2012 IEEE 14th International Conference on High Performance Computing and Communications

Efficient Packet Pattern Matching for Gigabit Network Intrusion Detection using GPUs

Che-Lun Hung
Dept. of Computer Science & Communication Engineering
Providence University
Taichung, Taiwan
clhung@pu.edu.tw

Hsiao-hsi Wang, Chin-Yuan Chang
Dept. of Computer Science & Information Management
Providence University
Taichung, Taiwan
hhwang@pu.edu.tw, a3014006@gmail.com

Chun-Yuan Lin
Department of Computer Science and Information Engineering,
Chang Gung University,
Taoyuan, Taiwan
cyulin@mail.cgu.edu.tw

Abstract—**With the rapid development of network hardware technologies and network bandwidth, the high link speeds and huge amount of threats poses challenges to network intrusion detection systems, which must handle the higher network traffic and perform more complicated packet processing. In general, pattern matching is a highly computationally intensive process part of network intrusion detection systems. In this paper, we present an efficient GPU-based pattern matching algorithm by leveraging the computational power of GPUs to accelerate the pattern matching operations to increase the over-all processing throughput. From the experiment results, the proposed algorithm achieved a maximum traffic processing throughput of 2.4 Gbit/s. The results demonstrate that GPUs can be used effectively to speed up intrusion detection systems.**

Keywords-GPU; parallel processing; patttern matching; intrusion dection systems

I. INTRODUCTION

With the repaid development of network hardware technologies and network bandwidth, the high link speeds provide various platforms and web services to satisfy varied requirements on internet. Cloud computing as a new Internet service concept has become popular to provide various services to user such as multi-media sharing, on-line office software, game and on-line storage. Therefore, the huge amount of threats poses challenges to modern network security systems, which must handle the higher network traffic and perform more complicated packet processing. Network security architectures as such as firewalls and Network Intrusion Detection Systems (NIDS) are utilized to detect malice by monitoring the incoming and outgoing suspicious network packets. Most of current NIDS adopts a set of rules to compare against packets. In general, a rule consists of a filter and a pattern. Filter is used to determine the resulting action that a packet should be dropped or passed according to packet header fields. Pattern is used to search packet payload to find the location where that the pattern is presented, and an associated action is taken if the pattern is found.

Pattern matching is computational intensive process that affects the performance of NIDS. It also occupy about 75% of the total CPU processing time of modern NIDS [1, 2]. Usually, pattern matching algorithms are used to search for matches among a large set of strings from all patterns that apply for a particular packet. Pattern matching algorithms can be classified into single and multiple pattern algorithms.

In single pattern matching algorithms, each pattern is used to search a given string. Knuth-Morris-Pratt [3] and Boyer-Moore [4] are the common-used single pattern matching algorithms. Knuth-Morris-Pratt is able to skip mismatch characters in the comparison phase by using a partial-match table for each pattern. Each table is built by preprocessing every pattern separately. In the Boyer-Moore algorithm, the execution time can be sub-linear when the suffix of the string appears infrequently in the input stream, due to the skipping heuristics that it uses.

Multi-pattern matching algorithms [5, 6, 7] search for a set of patterns in a string in parallel. In such algorithms, a set of patterns is preprocessed to build a state machine, and then the state machine is used scan the string. Each character of the string is searched only once. Multi-pattern matching scales much better than single pattern matching algorithms.

Most of NIDS use finite state machines and regular expressions to detect patterns. These NIDS are developed based on Aho-Corasick [5] and Boyer-Moore algorithms. Coit *et al.* [8] combined the Aho-Corasick trie structure with the skipping feature of the Boyer-Moore algorithm to improve the performance of Snort [9]. Set-wise Boyer-Moore-Horspool algorithm [10] is faster than both Aho-

978-0-7695-4749-7/12 $26.00 © 2012 IEEE
DOI 10.1109/HPCC.2012.235

Corasick and Boyer-Moore algorithms for sets with less than 100 patterns. Tuck *et al*. [11] enhanced the Aho-Corasick algorithm by applying bitmap node and path compression.

To speed-up the pattern matching process, specialized hardware can be suitable to be used to improve performance of NIDS. Sidhu and Prasanna [12] implemented a regular expression matching architecture based on FPGAs. Baker *et al*. [13] also investigated efficient pattern matching as a signature based method on FPGAs. Attig *et al*. [14] developed a framework to process packet header and scan payload content on FPGAs. However, it is high cost to adjust a new rule set in such FPGA architectures. It needs to program a new circuit, which is then compiled by using CAD tools. Any change in the rule set requires the recompilation, regeneration of the automation, resynthesis, replacement and routing of the circuits which is a time consuming and difficult procedure.

Network processor architecture is used to pipeline the processing stages as well as the entire implementation of an NIDS on a processor [15, 16]. In addition, network processor clusters have been proposed to process intrusion detection by many processors in parallel. However, the cost is still high since it requires multiple processors, a distribution network, and a clustered management system.

With the rapid development of mulit-core hardware, Graphics Processing Units (GPUs) have been used in many applications to enhance the computational performance. GPUs have low design cost while their increased programmability makes them more flexible than FPGAs. General-Purpose Graphics Processing Units (GPGPU) programming has been successfully utilized in the scientific computing domains which involve a high level of numeric computation. The greatest benefit is that the processing units grow from many (CPU, about 2-16) to massive (GPU, about 128-512). In 2006, NVIDIA proposed the Compute Unified Device Architecture (CUDA). CUDA uses a new computing architecture named Single Instruction Multiple Threads (SIMT) [17]. This architecture allows thread to execute independent and divergent instruction streams, facilitating decision based execution which is not provide for by the more common Single Instruction Multiple Data (SIMD). Jocob and brodley [3] implemented the Knuth-Morris-Pratt algorithm on GPU as a pattern matching engine for NIDS in PixelSnort [18]. However, their performance results indicated insignificant improvement. Giorgos *et al*. [19] developed a modified Snort, called Gnort, by implementing Aho-Corasick algorithm on GPU. They have shown two approaches to search patterns. First, each packet is processed by a specific thread block, executed by on multiprocessor. Secondly, each packet is processed by a different thread. It achieved a maximum traffic processing throughput of 2.3Gbit/s.

In this paper, we propose an efficient pattern matching method to classify huge number of packets simultaneously by using GPGPU device. By leveraging nVidia CUDA device can achieve low cost, commodity GPU co-processors to accelerate processing throughput. We also implement the proposed pattern matching algorithm on a variety of memory architectures on GPU to discuss the performance of proposed method. Furthermore, we take advantage of DMA execution of GPUs to impose concurrency between the operations handled by the CPU and the GPU. The experiment results demonstrate that the proposed method can achieve 10X speed up over the CPU implementation of Aho-Corasick algorithm. It presents that GPUs is useful for improving overall performance of NIDS.

The structure of this paper is as follows. Section 2 introduces the GPU programming. Section 3 describes the proposed method. Section 4 presents the experiment results. We conclude with section 5, providing a brief summary and conclusion.

II. GPGPU PROGRAMMING

As the GPU has become increasingly more powerful and ubiquitous, researchers have begun developing various non-graphics, or general-purpose applications [20]. The modern GPUs are organized in a set of multiporcessors, each of which contains a set of stream processors executing the same instructions on multiple data streams simultaneously.

nVidia released the Compute Unified Device Architecture (CUDA) SDK to assist developers in creating non-graphics applications that run on the GPUs. A CUDA programs typically consist of a unit of work issued by the host computer to the GPUs is called a kernel that runs in parallel on the GPUs. Input data is copied to the on-board memory of the GPUs from host computer memory through the PCI-E bus prior to invoking the kernel, and output data is copied to host computer memory from GPU's 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 writhe 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.

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 1 illustrates the memory architectures of CUDA device.

Global Memory

Global memory is the biggest memory region available on CUDA devices and is capable of storing hundreds of megabytes of data. In the CUDA Fermi architecture [17], the L1 cache per SM multiprocessor is configurable to support the global memory operations. Therefore, the access latency

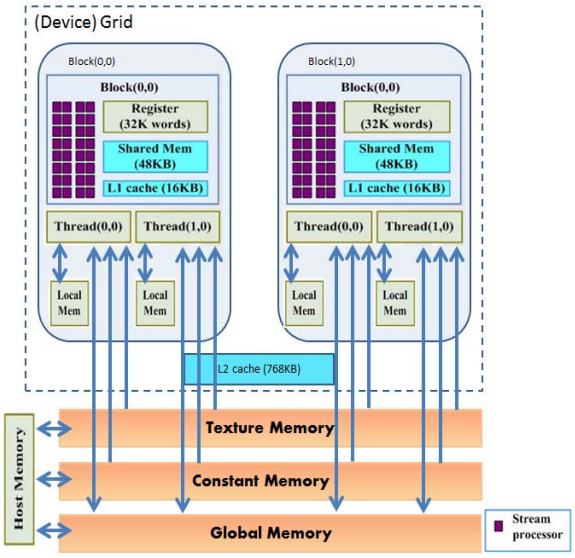


Figure 1. Memory architectures of CUDA device.

of global memory is comparable to other GPU memory architectures.

Constant Memory

Constant memory is a small read-only memory region that resides in DRAM on CUDA devices. It is globally accessible memory for all threads. Since Constant memory has on-chip cache, the access latency is short. The cost of a cache-hit is as a local register access, but the cost of a cache-miss is as a global memory access on devices. The constant memory is limited to its size.

Texture memory

Each multi-processor on the CUDA device equips a 64KB texture cache which can be bound to one or more arbitrarily sized region of global memory. Texture memory is read only as constant memory.

Shared memory

Shared memory is block-local that facilitates cooperation between multiple threads in a thread block. Shared memory is limited to 64KB per multi-processor on CUDA Fermi devices. The access latency of shared memory is equivalent to that of register.

Registers

Each block on CUDA device equips a register file that contains registers. The register provides fast thread-local storage during kernel execution. In the Fermi architecture, each multi-processor contains the amount of 32-bit registers (32,000) that are shared for all threads in the executing thread block.

III. METHOD

In the previous implementations of GPU-based pattern matching algorithms, each thread has different work load. Due to imbalance work load among the threads, the performance of pattern matching is inefficient. Therefore, we proposed a method to balance the work load among the thread, and it performs the pattern matching based on hierarchical hash table architecture on GPU. It consists of three phases: initialization, pattern matching and data output. The initialization phase is to combine a number of packet payloads as a very long string, and then this combination is transferred to GPUs. In the pattern matching phase, each thread searches a fixed window size of combined data. The final phase is to transfer the results to host computer. The overall processing of the proposed method is shown in Figure 2.

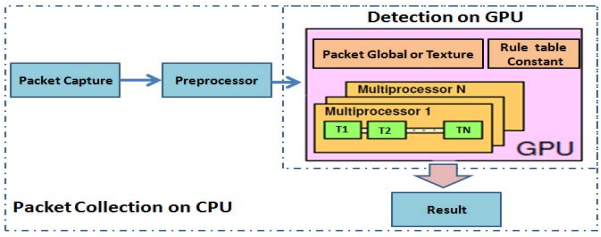


Figure 2. The architecture of the proposed algorithm.

A. Initialization

In the proposed method, the first step is to transfer packet data from the network interface to the memory of GPU device. The simplest approach is to transfer each packet directly to the GPUs. However, the overhead associated with a data transfer operation to the GPUs is very huge.

The performance of host-to-device transferring can be improved by ZeroCopy technique. ZeroCopy allows threads to access main memory on host directly. A special type of memory, called page-locked memory, is allocated in the physical memory of host to implement ZeroCopy. The use of page-locked memory results to higher data transfer throughput between the host and the device [17]. Furthermore, the copy from page-locked memory to the GPU is performed using DMA, without occupying the CPU, and threads can access page-locked memory though PCI-E bus.

In our approach, a number of packet payloads are combined to form a long packet payload, and then this long packet payload is copied to memory on GPU device. Each thread searches in n bytes where n is the maximum pattern length. Therefore, the total size of combined packets is

$$L = n \times T,$$

where L is length of combined packets and T is the number of threads. Then, the combined packet data is copied to GPU memory. An advantage of combination of packets into a long string is that all threads are assigned the same amount of work, so execution does not diverge, which would hinder the SIMT execution. Figure 3 shows the use of the memory on CUDA device.

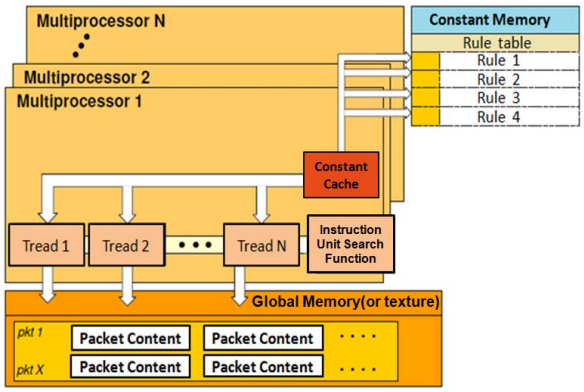


Figure 3. The use of memory on CUDA device.

B. *Pattern Matching processing on GPU*

In this step, the pattern matching processing is used to compare the packet data with patterns. Due to the property of branch instructions of GPU, the hash table strategy seems to be a perfect candidate for SIMT processors. Therefore, a simple and efficient pattern matching algorithm based on hierarchical hash table architecture is developed in this work.

In the hierarchical hash table architecture, the hash table in each level is stored as a one-dimensional array. The number of level is equal to the length of the pattern. Each cell of the hash table is the key to the entry of hash table in next level. In case the hash table is the final level, the corresponding cell will contain the unique identification number of the matching pattern, otherwise zero. A drawback of this structure is that hash tables are sparsely populated when the number of patterns is small or patterns can be classified in few groups by prefix or suffix of patterns. However, the some efficient storage structures to compress the memory space are much more complex to be implemented in GPU device. Figure 4 presents example of the proposed algorithm. The hierarchical hash tables are constructed in host memory by the CPU, and are then copied to constant memory that is accessible directly from the GPU.

In the proposed algorithm, each packet is spitted into fixed equal parts and each thread searches each portion of the packet in parallel. As a pattern matching example shown in figure 4, the first pattern is "CDEF" and a packet data is spitted by four characters. In case the substring "ABCD" processed by first thread (thread 0), the first character "A" is not searched in the first pattern and then the value 0 is returned. The third thread (thread 2) handles the substring "CDEF". According to hash functions, the corresponding positions in hierarchical hash tables are found, and then the final value 1 is returned. The results are stored in the result table that each entry records the pattern id and position of found pattern in the combined payload. The pseudo code of the proposed algorithm is shown in figure 5.

C. *Data Output*

The result table has been allocated and stored in the GPU device memory. This table is copied to the host memory after pattern matching execution has been completed. It is easy to

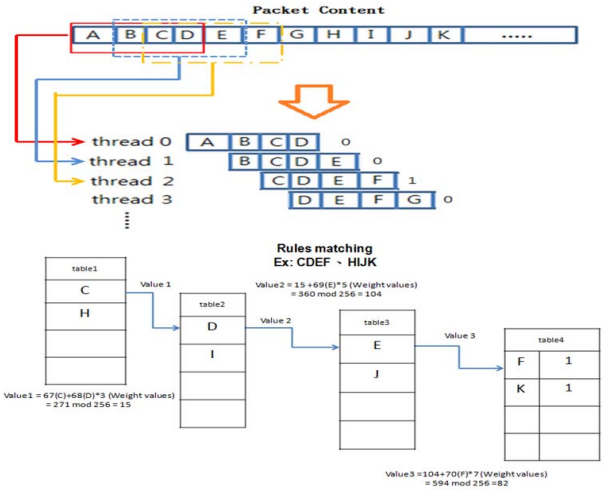


Figure 4. The example of pattern matching parallelization approach.

Content Match the method

```
Input : GpuSrcPacket , GpuDstPacket, con_eqID, con_RuleIndex, con_Ruletable, con_mask
            /* d_data_pk: Packet Content
                d_result: Calculation results
                All filter rules are saved in constant memory
                RuleSize : The number of filter rules
                __constant__ u_char con_key1[RuleSize];
                …
             */
Output :  result
Method
Begin
1.              int idx = blockIdx.x * blockDim.x + threadIdx.x;
2.              int idy = blockIdx.y * blockDim.y + threadIdx.y;
3.              Int  cloum = packet size
4.           d_result [idx]= con_key4[con_key3[(con_key2[(con_key1[d_data_pk[idy*cloum +idx]*1
   +[d_data_pk[idy*cloum
                                          +idx+1]*3) %256]  +d_data_pk[idy*cloum +idx+2]*5)
   %256]+[d_data_pk[idy*cloum +idx+3]*7 ];
5.              if (idx < RUN_NUMBER)
6.                    for j = 0 to packet size
7.                      result[ idx ] = result[ idx ]| d_result[j];
8.              end
End
```

Figure 5. Pseudo code of pattern matching parallelization approach.

find which packet matches a specific pattern from the result table. In the initialization stage, the length of each packet payload is recorded in a packet offset table. Therefore, the matched packet can be found by subtract the packet offsets from the position of the combined payload. This process is executed in the host.

IV. EXPERIMENT

We implemented the proposed algorithm on single NVIDIA GeForceGTS 450 graphics card (Fermi architecture) and installed in a PC with an Intel i3 540 3.07 GHz CPUs and 8GB DDRIII-1333 RAM running the Linux operating system. In the experiment, the size of each packet is fixed to 1536 bytes. The packets are merged together and are copied to GPU memory. The maximum size of merged packets is 40,000 bytes. In this experiment, the number of processed packets is

$$P_n = T \times 4 / 1536,$$

since each thread processes 4 bytes data.

TABLE I. COMBINATION OF VARIOUS MEMORY ARCHITECTURES ON CUDA DEVICE

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

First, we measure the scalability of the proposed algorithm implemented by storing patterns and packet payloads on various GPU architectures which are shown in table 1. 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, the packet data is copied to global and texture memory and the patterns are copied to constant memory. We then compare the performance of the various algorithms for different number of patterns and packet sized.

In this experiment, we evaluate the performance of different implementations of the proposed algorithm on various GPU memory architectures. The tested patterns are randomly generated which size is 4 bytes and the number of patterns is 1200. Figure 6 shows the speedup of implementations of the proposed algorithm on GPU (displayed as GPU1, GPU2, GPU3 and GPU4 corresponding to table 1 in the figure) compared to Aho-Corasick algorithm implemented on CPU. All implementations on GPU reach 11 fold speed-up compared to Aho-Corasick algorithm on CPU. Furthermore, the proposed algorithm implemented with Zero-Copy outperforms other implementations on GPU. Figure 7 shows the execution time for each of implementations on GPU. The result presents that the implementation on GPU which uses global memory with Zero-Copy can achieve the minimum execution time among other implementations.

In the early CUDA devices, cache mechanism was not equipped for Global memory, and only texture memory has 8k cache memory. Therefore, the performance of accessing Texture memory is better than that of accessing Global memory. But, the size of Texture memory is much less than the size of Global memory. In new CUDA Fermi architecture, Global memory equips 768kb L2 cache memory. Therefore, the packet data can be copied to Global memory without considering the cache and memory size. We evaluated how each detection algorithm scales with the number of patterns. Figure 8 shows the throughput achieved for various implementations respectively, to perform pattern matching through pattern-sets of size 100 up to 1200. As shown in Figure 8, the proposed algorithm implementations outperform CPU implementation of Aho-Corasick algorithm. Obviously, in the case of the proposed method, the throughput remains constant independently of the number of

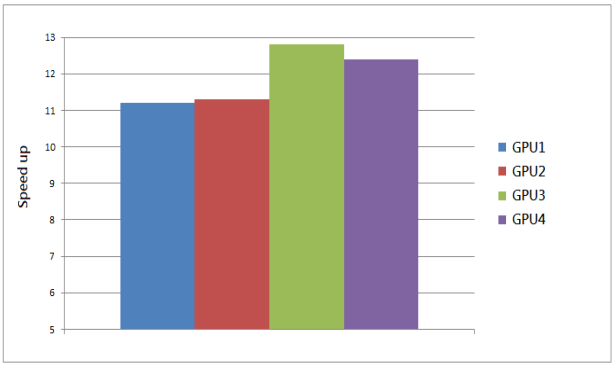


Figure 6. Speed-up sustained for the proposed algorithm comparing to CPU implementation.

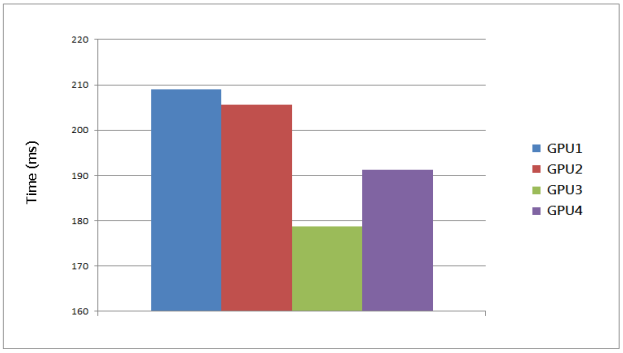


Figure 7. Speed-up sustained for the proposed algorithm comparing to CPU implementation.

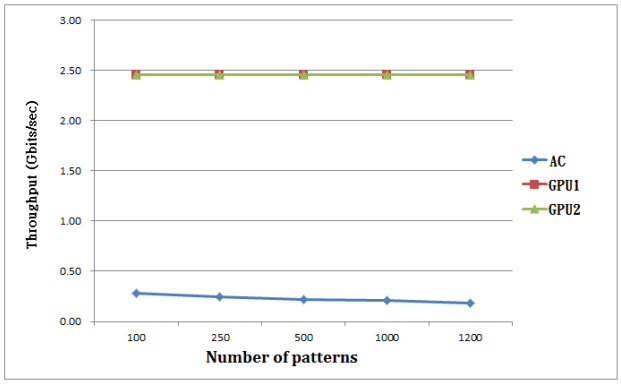
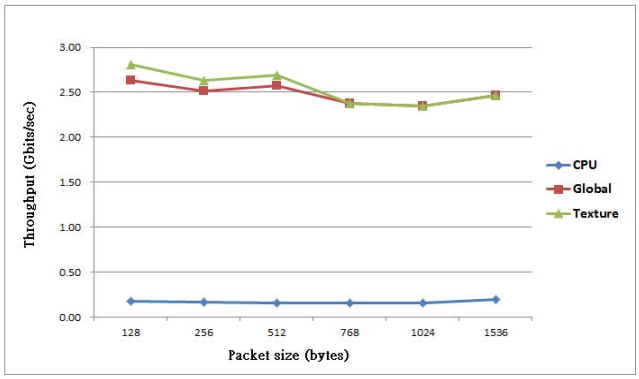


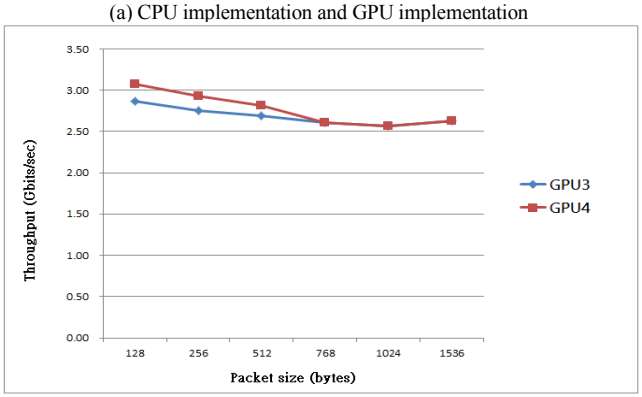
Figure 8. Throughput sustained for the proposed algorithm comparing to CPU implementation in different number of patterns.

patterns, a behavior expected for a multi-pattern method. The throughputs of both of GPU implementations are approximate to 2.5Gbits/S.

Figures 9 and 10 show the throughput achieved for various UDP packet sizes. Each packet contains random data. CPU implementation of Aho-Corasick algorithm presented a stable performance of around 0.2 Gbit/s. the throughput of our implementations reached over 2.3 Gbit/s, giving a total speed-up of 11.x compared to the CPU implementation. Since the exact packets matched the patterns are calculated



(a) CPU implementation and GPU implementation



(b) GPU implementation

Figure 9. Throughput sustained for the proposed algorithm comparing to CPU implementation in different size of packets.

in final stage in the host, the performance is affected by the packet size.

V. CONCLUSTION

In this paper, we presented a GPU-based network intrusion detection system that offload pattern matching processing from CPU. We developed hierarchical hash table architecture to perform the pattern matching operations on GPUs. From the experiment results, the proposed algorithm achieved a maximum traffic processing throughput of 2.4 Gbit/s.

In the future work, we will modify the current data structure to perform the multi-pattern matching operations in various pattern lengths simultaneously. We also plan to use multiple GPUs as GPU cluster to support multiple Network Intrusion Detection Systems.

ACKNOWLEDGMENT

This research was partially supported by the National Science Council under the Grants NSC-99-2632-E-126-001-MY3.

REFERENCES

[1] S. Antonatos, K. Anagnostakis, and E. Markatos, “Generating realistic workloads for network intrusion detection systems,” Proc. ACM Workshop on Software and Performance, Jan. 2004, pp. 207-215.

[2] J. B. D. Cabrera, J. Gosar,W. Lee, and R. K. Mehra, “On the statistical distribution of processing times in network intrusion detection,” Proc. IEEE Conf. on Decision and Control, Dec. 2004, pp. 75-80.

[3] D. E. Knuth, J. Morris, and V. Pratt, “Fast pattern matching in strings,” SIAM J. on Computing, vol. 6, 1997, pp. 127-146, 1977.

[4] R. S. Boyer and J. S. Moore, “A fast string searching algorithm,” Communications of the Association for Computing Machinery, vol. 20, Oct. 1977, pp. 762-772.

[5] A. V. Aho and M. J. Corasick, “Efficient string matching: an aid to bibliographic search,” Communications of the ACM, vol. 18, June 1975, pp. 333-340.

[6] B. Commentz-Walter, “A string matching algorithm fast on the average,” Proc. Intl. Colloquium on Automata, Languages and Programming, 1979, pp. 118-131.

[7] S. Wu and U. Manber, “A fast algorithm for multi-pattern searching,” Technical Report TR-94-17, 1994.

[8] C. Coit, S. Staniford, and J. McAlerney, “Towards faster string matching for intrusion detection or exceeding the speed of Snort,” Proc. DARPA Information Survivability Conference & Exposition II, June 2001, pp. 367-373.

[9] M. Roesch, “Snort: Lightweight intrusion detection for networks,” Proc. USENIX LISA Systems Administration Conference, Nov. 1999, pp. 229-238.

[10] M. Fisk and G. Varghese, “Applying fast string matching to intrusion detection,” Technical Report In preparation, successor to UCSD TR CS2001-0670, 2002.

[11] N. Tuck, T. Sherwood, B. Calder, and G. Varghese, “Deterministic memory-efficient string matching algorithms for intrusion detection,” Proc. IEEE Infocom Conf., 2004, pp. 333-340.

[12] R. Sidhu and V. Prasanna, “Fast regular expression matching using FPGAs,” Proc. IEEE Symp. Field-Programmable Custom Computing Machines, 2001.

[13] Z. K. Baker and V. K. Prasanna, “Time and area efficient pattern matching on FPGAs,” Proc. ACM/SIGDA Intl. Symp. Field Programmable Gate Arrays, 2004, pp. 223-232.

[14] M. Attig and J. Lockwood, “A framework for rule processing in reconfigurable network systems” Proc. Annual IEEE Symp. Field Programmable Custom Computing Machines, 2005, pp. 225-234.

[15] H. Bos and K. Huang, “Towards software-based signature detection for intrusion prevention on the network card,” Proc. Intl. Symp. Recent Advances in Intrusion Detection, Sept. 2005.

[16] D. L. Cook, J. Ioannidis, A. D. Keromytis, and J. Luck, “Cryptographics: Secret key cryptography using graphics cards,” Proc. RSA Conference, Cryp-tographer’s Track, 2005, pp. 334-350.

[17] Nvidia, “cuda c best practices guide”, version 4. Online, March 2011.

[18] N. Jacob and C. Brodley, “Offloading IDS computation to the GPU,” Proc. IEEE. Annual Computer Security Applications Conf., 2006, pp. 371-380.

[19] G. Vasiliadis , S. Antonatos , M. Polychronakis , E. P, and S. Ioannidis, “Gnort: High performance network intrusion detection using graphics processors”, Proc. Intl. Symp. Recent Advances in Intrusion Detection, 2008, pp. 116-134.

[20] 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, vol. 26, 2007, pp. 80-113.