GPU-based Multi-stream Analyzer on Application Layer for Service-oriented Router

Kazumasa Ikeuchi, Janaka Wijekoon, Shinichi Ishida, Hiroaki Nishi
Nishi Laboratory, Graduate School of Science and Technology, Keio University, Japan
{ikeuchi, janaka, sin}@west.sd.keio.ac.jp, west@sd.keio.ac.jp

***Abstract*—Service-oriented router (SoR) is a new router architecture for providing rich services to Internet users by utilizing useful information extracted from network traffic. In SoR, stream reconstruction and selection is a fundamental process for providing the services in the application layer. After real-time reconstruction of stream data, SoR used a software character string analyzer to extract important required information. One of the promised services is a router-level network intrusion detection system. Because a network consists of hundreds of thousands of data streams, achieving an intended throughput while analyzing these stream data is a critical problem. We propose an acceleration method of string matching based on a heterogeneous system consisting of a CPU and a graphics processing unit. In addition, we designed and implemented a task controller that improves the distribution of POSIX-thread-based processes so that string matching can be performed concurrently depending on the status of the string matching system.**

Keywords—Service-oriented router; string matching; GPU; application layer analysis

I. INTRODUCTION

A router forwards a packet to another router after receiving it from an end-host. This process is repeated until the packet arrives at a destination end-host. This forwarding process is performed on the basis of both the destination IP address indicated in the packet header and a router forwarding table. A typical router checks only the packet header for seeking the next hop. A security attack is mainly hidden in the packet body as contents of the packet. If a router can control or refuse the forwarding process of a malicious packet to targeted clients autonomously, the risk of clients being intruded over the Internet can be reduced. To achieve this level of network security services, a stream reconstruction function is crucial because TCP/IP divides the original data stream into multiple packets. That is, the router providing the services has to reconstruct data streams from fragmented packets. After that, to find the security attacks from original data streams, a high-throughput string matching function is required. Without these reconstruction and string matching functions, routers cannot obtain the data stream or analyze information from the data stream.

To realize new services including router-level security, we propose a service-oriented router (SoR) [1] as a new router architecture. The SoR reconstructs stream data to extract application layer information and decides the forwarding route of a packet according to the contents of the packet. In the SoR, upon arrival at the SoR, packets are stored in an in-memory database (DB) after pre-processing of TCP/IP stream reconstruction in a network processor (NP). A stream processing engine (SPE), as a software-based string matching function, extracts desired information from stream data stored in the in-memory DB.

To provide a network intrusion detection system (NIDS) as a router-level security application, the SoR should search signatures of malicious packets. When the SoR detects a stream that is trying to attack any other clients, it suspends the forwarding process of the malicious stream, and it can notify the clients that they are the targets of an attack. After the notification, the SoR stores information about the attack in the DB. In addition to the SoR-based NIDS, the SoR can provide more flexible services that can be accomplished only by the SoR, such as a recommendation service that is based on cross-sectional behavioral analysis and efficient content delivery networking that is based on requested content [2].

One of the problems to be overcome in contents-based services on the SoR is that an efficient processing method is needed to handle a large amount of string information, which is not required in typical Layer-3 routing based on the IP address of a packet. The SPE has to handle a number of string data extracted from reconstructed streams. To achieve this, a typical router with an ASIC-based stream processing co-processer can accelerate the string matching function with reduced processing flexibility.

In this study, we present a software-based string analysis system that has both sufficient throughput for string searching and flexibility for providing services. To accomplish flexible and high-throughput processing, we used a graphics processing unit (GPU) as a high-performance processor that has highly parallelized architecture and data structures for achieving effective calculation power. A GPU is typically used for image processing such as image rendering and vertex calculation to create 3D images. However, recently, the opportunities to use the processer for general applications have increased because the requirements of flexible processing and low-cost computation are increasing. NVIDIA, a major provider of GPUs, also provides a flexible GPU-based program development environment called Compute Unified Device Architecture (CUDA) [3] as the development application programming interface (API) of general-purpose GPUs (GPGPUs). In the near future, a high-performance router with conventional PC parts such as the GPGPU will be introduced to the market.

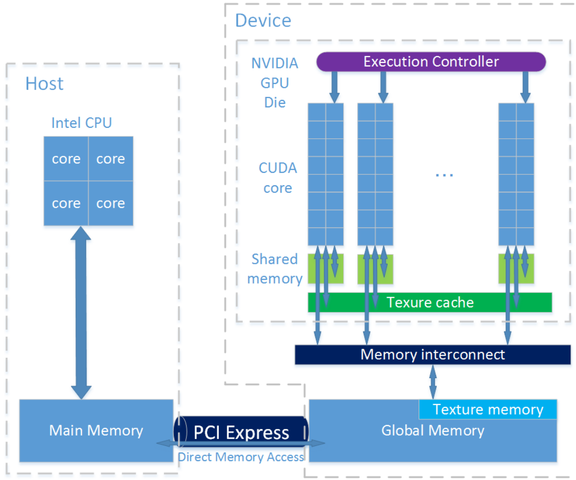

Fig. 1. A typical unified system consists of a CPU and a GPU.

According to the advancement of commoditization of routers, Maxeler Technologies released a new router named MaxNode 10G [4], which can be attached GPUs to its main system to accelerate processing. Moreover, Intel is developing the Intel Data Plane Development Kit [5], which realizes effective packet processing on a commonly used single Intel architecture CPU. In addition, Juniper Networks is developing the JunosV App Engine [6], which enables customization and optimization of network services by virtualizing network applications and providing services on general devices. With the advancement of GPUs and commoditization of router architecture, the possibility is very high that highly functional routers will use GPU-based technology to accelerate application layer information analysis.

Several studies attempted to solve the string matching problem on GPUs, and several new GPU-based string matching methods were recently proposed [11] [12]. According to related works and our preliminary study about GPU-based string matching, we proved that the GPU could achieve high-performance string matching by tuning the algorithm of parallel string matching. However, real-time and high-throughput processing for multiple-stream data of traffic has not been proposed but is indispensable for the Layer-7 analysis on the router. Our aim is to propose a multiple-stream processing algorithm of string matching on a GPGPU, which is also indispensable to the future SoR with conventional devices.

The main contribution of this paper is to provide a software solution using a GPU for string matching with multiple streams and multiple sets of queries extracted from NIDS software. We designed and implemented a GPU-based string matching program on CUDA, and the architecture of a task controller that monitors the status of the stream buffer and GPU resources and schedules processes depending on the status. The task controller uses a POSIX threads (pthreads) library to issue tasks concurrently and an effective asynchronous process scheduling method. POSIX thread, usually referred to as pthread, is a POSIX standard for threads [13]. We tested the combination string matching and task control system in an offline experimental environment and evaluated its performance.

TABLE I. FEATURES OF GPU MEMORIES

Memory	Location	Speed	Cache	Accessibility	Capacity
Register	on die	fast	n/a	read/write	16 KB/block
Shared	on die	fast	n/a	read/write	48 KB/block
Texture	off	fast (cache hit)	spatial cache	read only	n/a
Global	off	slow	yes	read/write	2,024 MB

This paper is arranged as follows. We explain features of the GPU and the parallel computing platform on CUDA in Section 2. We present related works about multiple string matching and adaptation of the string matching to the GPU architecture in Section 3 and describe the problem, goal, setup, design of the task controller, and implementation in Section 4. We evaluate our experimental results in Section 5 and finally conclude the paper with possible future works in Section 6.

II. ARCHITECTURE OF GRAPHICS PROCESSING UNIT

A GPU has more than 100 of cores and several types of memories that have different access speeds and cache mechanisms. A GPU typically performs well when processing highly parallelized datasets such as in image rendering. In addition, the software design using GPU technology becomes more general in parallel processing, such as in financial simulation and genetic analysis. This design paradigm is called GPGPU. NVIDIA has provided a unified development environment named CUDA, which is a programming language used to design and program parallel processing for both the CPU and the GPU. CUDA uses general C/C++ code and the NVCC compiler, which is provided by NVIDIA to compile CUDA code. NVCC compiles the unified code and generates two types of executable code: one is host code that is executed on the CPU and the other code is kernel, which is executed on the GPU. After the host transfers the compiled kernel codes to the GPU via a PCI Express I/O serial interface, multiple GPU cores run the parallel program written in the kernel. All I/O data required in the process have to be transferred via PCI Express.

Figure 1 shows an example of a unified system that mainly consists of a CPU, a PCI Express bus, and a GPU. The GPU has more than 100 processing cores called CUDA cores, and it manages the CUDA cores as units of a streaming processor (SM), which consists of 32 CUDA cores. The CUDA architecture is based on the single-instruction, multiple-thread model, which executes a single instruction with multiple logical threads simultaneously. CUDA executes a parallelized kernel program with 3D threads and blocks that are hierarchically constructed in the grain of execution. A thread is the smallest unit of a logical processing unit. A kernel can employ a number of threads, and management of the threads strongly affects the processing performance. Although this processing strategy is suitable for effective parallel processing, warp divergence, namely fatal performance degradation, occurs when part of a thread in a single warp diverges by executing an

“if” branch instruction. Hence, it is better to avoid warp divergence whenever possible.

A GPU has several kinds of memory, and Table I summarizes features of those memories. Global memory, also called device memory, has the largest capacity, approximately 2–6 GB among the GPU memories. However, the access speed to global memory is extremely slow, which consumes about 400–600 clock cycles. Shared memory has approximately 48 KB of capacity and is located on each block. This memory can be accessed only in a few clock cycles, because it is located on the GPU die. Texture memory shares the memory space of global memory, and the memory has a hardware cache mechanism to accelerate the accesses by using a spatial locality. Effective use of both the hierarchically structured threads and various memories is vital to high-performance processing on a GPU.

III. Multi-Pattern String Matching Algorithm

A string matching algorithm is used for searching multiple text patterns from other text data. String matching algorithms can be classified into two types: 1) string matching for a single pattern and 2) string matching for a pattern set consisting of multiple patterns simultaneously. Single-pattern string matching can be classified into prefix matching and suffix matching. One well-known prefix matching algorithm is the Knuth-Morris-Pratt (KMP) algorithm [7]. In the suffix matching algorithm, the matching process is started from the suffix of a pattern. The Boyer-Moore (BM) algorithm [8] is an effective string matching algorithm that consists of two processes, namely, a process to construct failure transition and a process of matching that is started from the suffix of a pattern.

Finding a pattern set that consists of multiple patterns requires some advanced techniques. A potentially possible solution to effective string matching with multiple patterns is the Aho-Corasick (AC) algorithm [10]. We thus used the AC algorithm as the basis of our GPU-based string matching algorithm for the SoR. In the next subsection, we explain how it works, why the AC algorithm can potentially satisfy our purpose, and how it can be applied to the GPU-based string matching system on the SoR.

A. Aho-Corasick algorithm

Many studies have recently been conducted around string matching algorithms. Alfred V. Aho proposed the AC algorithm, which uses a deterministic finite automaton (DFA) constructed by using a pattern set to find desired patterns [10]. The AC algorithm searches all patterns in a single path with $O(N)$ of time complexity, where N indicates the number of text data.

The DFA traverses its transition state to search patterns over a state transition table (STT) depending on the stream of input text. A STT is represented as a 3D table in which rows are indexed by a state and columns are indexed by a possible input character. The AC algorithm uses the STT to define the behavior of the DFA by using the following branches on condition. The first branch is the “goto” function, which defines transition to the next state corresponding to the combination of the current state and an input character. The second branch is the “output” function, which determines whether the next state matches the state of any patterns. When the “output” function returns a matched state, the program outputs the combination of the matched pattern and the position in the input text where the matched pattern exists. The third branch is the “failure” function, which defines the state transition corresponding to the current state, regardless of input character whenever the “goto” function reports a failure of state transition to the automaton. Figure 2 shows an example of DFA associated with five patterns {“he,” “his,” “she,” “her,” “hers”}. The solid lines in Figure 2 indicate transition of the “goto” function, the dashed lines indicate transition defined as a failure transition on the “failure” function, and at double-circled nodes, the “output” function has reported a match of a specific pattern.

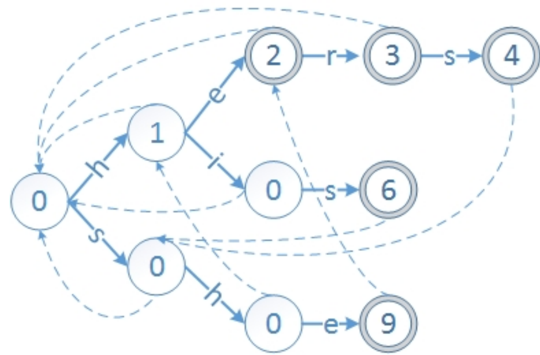

Fig. 2. An example behavior of DFA associated with five patterns {“he,” “his,” “she,” “her,” “hers”} in AC algorithm.

When implementing the AC algorithm to a GPU program written in CUDA directly, a couple of problems interfere with effective matching. The first problem is that the branch on condition leads to significant degradation of throughput. Because processing performance is highly dependent on the divergence of threads, we should manage our GPU program to minimize the number of “if” branches. In addition, because the “goto,” “failure,” and “output” functions cause performance degradation, these functions should be eliminated. The second problem is management of memory allocation and memory accesses. The AC algorithm requires many kinds of memory accesses, such as reading the character from an input stream and next transition from the STT and writing the matching result account for those memory accesses. Optimization of these memory accesses is required to exploit resources of the GPU and to improve performance.

B. Adaptation of the Aho-Corasick algorithm for a GPU

To address the problems of warp divergence and memory access described above, Lin et al. proposed the Parallel Failure-less Aho-Corasick (PFAC) algorithm as a GPU-based multiple-string matching algorithm [11]. In the PFAC algorithm, the string matching process is performed with a pre-constructed PFAC automaton, which traverses the STT only by the “goto” function. This method eliminates the warp divergence caused by branch instructions of failure transitions.

The PFAC automaton can be represented only by two branches: one is a state transition defined by the "goto" function, and the other is the "output" function. In this method, however, another problem occurs because the "failure" function lacks a "backtrack" function. Figure 3 shows behavior of the DFA associated with the five patterns in the PFAC algorithm. The PFAC automaton cannot detect patterns that appear in the middle of a matching process of a thread, and this is a fatal problem. To compensate for the lack of a "backtrack" function, every thread begins the matching process from every character of the text data as a starting point correspondingly. A searching process of each thread is terminated when the thread fails to match patterns. This matching method uses each character in every position of text data as a starting point. Thus, the method eliminates any misdetection of pattern that is caused by the lack of a "backtrack" function.

It is important to optimize memory accesses, especially in the PFAC algorithm. Memory accesses in the PFAC algorithm are classified into three parts. The first part entails reading a byte of character from an input stream. The second part consists of reading the number of next states corresponding to the combination of current state and an input character, and the third entails writing a matching result after the matching process is terminated. In the PFAC method, the STT is allocated in texture memory, which has a cache mechanism to accelerate accesses by using a spatial locality. HTTP traffic shows strong spatial locality because the traffic data information is represented in structured languages, such as XML and HTML. Threads frequently access columns associated with specific characters of the STT. Furthermore, the frequent access to rows of the STT associated with the characters is also dominant.

IV. Design of Task Controller and Implementation

A. Design of task controller

For a string matching function on a SoR, throughput and latency are the most important factors to be considered in order to provide a better user experience. Another requirement of the string matching on a SoR is fault avoidance. The faults in the string matching process are situations where some of the matching processes in the device terminate in failure. The faults can occur when there is not enough memory in the device. To improve the actual throughput of the string matching process, and to avoid faults, we propose a task controller that handles multiple-string matching tasks depending on the status of the host and the device.

When designing the task controller, we found it imperative to consider the reasonable architecture and parameterization of the status of the system: CPU memory usage, cache hit ratio, etc. The total performance of the proposed PFAC automaton strongly depends on this architecture. The statuses of the system that possibly affect the throughput of the process are a construction of the PFAC automaton, data transfer between host and device, and kernel execution on device. Every single process of the PFAC method has already been optimized sufficiently at the algorithmic level. On the other hand, actual use of the PFAC method for a string matching function on a router requires optimization of the task handling method.

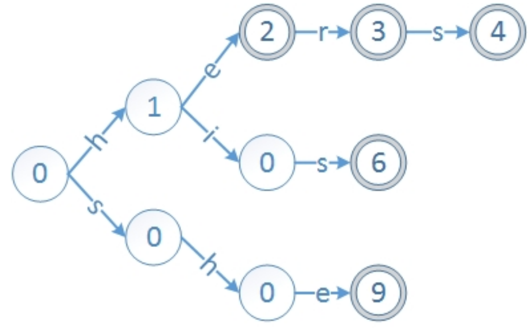

Fig. 3. An example behavior of the DFA associated with five patterns {"he," "his," "she," "her," "hers"} in the PFAC algorithm.

As a first step toward improving the task handling method, the task controller distributes multiple-string matching processes to multiple pthreads in order to process the multiple tasks effectively. The task assigned to a pthread is executed by the same single pthread, and the task controller is issued the multiple pthreads asynchronously. The task controller improves the overall throughput of the string matching processes by processing the multiple tasks concurrently in the method. As a second step to improve throughput, we set a threshold for the amount of accumulated data in the stream buffer. The string matching process consists of kernel execution and data transfer between host and device. The processing throughput can be maximized when enough stream data are stored in the buffer. The amount of accumulated stream data in a single request of the string matching process significantly affects the throughput of the execution of the string matching. Therefore, the task controller issues the ignition request of the string matching process to a pthread only when the amount of data in the stream buffer exceeds the threshold. In this method, each pthread can exploit the resources of the device.

In order to reduce the processing latency of the string matching process, we also set a threshold of waiting time for completing input stream data. If possible, the task controller waits until the stream buffer is filled with input stream data up

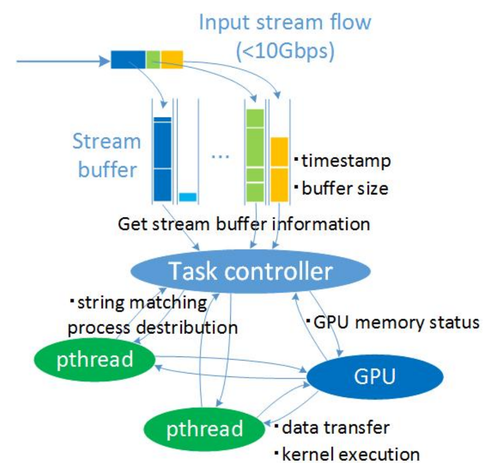

Fig. 4. Implementation of process destribution and monitoring mechanisms.

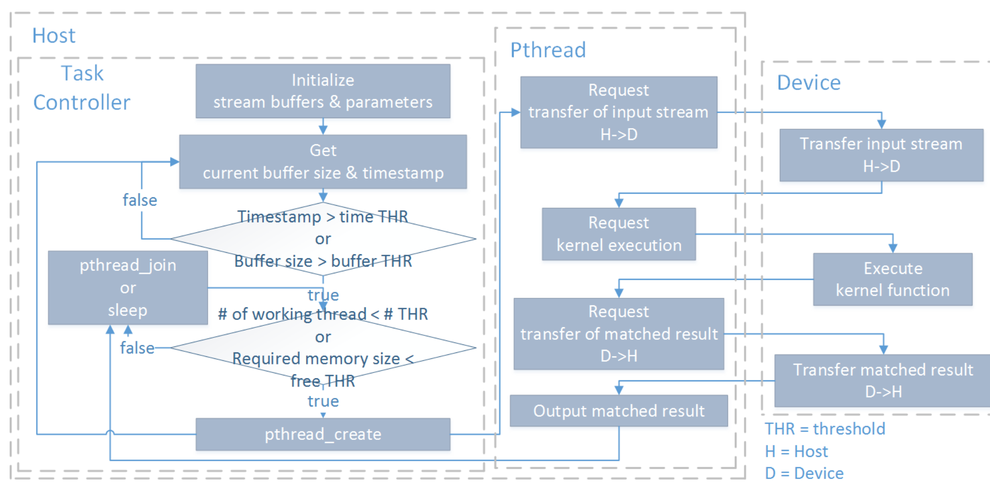

Fig. 5. Flowchart of process destribution and monitoring mechanisms of overall systems.

TABLE II. BASIC INFORMATION OF EXPERIMENTAL COMPONENTS

Component		Details
Host	CPU	Intel Core i7-3930K CPU @ 3.2 GHz
	Memory	32 GB DDR3 @ 1,600 MHz
Device	GPU	NVIDIA GeForce GTX 680
	Memory	2,048 MB 256-bit-GDDR5 memory

to a timeout. However, in terms of delay, the task controller should not wait for a long time because the waiting time for the completion of input stream data is added to the total delay of packet forwarding and occasionally causes degradation in user experience. When the waiting time exceeds a timeout, the task controller requests the string matching process to a pthread, and the device executes the process regardless of the amount of accumulated data in the buffer. On the other hand, waiting time timeouts that are too short may also cause deterioration of throughput and exhaustion of both GPU resources and the PCI Express bandwidth. One of our purposes is to decide the optimal value of the threshold practically through experiments.

On GPU-based string matching, a fault, such as lack of device memory at runtime, is a critical problem. Because the amount of available device memory is limited to several gigabytes, GPU-memory management is vital to improving fault avoidance. Thus, to maintain fault avoidance during the string matching process, we have to both manage and control usage of device memory. Accurate control of the device memory is necessary because it influences the number of kernels executing on a GPU simultaneously. To manage and control the usage of the device memory, the task controller provides current memory usage feedback to the host. When the task controller launches the pthread to execute a new string matching process, the controller checks the memory usage. The controller estimates the amount of memory needed to execute a new string matching process according to the stream buffer size. Moreover, the controller compares the amount of available memory with the amount of memory required to execute the new process. If the amount of available memory does not meet the requirement, a new pthread is not launched until enough memory is released by the finishing pthreads.

The task controller operations for managing throughput, latency, and scalability are presented in Figure 4 as an architecture and in Figure 5 as a flowchart. After initializing stream buffers and pthreads, the controller obtains the states of the buffers and a timestamp. When a timeout occurs or the amount of the stream buffer exceeds the threshold, a condition flag becomes true, which indicates whether the stream data should be processed or not. If the task controller detects a stream whose condition is true, the controller checks the status of device memory. If all conditions are satisfied, the task controller calls the "pthread_create" API to create a new pthread to request a string matching process of accumulated stream data. If the conditions are not satisfied, the task controller waits for a timeout. Finally, the controller calls the “pthread_join” API to terminate a pthread.

B. Implementation

Table II shows specifications of our implemented experimental environment. The host components have an Intel i7-3930K CPU and 32 GB of Double-Data-Rate 3 (DDR3) main memory. The device consists of an NVIDIA GeForce GTX 680 GPU. The host and the device are connected via a PCI Express 2.0 x16 interface, which has 16 GBps of bidirectional bandwidth. Our experimental software environment included a task manager program written in C/C++ language with a pthreads library, and an NVCC compiler that compiles the program. We used the CUDA toolkit to develop an application that provides a comprehensive development environment for C/C++ developers to use to build GPU-accelerated applications. The toolkit and compiler run on CentOS 6.3 on a 64-bit Linux system. In addition, we used Snort Rules [14] as a desired pattern set. Snort is open-source signature matching NIDS software. We extracted signatures specified by a “content” statement and used the signatures as malicious patterns to be searched. As input text streams for the experiment, we used traffic data captured from December 5 to 12, 2011, by the gateway that connects the Nishi Laboratory to the Internet.

TABLE III. RATIO OF STREAM SIZE IN TRAFFIC

Data size	Percentage
>16 MB	62.1
>8 MB	66.1
>4 MB	72.1
>2 MB	76.8

V. EVALUATION

We first evaluated the throughput of the string matching process in terms of the number of concurrent pthreads. Figure 6 shows the throughput of each kernel on a device when the stream size and concurrent numbers of pthreads are varied. In Figure 6, the horizontal axis corresponds to the amount of stream buffer to be processed, and the vertical axis corresponds to the throughput. In this case, the controller issues up to eight pthreads concurrently. The processing throughput for small-sized stream data is poor regardless of the number of concurrent streams. This is because the constant overhead to launch a kernel is relatively large. On the other hand, the string matching process for the stream buffer that stores more than 16 MB of data achieved a throughput of up to 108 Gbps, and this throughput is 3.4 times faster than the performance achieved in a previously published study [11]. On the other hand, the processes with four and eight concurrent pthreads show poor performance, at 49.8 Gbps and 14.2 Gbps, respectively, in the best case because of the congestion in the execution engine. From the evaluation, we found that well-managed assignment of multiple processes may improve the performance of the kernel function of application layer analysis.

We also evaluated the relationship between the waiting time for the stream buffers and the throughput of the overall task in a pthread (Figure 7). The horizontal axis corresponds to the timeout, and the vertical axis corresponds to the throughput. The processes in a pthread include transferring stream data to a device from a host, kernel execution, and transferring matching results to a host from a device. Each line in Figure 7 shows the throughput of a single process that works with different numbers of concurrent pthreads. The single pthread with longer waiting time showed better throughput, at most 8.0 Gbps under the condition of 12,800 ms of waiting time. In this case, approximately 16 MB of traffic data was accumulated in the stream buffer. The stream buffer with longer waiting time can accumulate more traffic data in the stream buffer. The performance of string matching depends significantly on the amount of stream size, as shown in Figure 7. Table III shows the ratio of the total amount of stream data communicated in streams of 2, 4, 8, and 16 MB. According to the table, streams communicating more than 4 MB occupy 72.1% of the amount of all traffic data communicated in the network. If it is assumed that the task controller performs the process only for streams that communicate more than 4 MB of data, approximately 7.21 Gbps of throughput on average is a requirement. In this experiment, the throughput of the processes with 6,400 ms of waiting time fully meets the requirement except concurrent execution with eight pthreads.

As mentioned above, fault avoidance is an important factor; thus, we implemented three methods memory management and evaluated both the number of errors and the memory usage at runtime. Figure 8 shows the amount of memory used by two

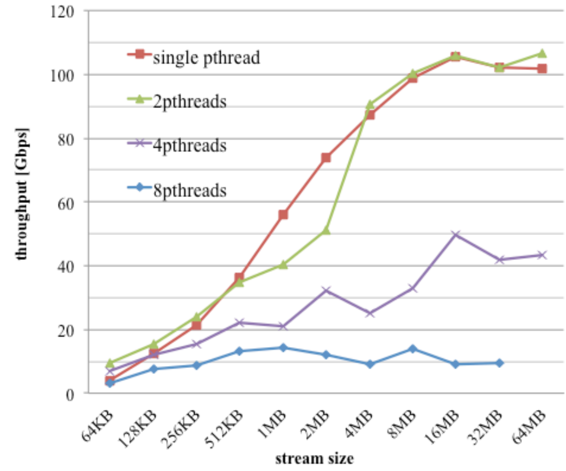

Fig. 6. Comparison of throughputs of processing in the GPU kernel with different numbers of concurrent pthreads.

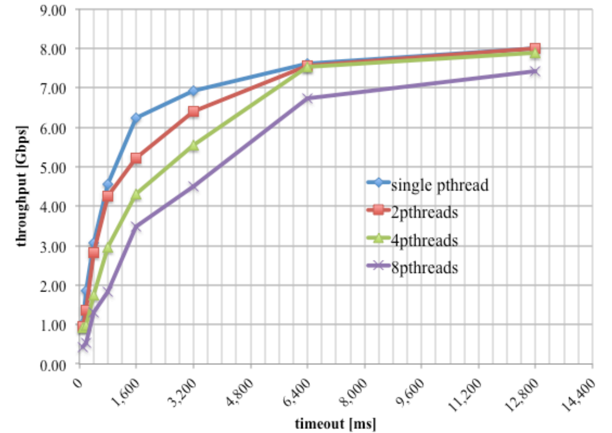

Fig. 7. Relationships between the length of timeout of stream buffer and the throughput of overall processing in each pthread.

pthreads executing different string matching processes concurrently in each management method. The timeout of the stream buffer is set to 400 ms because the evaluation of the relationship between waiting time of the stream buffer and throughput of the string matching process implied that a waiting time of over 400 ms can meet the throughput requirement. The three management methods are titled “no threshold,” “constant threshold,” and “dynamic threshold” in Figure 8. In the “no threshold” results, the task controller issues string matching processes to pthreads regardless of the status of the device memory. When the task controller detects that a stream buffer is filled with enough stream data or that a timeout of the stream buffer has occurred, the controller issues a new process to handle the string matching. The pthreads processing string matching is issued up to the maximum number of pthreads. In this experiment, the memory usage results with the “no threshold” method in Figure 8 indicate that 12 of 829 processes aborted because of an out-of-memory error.

We also tested the “constant threshold” method, in which the task controller can issue new processes only when there is enough available memory in the device. The amount of available memory is constant throughout the whole processes. We set the constant threshold of memory size to 800 MB of free memory. Although doing this eliminated all runtime errors,

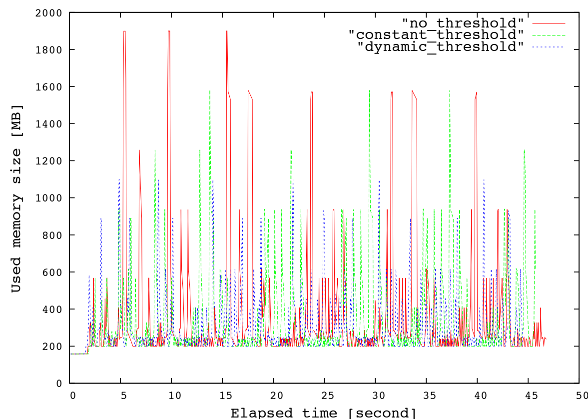

Fig. 8. The memory usage at runtime with three management methods.

the overall execution time increased by 8.3% compared with that of the "no threshold" method.

The third implementation method named “dynamic threshold” resulted in the best proposed fault avoidance method in terms of the tradeoff between fault avoidance and increased overall execution time. In this method, the task controller decides the threshold value depending on the size of the stream buffer waiting to be processed. Before launching a new pthread to process a waiting stream buffer, the task controller obtains the status of the device memory and then compares the amount of available memory on the device and the memory size required for the string matching process and determines whether to issue the process. The memory requirement is determined to be 5 times the amount of the stream data to be processed, as mentioned above. If the amount of available memory is less than the requirement, the task controller does not issue the process. After that, the controller waits 100 ms to accumulate enough data in memory. The frequent reference to the device status deteriorates the throughput of the string matching process because the reference places a burden on the device. With these techniques, we also eliminated all runtime errors and improved the performance reduction to only a 0.8% increase of overall execution time, which is less than one-tenth of an increase with the “constant threshold” method.

VI. Conclusion

We proposed and implemented a GPU-based application layer analyzer for an SoR. This study is the first proposal and implementation that enables the parallel and multiple-stream analysis that is an essential requirement to use the router. The analyzer accelerated a string matching function that is required to analyze application layer information extracted from reconstructed streams on the SoR. The main contribution of the analyzer is a task controller, which is designed to optimize the workload of GPU tasks for the string matching process. The task controller on a host system distributes the string matching processes to multiple pthreads optimally and concurrently by monitoring GPU memory usage, the amount of stream data stored in stream buffers, and timestamp of a stream. The task controller improves overall throughput and latency of the string matching process and fault avoidance.

In our experiment using actual traffic data captured in Nishi Laboratory and NIDS rules extracted from the filtering database used in Snort, we found that the proposed GPU-based string matching achieved up to 108 Gbps of kernel-level throughput when processing 16 MB of stream data. We concluded that 4 MB is an optimal stream buffer threshold size to process the stream data effectively. From an evaluation of the waiting time of the stream buffer, we concluded that 6,400 ms of waiting time is sufficient to achieve at least 7.41 Gbps of throughput required for processing data streams that communicate more than 4 MB of data. To realize fault avoidance of the analyzer, we evaluated GPU memory management methods in three ways. We confirmed that the “dynamic threshold” method shows most stable memory usage without causing fatal errors. In addition, the method suppressed the degradation of overall execution time of the string matching process to only 0.8%.

Acknowledgment

This work was partially supported by Funds for integrated promotion of social system reform and research and development, Ministry of the environment and Grant-in-Aid for Scientific Research (B) (25280033).

References

[1] Inoue, K., Akashi, D., Koibuchi, M., Kawashima, H., and Nishi, H., “Semantic router using data stream to enrich services,” Proc. International Conference on Future Internet Technologies (CFI08) Seoul, Korea, June 18–20, 2008, pp. 20–23.

[2] Wijekoon, J., Harahap, E., and Nishi, H., "SoR based request routing for future CDN," 6th International Conference on Application of Information and Communication Technologies (AICT) 2012, October 17–19, 2012, pp. 1–5.

[3] NVIDIA, Developer Zone, https://developer.nvidia.com/category/zone/cuda-zone.

[4] MAXELER Technologies, http://www.maxeler.com/.

[5] Intel Data Plane Development Kit Video, http://www.intel.com/content/www/us/en/communications/embedded-data-plane-development-kit-video.html?wapkw=data+plane+development+kit.

[6] JunosV App Engine - Network Virtualization Software Platform - Juniper Networks, http://www.juniper.net/us/en/products-services/software/junosv-app-engine/.

[7] Weiner, P., "Linear pattern matching algorithms," IEEE Conference Record of 14th Annual Symposium on Switching and Automata Theory, 1973. SWAT '08, October 15–17, 1973, pp. 1–11.

[8] Moore, J.S., "Introducing iteration into the Pure Lisp theorem prover," IEEE Transactions on Software Engineering, vol. SE-1, no. 3, pp. 328–338, September 1975.

[9] Karp, R.M., and Rabin, M.O., "Efficient randomized pattern-matching algorithms," IBM J. Res. Dev., vol. 31, no. 2, pp. 249–260, March 1987.

[10] Aho, A.V., and Corasick, M.J., “Efficient string matching: an aid to bibliographic search,” Commun. ACM, vol. 18, pp. 333–340, June 1975.

[11] Peng, J., Chen, H., and Shi, S. "The GPU-based string matching system in advanced AC algorithm," 2010 IEEE 10th International Conference on Computer and Information Technology (CIT), June 29–July 10, 2010, 1158–1163.

[12] Zha, Xinyan; Sahni, Sartaj, "GPU-to-GPU and Host-to-Host Multipattern String Matching on a GPU," *Computers, IEEE Transactions on* , vol.62, no.6, pp.1156,1169, June 2013

[13] Information technology -- Portable Operating System Interface (POSIX) -- Part 1: System Application Program Interface, http://www.iso.org/iso/catalogue_detail.htm?csnumber=24426.

[14] Snort, Home Page, http://www.snort.org/.