPU. Han et al., [38] proposed a GPU-based IP routing approach named PacketShader. The experiment results show that GPU-based IP routing algorithm can enhance the performance over the CPU-based IP routing approaches. Hung et al.,[39] presented a GPU-based network packet pattern-

matching algorithm for network intrusion detection systems by leveraging the computational power of GPUs to accelerate pattern-matching operations and subsequently increase the overall processing throughput. The experimental results showed that their algorithm can achieve a maximal traffic processing throughput of over 2 Gbit/s. These articles present an alternative of developing the parallel packet classification by leveraging GPU devices.

3. CUDA Programming Model

As the GPU has become increasingly more powerful and ubiquitous, researchers have begun developing various non-graphics, or general-purpose applications[13]. Generally, the GPUs are organized in a streaming, data-parallel model in which the co-processors execute the same instructions on multiple data streams simultaneously. Modern GPUs include several (tens to hundreds) of types of stream processors, both of graphical and general-purpose applications thus are faced with parallelization challenges in using GPUs[41]. nVidia released the Compute Unified Device Architecture (CUDA) SDK to assist developers in creating non-graphics applications that run on GPUs. A CUDA programs typically consist of a component that runs on the CPU, or host, and a smaller but computationally intensive component called the kernel that runs in parallel on the GPU. Input data for the kernel must be copied to the GPU's on-board memory from host's main memory through the PCI-E bus prior to invoking the kernel, and output data also should be written to the GPU's memory first before copying to host's main memory. All memory used by the kernel should be pre-allocated.

Kernel executes a collection of threads that computes a result for a small segment of data. To manage multiple threads, kernel is partitioned into thread blocks, with each thread block being limited to a maximum of 512 threads. The thread blocks are usually positioned within a one or two-dimensional grid. Each thread can be positioned within a given block where it belongs, and this given block can be positioned within the grid. Therefore, each thread can calculate which elements of data to operate on, and which regions of memory to write output to by an algebraic formula. Each block is executed by a single multiprocessor, which allows all threads within the block to communicate through on-chip shared memory. The parallelism architecture of GPGPU is illustrated in Fig. 1.

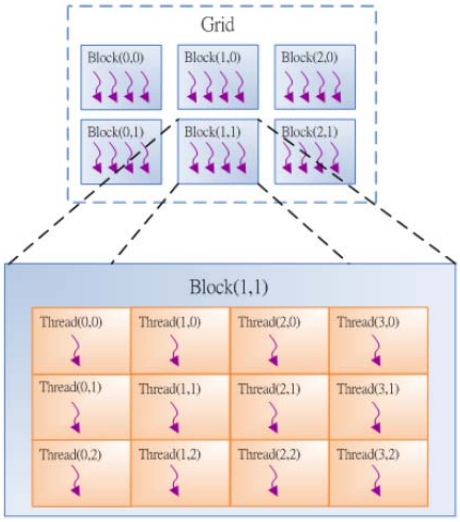

Fig. 1. The parallelism architecture of CUDA.

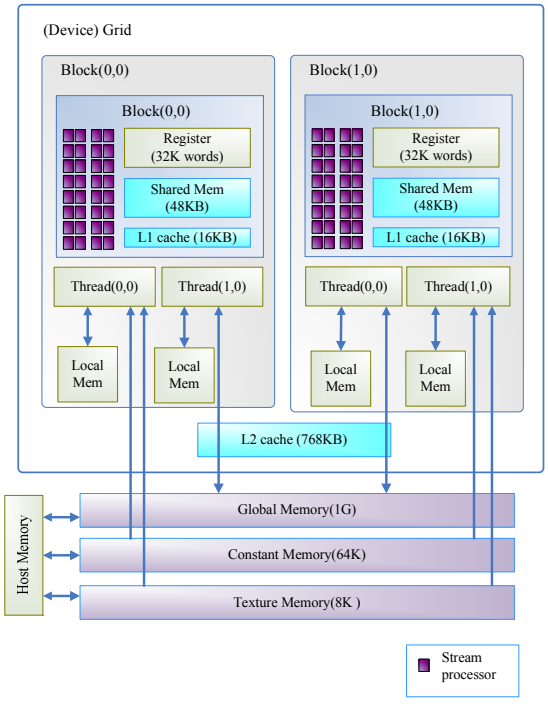

Fig. 2. Memory architecture of CUDA device, nVidia GTS450.

Table 1. Overview of memory architectures in CUDA device.

Memory	Scope	Hardware	Latency	Bandwidth	Access
Register	Thread	Chip	Immediate		R/W
Shared	Block	Chip	4-6 clock	200 GB/s	R/W
Constant	Grid	Cache	4-600 clock	200-300 GB/s	R
Texture	Grid	Cache	4-600 clock	200-300 GB/s	R
Global	Grid	DRAM	400-600 clock	100 GB/s	R/W
Local	Thread	DRAM	400-600 clock	100 GB/s	R/W

CUDA devices provide access to several memory architectures, such as global memory, constant memory, texture memory, share memory and registers, with their access latencies and limitations. The performance of device is relevant to the memory variants. Figure 2 illustrates the memory architectures of CUDA device. Table 1 shows the characteristics of memory architectures.

- Global Memory: Global memory is the biggest memory region available on CUDA devices and is capable of storing hundreds of megabytes of data. However, the access latency is highest than others.
- Constant Memory: Constant memory is a small read-only memory region that resides in DRAM on CUDA device. It is globally accessible memory for all threads. Since Constant memory has on-chip cache, the access latency is low.
- Texture Memory: Texture memory is a compromise between global and constant memory. Each multi-processor on the CUDA device equips a 64K B texture cache which can be bound to one or more arbitrarily sized region of global memory. Texture memory is read only as constant memory.
- Register: Each thread block in CUDA device equips a register file that contains registers. The register provides fast thread-local storage during kernel execution.
- Shared Memory: Shared memory is block-local that facilitates cooperation between multiple threads in an executing thread block. The access latency of shared memory is equivalent to that of register.

The G80 that introduced the CUDA architecture had 86.4GB/s of memory bandwidth, plus an 8 GB/s communication bandwidth with the CPU. A CUDA application can transfer data from the system memory at 4 GB/s and at the same time upload data back to the system memory at 4 GB /s. altogether, there is a combined total of 8 GB/s. the communication bandwidth is much lower than the memory bandwidth and may seem like a limitation. Two memory transmission models, memory direct copy and zerocopy, in CUDA architecture is used to copy data from system memory to device memories including global, constant, texture and registers.

The CUDA runtime system provides application programming interface (API) functions to allocate and release memory on the device and transfer pertinent data from the host memory to the allocated device memory. It could be the bottleneck of GPU computing performance. The zerocopy function is a feature that was added in version 2.2 of the CUDA Toolkit. It enables GPU threads to directly access host memory. The zerocopy function can be used in place of streams because kernel-originated data transfers automatically overlap kernel execution without the overhead of setting up and determining the optimal number of streams. Different to previous two memory transfer models, streaming model is another way to improve the use of the threads and data transfer. It is a pipeline of asyconizing the data transmission. A stream is a sequence of operations that are performed in order on CUDA device. Figure 3 shows the streaming model of CUDA architecture. In Fig. 3, data is copied from host memory to device memory and stream 1 then lunches kernel function 1, and when stream1 is copying output data from device memory to host memory and stream 2 starts to lunch kernel function 2. The processes for stream 3 and stream 4 are as same as pervious processes of stream1 and stream 2.

4. Method

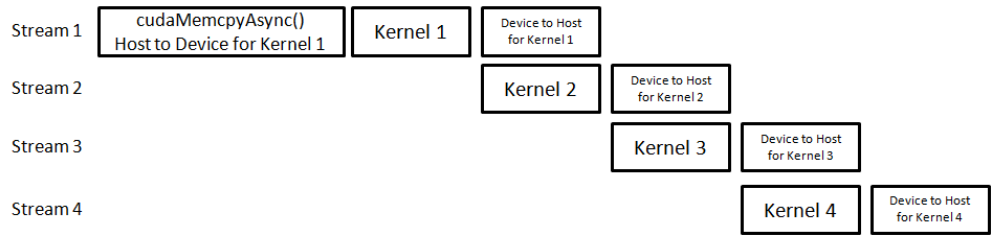

Fig. 3. Streaming approach on CUDA.

In CUDA devices, each physical multiprocessor contains only a single instruction register which drives eight independent processing cores simultaneously. Therefore, any divergence between thread executing on the same multiprocessor forces the instruction register to issue instruction for all thread paths sequentially whilst non-participating thread sleep[18]. The significant thread divergence can dramatically impair performance. To avoid thread divergence, each thread should process the similar length of data. In a filter set, the rules of each filter have a various number of fields. Due to this reason, we restrict that he number of fields of each rule should be the same at the same filter. Table 2 presents the revised filter rules. The filter sets are stored in constant memory or register files, and the packet data is stored in global memory or texture memory. Figure 4 illustrates the memory usage for filter sets stored in constant memory and packet data stored in global memory on CUDA device. A thread process a packet data according the filter sets. Currently, we include two famous filter classification algorithms, BPF and BitMap-RFC, in the proposed method. We chose these two algorithms because they are totally different data structures and well-known algorithm. BPF uses data structure as decision tree structure and BitMap-RFC uses data structure as hash table. Through different data structures, it can be measured that which data structure is suitable for GPU.

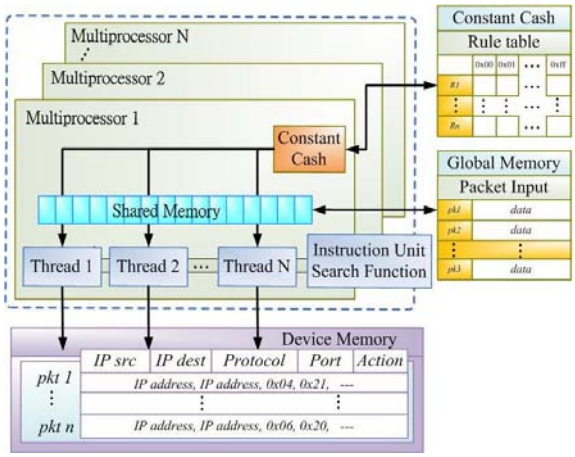

Fig. 4. The memory storage for filter sets (constant memory) and packet data (global memory) on CUDA device.

4.1. *GPU-Based BPF Filtering Algorithm*

BPF was originally proposed for BSD UNIX, it is independent of the TCP/IP stack, and gives user mode processes an interface to access data link layer. It is an elegant and commonly used solution for packet filtering and achieves better performance than other packet filtering systems. BPF consists of two main components: network tap and packet filter[22]. The network tap is the first interface to copy packets from network interface card driver and moving them to the listening user processes. Then, the packet filter adopts

Table 2. Packet filtering rules

Rule	Source IP	Destination IP	Protocol	Port Number
1	140.128.1.0/ 255.255.255.0	*	*	*
2	*	140.128.2.99/ 255.255.255.0	HTTP	8888
3	140.127.0.0/ 255.255.0.0	123.6.0.0/ 255.255.0.0	*	*
4	219.88.0.0/ 255.255.0.0	123.6.22.0/ 255.255.255.0	FTP	*

the filtering rules to determine whether a packet should be delivered to the upper-level component or it should be discarded in the kernel space. BPF adopts a directed acyclic control flow graph (CFG) to represent a packet filter. Figure 5 shows a sample BPF filter. Each path in the BPF filters represents a comparison procedure needs to be completed for a particular packet pattern. In the proposed GPU-based BPF filtering method, each thread compares the fields of a packet head with the filtering rules. As figure 6, a thread start to do the comparison of the field corresponding to the first condition of rule. If the condition of the rule is hit, the thread then thread stops the comparison and output the results. Otherwise, the thread passes this condition and go to compare next condition. If no condition of a rule is hit, the thread will pass this rule and go to compare next rule. Figure 6 shows the proposed GPU-based BPF filtering algorithm. Since GPU has different memory architectures, filtering rules and packet head data can be stored in these memory architectures shown in Fig. 4.

4.2. *GPU-Based BitMap-RFC Filtering Algorithm*

Different to BPF algorithm, the RFC algorithm is a decomposition-based algorithm that is able to provide very high lookup throughput at the cost of low memory efficiency. RFC consists of two phases; direct table lookup and crossproducting. In the first phase, it performs the parallel table lookups on each filter field first. This step can achieve the best throughput performance by utilizing a direct table look up. Figure 7 shows the data structure of this table. The table entry *i* of chunk *j* stores the set of filters for which the chunk *j* covers the value *i*. Each unique set of filters is binary encoded. The eqID is the identifier of the encoded set. The second phase is to build a cross-product table with the number of entries equaling to the multiplication of the number of eqIDs of each chunk. However, the number of entries of a cross-produce table can be very large and it leads to inefficient use of use. To solve this problem, the crossproducting phase is conducted recursively to build multiple cross-product tables.

In the proposed GPU-based BitMap-RFC method, each thread copies with a packet data. In the initial stage, CPU builds all tables, and these tables then are copied to GPU's memory. In the second stage, each thread performs RFC algorithm and store the result to GPU's global memory. After all th reads complete filtering computation, the results will be copy to host's memory.

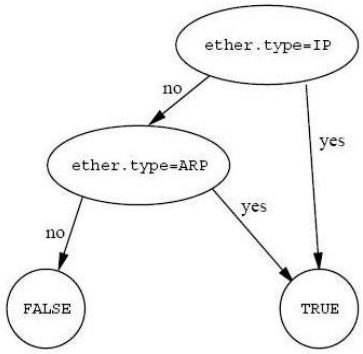

Fig. 5. BPF Filter Model.

```
BPF algorithm on GPU
Input : GpuSrcPacket , GpuDstPacket, con_RuleIndex, con_Ruletable, con_mask
        /* GpuSrcPacket: the source IP address of packet
           GpuDstPacket: the destination IP address of packet

           All filter rules are saved in constant memory
           RuleSize : The number of filter rules
con_RuleIndex: Save the Index for con_Ruletable
           con_mask: sub-network mask if necessary
__constant__ u_int16_t con_RuleIndex [RuleSize];
__constant__ u_int16_t con_Ruletable [RuleSize];
__constant__ u_int16_t con_mask;
        */
Output : GpuPacketOut Result /*Marked the packet that should be filtered our */
Method
Begin
1.      int tid = blockIdx.x * blockDim.x + threadIdx.x; /* get thread id */
2.      /* compare rules */
3.      for j = 0 to RuleSize − 1
4.          if ((GpuSrcPacket + tid)& con_mask) == con_RuleIndex[j])
5.              GpuPacketOut Result[t id ] = con_Ruletable[j];
6.      End
End
```

Fig. 6. GPU-based BPF algorithm.

Figure 8 shows the proposed GPU-based BitMap-RFC filtering algorithm. Figure 9 shows the GPU-based packet filter model for BPF and RFC algorithms.

4.3. *GPU Streaming for Implementation of Packet Filter*

In the above section, both of two proposed GPU-based packet filtering methods can adopt direct memory copy and zero-copy models to transfer data to GPU from host and transfer data to host from GPU. In addition, CUDA provides a mechanism, called streaming, to overlap the data transferring and thread computation to enhance the performance. Therefore, the time for transferring data between host and GPU can be decreased dramatically. To implement the two proposed methods, all filtering rule tables are copied to GPU's memory first. The packet data are split in many chunks, and then these chunks are copied to GPU memory by memory-copy stream sequentially. In the initial phase, the first chunk is copied to GPU's memory completely. Then, these

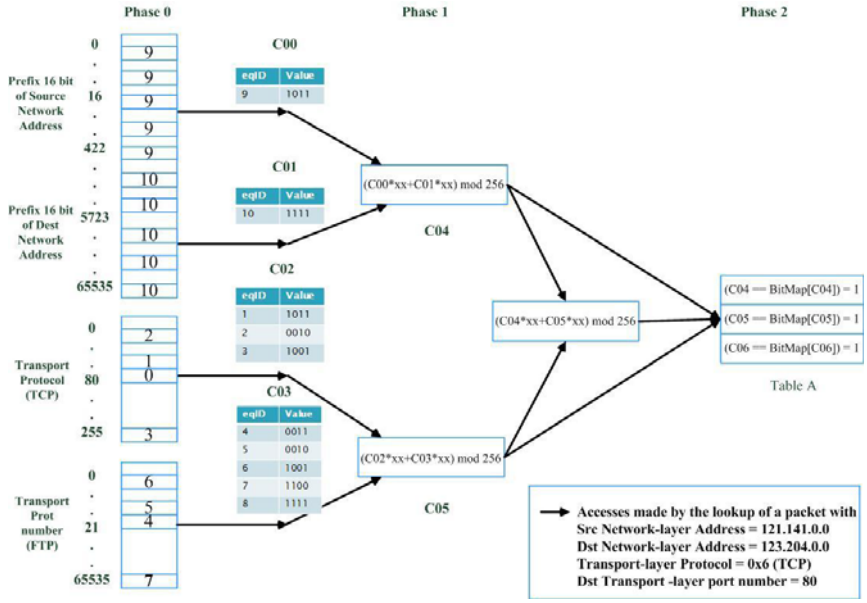

Fig. 7. This is the caption for the figure. If the caption is less than one line then it is centered. Long captions are justified manually.

```
BitMap-RFC algorithm on GPU
Input : GpuSrcPacket , GpuDstPacket, con_eqID, con_RuleIndex, con_Ruletable, con_mask
        /* GpuSrcPacket: the source IP address of packet
           GpuDstPacket: the destination IP address of packet

           All filter rules are saved in constant memory
           RuleSize : The number of filter rules
           con_eqID: Save the Index for con_RuleIndex, and the index is transformed from IP address
           con_RuleIndex: Save the Index for con_Ruletable
           con_mask: sub-network mask if necessary
__constant__ u_int16_t con_eqID[RuleSize];
__constant__ u_int16_t con_RuleIndex [RuleSize];
__constant__ u_int16_t con_Ruletable [RuleSize];
__constant__ u_int16_t con_mask;
        */
Output : GpuPacketOut Result
Method
Begin
1.      int src,dst;
2.      int tid = blockIdx.x * blockDim.x + threadIdx.x; // get threadID
3.      src = con_RuleIdex[con_eqID[(GpuSrcPacket + tid) & con_mask]*eqIDNumber];
4.      dst = con_RuleIdex[con_eqID[(GpuDstPacket + tid) & con_mask]*eqIDNumber];
5.      GpuPacketOutResult [tid] = con_Ruletable[(src+dst)%256];
End
```

Fig. 8. GPU-based BitMap-RFC algorithm.

packet data is processed and the second chunk is copied to GPU's memory in the second phase. The second phase is repeated until all packet data is processed completely and the filtering results are copied to host's memory.

5. Experiment

We implemented two packet classification algorithms, BPF and BitMap-RFC, on single NVIDIA GeForceGTS 560ti graphics card (Fermi architecture) and installed in a PC with an Intel i7-3930k CPU and 16GB DDRIII-1333 RAM running the Linux operating system. We simulated 65 million packets with the random source address, destination address, source port, destination address and protocol for the experiments. The packet filter has three rules and each rule has two fields as shown in Table 2. 1000 classification rules are created by these three rules for the following experiments.

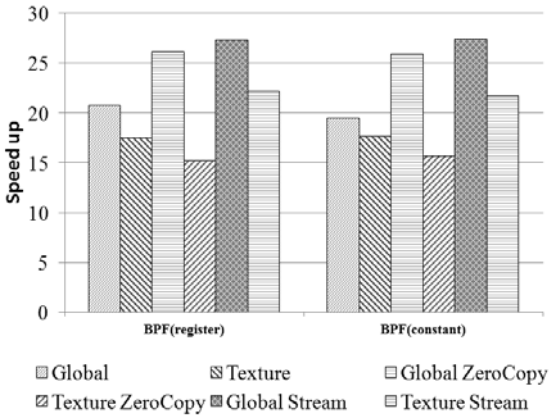

Fig. 9. Performance comparison between CPU-based and GPU-based BPF classification algorithms with a variety of memory usage.

5.1. *Performance Evolution of GPU-Based BPF Packet Filter*

In this experiment, we implemented BPF on CUDA with 10 combinations of CUDA memory storages and

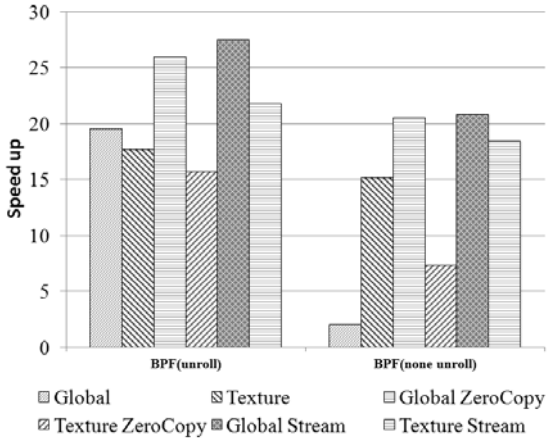

Fig. 10. Performance comparison between GPU-based BPF algorithms with unroll and non-unroll approaches.

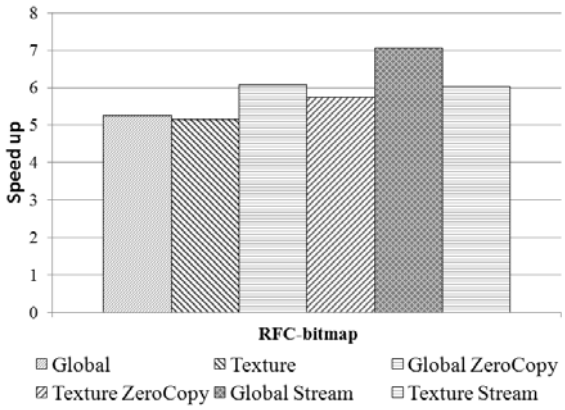

Fig. 11. Performance comparison between CPU-based and GPU-based BitMap-RFC classification algorithms with a variety of memory usage.

data transfer models shown in table 3. Global memory is the biggest memory region available on CUDA devices. Constant memory and register files can access data faster than global and texture memory. However, some limitations on these two structures. First is the storage size. Constant memory is suitable for frequent access but low data update rate. The function of register on CUDA is the same as the registers on CPU. The over-usage of register will decrease the performance of GPU. Therefore, we store the packet data in global and texture memory and the filtering rules are sto red in registers and constant memory. Also, we implement the proposed algorithm with three memory copy models, direct copy, zero-copy and streaming.

Figure 9 illustrates the performance comparison between CPU-based and GPU-based BPF classification algorithms. F igure 9 s hows that the GPU-based BPF algorithm can achieve 20x ~25x speedup over CPU-based BPF algorithm with storing classification rules in registers for 65 million packets; especially by using streaming model. It is obvious that the speedup of BPF by using global memory with streaming model is slightly superior to that of using global memory with zero-copy.

Actually, we implement GPU-based BPF classification algorithm with an optimization approach,

Table 3. Combination of various memory architectures for GPU-based BPFalgorithm

	Packet Storage	**Rule Storage**	**Data Transfer**
1	Global	Constant	Direct
2	Global	Constant	Zero Copy
3	Texture	Constant	Direct
4	Texture	Constant	Zero Copy
5	Texture	Constant	Stream
6	Global	Register	Direct
7	Global	Register	Zero Copy
8	Texture	Register	Direct
9	Texture	Register	Zero Copy
10	Texture	Register	Stream

Table 4. Combination of various memory architectures for GPU-based BitMap-RFC algorithm.

	Packet Location	Rule Location	Data Transfer
1	Global	Constant	NA
2	Global	Constant	Zero Copy
3	Global	Constant	Stream
4	Texture	Constant	NA
5	Texture	Constant	Zero Copy
6	Texture	Constant	Stream

unroll. In the implementation of GPU-based BPF algorithm, lots of loop operations are adopted to check the filter rules. However, it causes a critical problem that the number of registers is not enough for computing. The performance is affected. Therefore, an optimization approach, unroll, supported by CUDA is adopted to enhance the performance of GPU-based BPF algorithm. Figure 10 shows the performance between GPU-based BPF with unroll and non-unroll approaches. It is obvious that the performance of GPU-based BPF with unroll is highly superior to that with non-unroll.

5.2. *Performance Evolution of GPU-Based BitMap RFC Packet Filter*

In this experiment, we only implemented BitMap-RFC on CUDA with 6 combinations of CUDA memory storages and data transfer models shown in table 4. BitMap-RFC algorithm needs to build hash tables before filtering. Since the sizes of these tables are bigger than that of register files, the filter rules cannot be stored in registers. The results show that the GPU-based RFC algorithm can achieve 5x~7x speedup over CPU-based RFC algorithm in Fig. 11. Because CPU-based BitMap-RFC is much faster than CPU-based BPF, the performance enhancement is not dramatic to results in BPF.

5.3. *Performance Evolution between GPU-Based Packet Filter Algorithms*

Figures 12 and 13 show the throughput produced by GPU-based BPF and BitMap-RFC algorithms. Obviously, GPU-based BitMap-RFC algorithm can achieve higher throughput than GPU-based BPF algorithm. The reason is that BPF executes many *if-else* branch instructions. It causes the divergent branch

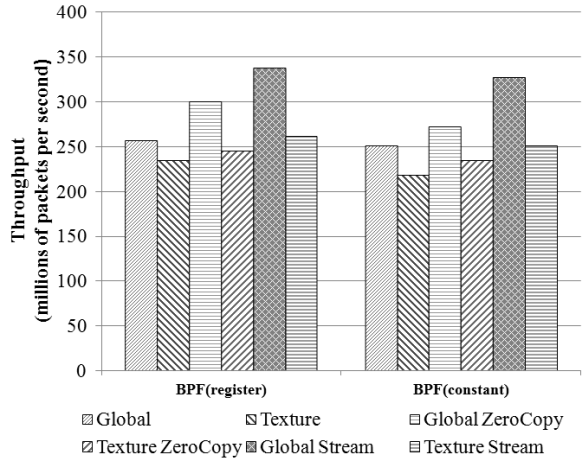

Fig. 12. Throughput comparison between memory usages on GPU-based BPF algorithm.

problem on CUDA device. Therefore, the performance of GPU-based BPF algorithm is decreased by divergent branch problem.

5.4. *Performance Evolution of GPU-Based Packet Filter Algorithms between Different CUDA Devices*

In this experiment, we implemented the proposed algorithms on different CUDA devices, such as GTS 450, GTX480 and GTX560ti. Figure 14 shows the throughput comparison between these three CUDA devices. From the result, GTX480 can achieve the highest throughput than other two devices, since GTX480 has more cores than other two devices; GTX480 has 480 cores, GTX450 has 192 cores and GTX560ti has 448 cores. The performance of the proposed algorithms is proportional to the number of cores of the CUDA device. Therefore, executing the

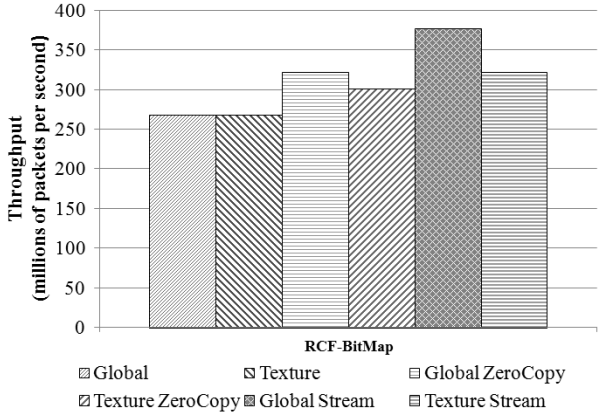

Fig. 13. Throughput comparison between memory usages on GPU-based BitMap-RFC algorithm.

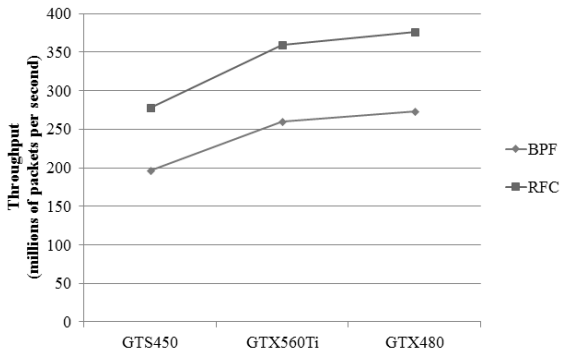

Fig. 14. Throughput comparison between two GPU-based fiilter algorithms on different CUDA devices.

proposed algorithm on high-level CUDA device and the performance can be enhanced.

6. Conclusion

In this paper, we propose two fast packet filter methods by leveraging power of GPU device. The proposed methods include two well-know filter algorithms, BPF and BitMap-RFC, to achieve rapid and reliable packet filter system. We implement these algorithms on GPU devices by using CUDA. In addition, we considered the performance of the various memory architectures available to CUDA kernels, and the different data transfer models to enhance the performance. The experimental results show the performance comparison among several combinations of memory architectures on CUDA device for the proposed methods. Obviously, the proposed methods can significantly enhance the performance over the filter algorithm executing on CPU.

In the future, we will apply GPU-based packet filter system to other network security systems, such as Botnet detection system and network intrusion detection system, to improve the performance. Also, we focus on improving the proposed methods to provide a real-time giga-bit network and fast network telescope packet analysis applications.

Acknowledgements

This research was partially supported by the National Science Council under the Grants NSC-102-2221-E-126-004.

References

1. D. E. Taylor, Survey and taxonomy of packet classification techniques, *ACM Comput. Surv.* **37**(3) (2005) 238–275.
2. A. Nottingham and B. Irwin, Parallel packet classification using GPU co-processors, *SAICSIT Conf. ACM,* (2010), pp. 231–24.
3. M. Roesch, Snort - Lightweight intrusion detection for networks, in *Proc. the 13th USENIX Conference on System Administration*, (1999), pp. 229–238.
4. V. Paxson, Bro: A system for detecting network intruders in real-time, *Computer Networks*, **31** (1999), pp. 2435–2463
5. W. Lu, M. Tav.allaee, G. Rammidi and A. A. Ghorbani, Botcop: An online botnet traffic classifier, in *Proc. the 7th IEEE Annual Communications Networks and Services Research Conference*, (2009), pp. 70–77.
6. F. Alserhani, M. Akhlaq, I. U. Awan and A. J. Cullen, Detection of coordinated attacks using alert correlation model, in *Proc. IEEE International Conference on Progress in Informatics and Computing*, (2010), pp. 542–546.
7. M. Szymczyk, Detecting botnets in computer net- works using multi-agent technology, in *Proc. the 4th International Conference on Dependability of Computer Systems*, (2009), pp. 192–201.
8. L. Braun, G. Munz and G. Carle, Packet sampling for worm and botnet detection in TCP connections, in *Proc. IEEE Network Operations and Management Symposium*, (2010), pp. 1542-1201.
9. S. Fide and S. Jenks, A Survey of String Matching Approaches in Hardware, *Dept. of Electrical Engineering and Computer Science, University of California, Irvine, Tech. Rep. TR SPDS 06-01*, (2006).
10. M. Colajanni and M. Marchetti, A Parallel Architecture for Stateful Intrusion Detection in High Traffic Networks,

in *Proc. of Workshop on Monitoring, Attack Detection and Mitigation*, (2006).
11. A. Nottingham and B. Irwin, GPU packet classification using OpenCL: a consideration of viable classification methods. In *Proc. SAICSIT Conf. ACM.*, (2010), pp. 160-169.
12. M. LCharalambous, P. Trancoso and A. Stamatakis, Initial Experiences Porting a Bioinformatics Application to a Graphics Processor, In *Proc. the 10th Panhellenic Conference on Informatics*, 2005, pp. 415-425.
13. J.D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A.E. Lefohn and T. Purcell, A Survey of General-Purpose Computation on Graphics Hardware, *Computer Graphics Forum*, **26** (2007), pp. 80-113.
14. C. L. Hung and G. J. Hua, Local Alignment Tool Based on Hadoop Framework and GPU Architecture, *Biomed Research International*, **2014** (2014), Article ID 541490.
15. S. T. Lee, C. Y. Lin, and C. L. Hung, GPU-Based Cloud Service for Smith-Waterman Algorithm Using Frequency Distance Filtration Scheme, *Biomed Research International*, **2013** (2013), Article ID 72173.
16. C. Y. Lin, C. L. Hung and Y. C. Hu, A Re-sequencing Tool for High- throughput Long Reads Based on UNImarker with non-Overlapping iNterval indexing strategy, *Information - An International Interdisciplinary Journal*, **16**(1(B)) (2013), pp. 827-832.
17. C. Y. Lin, S. T. Lee and C. L. Hung, Frequency-based re-sequencing tool for short reads on graphics processing units, *International Journal of Computational Science and Engineering*, **9**(1/2) (2014), pp. 3-10.
18. Nvidia cuda c best practices guide, version 4. Online, (2011).
19. A. Begel, S. McCanne, and S. L. Graham, BPF+: Exploiting global data-flow optimization in a generalized packet filter architecture, *SIGCOMM Comput. Commun. Rev.*, **29**(4) (1999), pp. 123-134.
20. S. Mccanne and V. Jacobson, The bsd packet filter: A new architecture for user-level packet capture, in *Proc. the USENIX Winter*, (1993), pp. 259-269.
21. E. Spitznagel, D. Taylor, and J. Turner, Packet classification using extended tcams, in *Proc. the 11th IEEE International Conference on Network Protocols*, (2003), pp. 120-131.
22. S. Singh, F. Baboescu, G. Varghese, and J. Wang, Packet classification using multidimensional cutting, in *Proc. the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, (2003), pp. 213-224.
23. V. Srinivasan, G. Varghese, S. Suri, and M. Waldvogel, Fast and scalable layer four switching, *SIGCOMM Comput. Commun. Rev.*, **28**(4) (1998), pp. 191-202.
24. D. R. Engler and M. F. Kaashoek, Dpf: Fast, flexible message demultiplexing using dynamic code generation, in *Proc. on Applications, technologies, architectures, and protocols for computer communications*, (1996), pp. 53-59.
25. F. Baboescu and G. Varghese. Scalable packet classification, *SIGCOMM Comput. Commun. Rev.*, **31**(4) (2001), pp. 199-210.
26. T. V. Lakshman and D. Stiliadis. High-speed policy-based packet forwarding using efficient multi-dimensional range matching, *SIGCOMM Comput. Commun. Rev.*, **28**(4) (1998), pp. 203-214.
27. M. Yuhara, B. N. Bershad, C. Maeda, J. Eliot and B. Moss. Efficient packet demultiplexing for multiple endpoints and large messages, *In Proceedings of the 1994 Winter USENIX Conference*, (1994), pp. 153-165.
28. H. Bos, W. D. Bruijn, M. Cristea, T. Nguyen and G. Portokalidis, FFPF: Fairly fast packet filters, In *Proc. the 6th conference on Symposium on Opearting Systems Design & Implementation*, (2004), pp. 24.
29. S. Ioannidis and K. G. Anagnostakis, XPF: Packet filtering for low-cost network monitoring, In *Proc. the IEEE Workshop on High-Performance Switching and Routing*, (2002), pp. 121-126.
30. Z. Wu, M. Xie, and H. Wang, Swift: a fast dynamic packet filter, In *Proc. the 5th USENIX Symposium on Networked Systems Design and Implementation*, (2008), pp. 279-292.
31. P. Gupta and N. McKeown, Packet Classification on Multiple Fields, *SIGCOMM Comput. Commun. Rev.*, **29** (1999), pp. 147-160.
32. T. Sherwood, G. Varghese and B. Calder, A Pipelined Memory Architecture for High Throughput Network Processors, in *Proc. the 30th annual international symposium on Computer architecture*, (2003), pp. 288-299.
33. M. Degermark, A. Brodnik, S. Carlsson, and S. Pink, Small Forwarding Tables for Fast Routing Lookups, in *Proc. ACM SIGCOMM*, (1997), pp. 3-14.
34. W. Eatherton, G Varghese, and Z Dittia, Tree Bitmap: Hardware/Software IP Lookups with Incremental Updates, *SIGCOMM on Computer Communication Review*, **34**(2) (2004), pp. 97-122.
35. Xianghui Hu, Xinan Tang, and Bei Hua, A High-performance IPv6 Forwarding Algorithm for a Multi-core and Multithreaded Network Processor, in *Proc. the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, (2006), pp. 168-177.
36. E. Spitznagel, Compressed Data Structures for Recursive Flow Classification, *Technical Report, WUCSE-2003-65*, (2003).
37. D. Liu, B. Hua, X. Hu, and X. Tang, High-performance packet classification algorithm for many-core and multithreaded network processor, in *Proc. international conference on Compilers, architecture and synthesis for embedded systems*, (2006), pp. 334-344.
38. S. Han, K. Jang, K. Park and S. Moon, PacketShader: a GPU-accelerated Software Router., in Proc. *ACM SIGCOMM*, (2010), pp. 195-206.
39. C. L. Hung, C. Y. Lin, and H. H. Wang, An Efficient Parallel-Network Packet Pattern-Matching Approach

Using GPUs, *Journal of Systems Architecture*, **60**(5), (2014), pp. 431-439.

40. N.K. Govindaraju, S. Larsen, J. Gray and D. Manocha, A memory model for scientific algorithms on graphics processors, In Proc. *the ACM/IEEE conference on Supercomputing*, 2006:89