GPU-based String Matching Method using Warp Shuffle Instructions for Service-oriented Routers

Satoshi Koibuchi, Kazumasa Ikeuchi, Shinichi Ishida, Hiroaki Nishi
Graduate School of Science and Technology, Keio University, Japan
{satoshi, ikeuchi, sin}@west.sd.keio.ac.jp, west@sd.keio.ac.jp

Abstract—Service-oriented Router (SoR), a new router architecture for providing useful Internet services that could not be given by a traditional router. As a service of SoR, to prevent a network intrusion in a network will become a significant service. To attain the service, we proposed SoR-Network Intrusion Detection System (SoR-NIDS) using deep packet inspection (DPI) in order to protect malicious streams on the router. Typical applications like this SoR-NIDS require an effective analysis mechanism of traffic information. Namely, a string matching function is an essential problem. Moreover, router architecture becomes more commoditized. It will be possible in the future to use GPUs for accelerating processing on routers. We propose a new GPU-based string matching design and efficient multistring matching function for multiple streams on a service-oriented router using warp shuffle (shfl) instructions to accelerate data stream analysis. The proposed method was evaluated, and the effectiveness of the method was confirmed.

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

I. Introduction

The Internet has become an indispensable communication tool, and the amount of Internet traffic is continually increasing. Accordingly, the threat of attacks on the Internet is also increasing. In particular, the number of attacks that exploit software vulnerabilities on client PCs is increasing. In general, vendors provide software patches for known vulnerabilities; however, the user is responsible for the installation of such patches, which can contribute to lack of security. In the client–server network model, the administrator of each end-host has discretion over all security; thus, the security level depends on the discretion. Therefore, a new security system that does not depend on the security level of the end-host is required. In future, many sensors will be distributed around the world, and these sensors will not have sufficient battery power, processing power, and memory. In addition, it may be difficult to install antivirus software on such sensors. If these sensors are cracked, a new threat will be introduced. This will strengthen the need for antivirus functionality on the Internet.

A router relays communication between end-hosts in a network. A typical router forwards a packet to the appropriate destination based on a routing table and the destination IP address contained in the packet. We propose a service-oriented router (SoR) as new router architecture. SoR reconstructs TCP streams in a router using the packet information of the plurality of fragments in a network by considering memory efficiency. Moreover, SoR can decode, extract, and analyze application layer information. In addition, based on the results of analysis, SoR can provide a new service. A network intrusion detection system (NIDS) will be one of the new services that SoR can provide, which we refer to as SoR-NIDS. It is possible to increase security of an end-host network by matching with a black list. In addition, NIDS can prevent potential threats and provide warnings to users. In general, NIDS searches for a signature represented as a string or regular expression to distinguish whether the target data can be permitted. SoR-NIDS can be provided to all Internet users. Similar to a general antivirus system, it is possible to provide robust security against new attack methods by updating the blacklist on SoR.

However, a problem must be solved before realizing SoR-NIDS. First, wire-rate processing throughput must be achieved in the router. Second, intrusion detection processing must be realized for multiple streams. A large number of streams flow through a router; thus, it is necessary to achieve high throughput processing for multiple streams. As an existing method, dedicated hardware, such as network processors or FPGAs, have been studied to achieve high throughput [1][2]. However, to reflect recent backbone router trends, the use of conventional cost-effective devices will be a practical solution to achieve SoR-NIDS. In addition, it is preferable to implement programmability and to continue with architecture trends for Internet backbone routers, i.e., commodity devices.

To achieve high-throughput processing of string matching functions, it is indispensable to parallelize the process. Graphics processing units (GPUs) [3] are used as coprocessors to realize high throughput. A GPU has hundreds or thousands of processing cores on one semiconductor die. GPUs can process at high throughput by using these cores in parallel. In addition, a GPU has dedicated memory, and its bandwidth is several times high than main memory. This means that it can obtain higher processor-memory bandwidth than common processors if conditions are appropriate.

In a general system, a GPU is connected through a peripheral component interconnect (PCI) interface that transfers data and instructions. Recently, router architecture
that can connect general-purpose co-processors to the system through a PCI interface have been developed [4, 5], and fast NIDS methods using GPUs have also been studied [6, 7]. Therefore, a GPU can be considered effective as a high-speed method for NIDS processing on SoR.

We propose a fast NIDS processing method for router architecture equipped with a general-purpose GPU and CPU using warp shuffle (shfl) instructions, which are extension instructions of recent GPUs. In this study, we discuss only the problem of string matching that does not include regular expressions because it is possible that a SoR-NIDS service can be provided using only string matching initially. This remainder of this paper is organized as follows. We describe a heterogeneous CPU and GPU system in Section 2. We discuss related string search research in Section 3. In Section 4, we propose a string matching method that is more efficient for multiple streams and discuss algorithmic descriptions using shfl functions. An evaluation of the proposed method is presented in Section 5, and the paper is concluded in Section 6.

II. Heterogeneous CPU and GPU System

Since it is impossible to control an entire program with only a GPU, it is necessary to use the GPU as a co-processor with the CPU. A GPU is connected to a system board through a PCI interface. The main system with a CPU and main memory is called a “host,” and the GPU subsystem connected to the host via a PCI interface is called a “device.” Instructions and all data processing performed on the device must be transferred through the PCI interface from the host. Therefore, processing throughput of the GPU is limited to the PCI interface’s bandwidth. For this reason, GPU processing is preferable because the cost of calculation processing is much larger than that of memory transfer.

GPU processing is based on the single instruction multiple data (SIMD) processing scheme. SIMD processes data to multiple columns of a single instruction sequence. NVIDIA GPUs have adopted parallelism with the SIMD scheme in part by operating 32 processing cores as one operation unit called a streaming multiprocessor (SM). However, NVIDIA released the Kepler architecture [8] in 2012. The Kepler architecture adopts a next-generation streaming multiprocessor (SMX). In an SMX, one operation unit consists of 192 processing cores. An SMX has 192 single-precision arithmetic units, 64 double-precision arithmetic units, 32 dedicated function operation units, and 32 load/store units. In addition, an SMX has four instruction schedulers for each arithmetic unit and eight instruction dispatchers. The instruction schedulers in the Kepler architecture are based on a simple implementation concept. While eliminating hardware stages to avoid data hazards in the calculation data path, the compiler predetermines the dependencies of instructions and incorporates them into the instruction information. An SMX has 65,536 32-bit bandwidth registers available to all cores and 64 KB integrated shared memory and L1 cache. The NVIDIA GTX680 has eight SMXs, i.e., it has 1,536 processing cores that perform in parallel.

GPUs have various types of memory, device memory with large capacity and large access latency, shared memory with small capacity and small access latency, and texture memory for accelerating the access of high spatial locality by mapping a texture region in the device memory.

III. Multipattern string matching algorithm

Here, string matching algorithms that search for a specific string from text is discussed. There are two types of string matching algorithms: string matching using a single pattern and string matching using a pattern set that consists of multiple patterns. Knuth–Morris–Pratt (KMP) and Boyer–Moore are well-known string matching methods that use single patterns. Rabin–Karp and Aho–Corasick (AC) are common string matching methods that use pattern sets. Parallel failure-less AC (PFAC) is a typical and efficient method for string matching on a GPU. PFAC is a specialized AC-based string matching algorithm for GPU architectures.

PFAC can perform a search process with high throughput in linear proportion to the size of a text. It is impossible to predict the data size of a stream in a network. PFAC can cope with a wide range of data sizes and achieve high-performance string matching with SoR. Therefore, we conclude that the PFAC algorithm is the most suitable method for implementing GPU-based string matching on SoR-NIDS.

A. Aho–Corasick algorithm

The AC algorithm [9] is an algorithm extended from KMP. AC uses deterministic finite automaton (DFA). In this paper, DFA used in AC is referred to as the AC automaton, and the table that defines the transition of the automaton is called the state transition table. The AC automaton consists of three processing steps. The first step is a transition function that defines the next state when an expected value is input. The second step is a failure function that defines the next state when an unexpected value is input. The third step is an output function that outputs the location where a pattern is matched. Figure 2 shows the AC automaton built from the pattern set 

\[
\{\text{http}, \text{https}, \text{html}, \text{ssh}, \text{smtp}\}
\]

To avoid complexity, the description of the failure transitions to state 0 is omitted from Figure 1. The state denoted by a double circle indicates that a pattern has been matched and is included in the pattern set. The AC automaton state begins at state 0, receives one character from the first character of text, and continues transitioning until all processing is finished. The termination condition of the process is to reach the end of the searched text. It is indispensable for AC string matching to build an AC automaton as a pre-process. Building an AC automaton process consists of a type of lexicographic pattern set, construction of the state transition table for the transition function, and construction of the state transition table for the failure function. If the type of characters in the pattern set is sufficiently diverse, the calculation cost of these processes is proportional to the number of characters in the pattern set. If the sum of all characters included in the pattern set is \(M\), the calculation cost is \(O(M)\). If the length of the input text is \(m\), the search process cost is \(O(m)\). Therefore, the calculation cost of the entire AC algorithm processing is \(O(M + m)\).

B. Parallel failure-less Aho–Corasick
The PFAC algorithm [10, 11] has been applied to the AC algorithm in GPU architecture. For parallel string matching, if the input text is divided into some pages and one thread explores one page, there is a boundary problem, i.e., a string that exists across pages is not explored. In the PFAC algorithm, each thread searches a substring starting from the position shown in Figure 2. Each thread continues to search until a miss occurs. Processing finishes when miss occurs. In the PFAC algorithm, a boundary problem does not occur because the text of all positions is processed equivalently. Furthermore, since the search processing is conducted from all text positions, it is unnecessary to guarantee consistency such that the pattern matches after a miss–hit occurs. Therefore, the PFAC algorithm can improve performance by reducing failure transitions in the AC algorithm.

IV. Design of string matching methods using a GPU for multiple streams

In this implementation, we propose a method of string matching using shfl functions based on the PFAC algorithm. We also use a task controller [12]. The task controller monitors the status of threads and multiple stream buffers and issues search processes to threads. Processing using a GPU cannot achieve high throughput if the stream size is very small. The task controller determines whether processing is performed by the GPU or CPU according to the status, such as stream buffer size. For GPU processing, a task controller monitors the available capacity of device memory and issues process only if there is sufficient processing capacity. Here, a task controller is a master thread that issues some threads as slave threads. Communication between the master thread and slave threads is established using a global variable. Processing in a thread consists of checking the parameter, searching, and storing the result. A thread determines whether string matching occurs on the GPU or CPU based on parameters from the task controller. If the CPU is specified, the search process is executed on the CPU with the AC method. If the GPU is specified, the search process is executed on the GPU with the extended string matching method based on PFAC.

A. Proposed method to improve PFAC by shfl functions

We propose an extended string matching method based on PFAC using shfl functions. The shfl functions are extended instructions implemented on the Kepler GPU architecture. In conventional GPU architecture, data accessed by multiple threads must be placed in shared memory. This requires two cycles. The shfl functions make it possible to access local variables in one cycle; variables are on the register of threads in a warp. The functions are executed in parallel for all active threads in a warp. Figure 3 shows an example of the shfl functions. The “__shfl” functions access the register of the thread by addressing the specified index. Each thread obtains a certain value as per the ascending/descending order of threads using “__shfl_down”/“__shfl_up” functions. The “__shfl_xor” instruction exchanges the values of the threads using a butterfly method.

In the PFAC algorithm, an input stream is split and stored in shared memory. Threads fetch the data from the shared memory. In the proposed method, threads obtain the stream using shfl functions in the beginning of the matching process. This is performed from shared memory after several cycles of the matching process. Figure 4 shows the flow for obtaining text using shfl functions. First, a series of data that each thread has in the register is collated. Then, all threads in the warp obtain the character using the “__shfl_up” instruction. All threads perform matching regardless of the failure or success of matching to obtain the character consistently. In contrast to PFAC, failed threads in the matching process become inactive. In an evaluation, efficient processing by shfl functions was investigated by analysis of data traffic and rule sets. In the proposed method, there is a trade-off between the character fetching cycle and efficiency of the thread allocation. The shfl instructions can access faster than shared memory. However, if a thread results in matching failure, the thread becomes inactive, and this thread cannot be reused while shfl is used. The matching process will be efficient if the number of failing threads is large and these threads are reused. This trade-off is described further in Section 5. As a result, matching process threads obtain the stream using shfl functions from the first matching process to the fourth matching process and obtain the stream from shared memory from the fifth matching process.

Slave threads transfer a stream to a device, perform a search process using PFAC, and transfer the result to the host. Streams are stored in both a register and shared memory. In a search process, GPU threads load stream data from the register and shared memory, and use the PFAC automaton in texture memory. If the thread finds an expected pattern, the result is stored in shared memory. Then, if there is no valid transition in a GPU thread, the automaton process is finished and all
At Nth matching

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>...</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
<td>e</td>
<td>f</td>
<td>g</td>
<td>h</td>
<td>i</td>
<td>j</td>
<td>...</td>
<td>z</td>
<td>A</td>
</tr>
</tbody>
</table>

At 2nd, 3rd 4th matching

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | ... | 31 |
|---|---|---|---|---|---|---|---|---|---|----|----|----|
| b | c | d | e | f | g | h | i | j | k | ... | A | B | C | D | E | ... | Z | Z |

At N matching

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | ... | 31 |
|---|---|---|---|---|---|---|---|---|---|----|----|----|
| b | c | d | e | f | g | h | i | j | k | ... | A | B | C | D | E | ... | Z | Z

At 1st matching

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
<td>e</td>
<td>f</td>
<td>g</td>
<td>h</td>
<td>i</td>
<td>j</td>
<td>...</td>
</tr>
</tbody>
</table>

---

Fig. 4. Flow of obtaining text using shfl functions

Fig. 3. Example of shfl functions

results are stored in device memory from shared memory. If all GPU threads finish the matching process, the process in the kernel is finished. Then, the results are transferred to host memory. Finally, slave threads provide the results as a global variable to the task controller and all processes of slave threads are terminated.

B. Proposed memory implementation

To perform a string matching process using a GPU, it is necessary to transfer a stream to be searched to device memory. Main memory and device memory have different address spaces; thus, data is transferred using the cudaMemcpy memory transfer API. Since host and device are connected by a PCI interface, the transfer of all processes is performed through the PCI interface. Figure 5(1) shows the flow for transferring data using cudaMemcpy. First, when the host requests that data be transferred from main memory to device memory, data is written back to main memory to guarantee data coherence in main memory. Next, if data coherence is guaranteed, the data is copied to the kernel and the PCI buffer. Then, the data is copied to device memory as a packet through the PCI interface.

In this study, we propose a method to improve transfer delay of stream data by skipping these steps. Recently, the cost of memory has decreased. Thus, it is easy to reserve abundant memory resources for a general-purpose device. We use main memory spaces as physical memory and prohibit swapping to disk by the OS, as shown in Figure 5(2). Therefore, it is possible to skip the write-back of data that is swapped on the disk. This method for using memory space as physical memory is referred to as pinned memory implementation.

In addition, the host does not load from or store to the memory space of the stream while processing occurs on the GPU. Therefore, we propose a method that uses the write-through cache line on the CPU. Thus, it is possible to always maintain data coherence in main memory. This means that it is unnecessary to write back CPU cache when transferring data. This host memory method is called the write-combined (WC) memory implementation. Figure 5(3) shows a summary of data transfer using the WC memory implementation. WC memory implementation improves the data transfer delay in the PCI interface. However, there is a disadvantage: the throughput for loading data from main memory by the host is slower than that of the malloc API.

Device memory and host memory address spaces can be managed as an integrated memory space. Therefore, by determining the data dependencies in the kernel at the time of compiling, data transfer and kernel execution can be processed as a single execution unit. Therefore, a programmer does not need to specify the timing of the transfer. This method is called the mapped memory implementation. Figure 5(4) shows a summary of data transfer using mapped memory implementation.

---

 RAW TEXT END
C. Implementation

The string matching application proposed in this study is expected to perform on a single machine built with a general-purpose GPU. We used an ASUS RAMPAGE 4 EXTREME motherboard with one Intel Core-i7 3930K quad-core processor and 64 GB of DDR3 memory (1,600 MHz operation frequency). The GPU is an NVIDIA GTX680 that has 8 units of 192 SMX cores (1,006 MHz operation frequency) and 2,048 MB of DDR5 memory (3,004 MHz operation frequency). The host and the device were connected through a PCI Express 2.0 ×16 interface. The operating system used was Ubuntu 13.04 with Linux kernel version 3.8.0-35. The program was written in C/C++. The program was compiled with the gcc 4.6.4/g++ 4.7.3 [13] and nvcc 5.0 compilers.

We used two types of rule sets, as shown in Table 1, i.e., the Snort set and the HTTP set. The Snort set [15] consists of 23,139 patterns. The average length of the patterns is 29.0 bytes. We also used the HTTP set, which is generated by selecting the HTTP rules from the Snort set. The HTTP set consists of 2,089 patterns. The average length of the patterns is 13.3 bytes.

We used two types of traffic data, as shown in Table 1, i.e., real traffic captured in the Nishi laboratory and an artificial trace in which a match occurs frequently. A normal trace was captured at the Nishi laboratory by tcpdump [14] of the 1000BASE-T Internet gateway port. This trace was captured on May 27, 2009, and we extracted the HTTP stream, which specifies port number 80 as the destination or source port. The dump file was 14 GB. The extracted L7 information was 11 GB, and the number of streams was 281,479. We created an artificial trace for evaluation in which matching of strings occurs frequently. The artificial trace only consists of patterns squeezed from the Snort set. We considered that the processing throughput becomes low using the artificial trace because the storing process increases with the frequent pattern matches. We used this artificial trace to evaluate the worst performance of SoR-NIDS.

V. Evaluation

Here, we evaluate the method for improving the PFAC algorithm using shfl functions. The failure ratio of GPU threads is considered to affect the performance in the method for obtaining a stream by shfl functions. Here, the active ratio of GPU threads is the ratio of the GPU threads that continue processing to the GPU thread that terminates by a false match. This active ratio is equal to 1 – failure ratio. The active ratio of GPU threads is determined by an input stream and pattern sets. If the prefixes of patterns are likely to match characters of a stream, the active ratio of the GPU thread is high. If the active ratio of GPU threads in warp is high, the number of accesses to inactive threads is reduced. However, using shfl functions to access inactive threads is a disadvantage. Therefore, if the number of accesses to inactive threads is small, the string matching process can be performed effectively.

First, we evaluated the active ratio of GPU threads in the

<table>
<thead>
<tr>
<th>Table 1 Rule Sets and Traffic Environment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
</tr>
<tr>
<td>Snort set</td>
</tr>
<tr>
<td>HTTP set</td>
</tr>
<tr>
<td>Normal trace</td>
</tr>
<tr>
<td>Artificial trace</td>
</tr>
</tbody>
</table>

![Flow of data transfer by proposed memory implementation methods](image-url)
proposed condition, as shown in Figure 6. When the number of times matching occurs is large, the active ratio decreases gradually. In Figure 6, the active ratio is comparatively high from approximately the beginning of the fourth matching process. Therefore, we estimate that the best time to shift from shfl operations to shared memory operations is after the fourth matching process. Table 2 shows the active ratio of GPU threads at fourth matching.

In this condition, we evaluated the difference using malloc, mapped memory, pinned memory, and WC memory implementations. Figure 7 shows the throughput of search processing using the HTTP set and the normal trace. The throughput using shfl function is slightly lower than that without shfl functions. According to the evaluation of active ratio of GPU threads, the active ratio of GPU threads using the HTTP set and the normal trace was always 43% or lower. Therefore, data fetching by inactive threads demonstrated reduced throughput.

Figure 8 shows the throughput of search processing using the Snort set and the normal trace. Overall, the throughput was improved. In this case, the throughput obtained using shfl functions was higher than that obtained without shfl functions. When the stream size was small, the performance improvement was significant. In this case, throughput increased by 11.1% on average. In the mapped memory implementation, the improvement was the highest and throughput increased by 12.0%. If the stream size is increased, performance improvement decreases. For the 128 MB stream, the throughput increased by 2.1%. It is considered that the shfl functions improved throughput because the active ratio of GPU threads was high at the initial search processing when using the Snort set.

Figure 9 shows the throughput of search processing using the HTTP set and the artificial trace. When the stream size was small, the throughput decreased slightly. When the stream size increased, the performance gap narrowed. The throughput decreased by 1.4% to 6.5% when the stream size was smaller than 8 MB. When the stream size was 8 MB or greater, throughput using shfl functions was approximately equal to the throughput without shfl functions. Otherwise, the active ratio of GPU threads becomes low using the HTTP set. In addition, throughput decreased significantly when the stream size was small.

Figure 10 shows the throughput of search processing using the Snort set and the artificial trace. In this combination of rule sets and traffic, the active ratio of GPU threads was the highest among all combinations, and the processing throughput was low because many matches occurred. As a result, the throughput using the shfl functions was higher than that without the shfl functions. Note that the improvement of throughput does not depend significantly on stream size. The throughput increased by 3.2% to 11.4% overall. The performance improvement in the WC memory implementation showed the highest improvement rate (9.1% for all stream sizes on average). Therefore, it was expected that the throughput using shfl functions would be higher than that without shfl functions when the active ratio of GPU threads was high. However, throughput was reduced when the size of the rule sets was small and the active ratio of GPU threads was low.

### VI. Conclusion

In this study, we have proposed a high throughput string matching method using a general-purpose GPU for NIDS on SoR. We proposed and evaluated a fast string matching method using shfl functions and several memory implementation methods. In this implementation, we used and improved a string matching method based on the PFAC algorithm to use shfl functions to improve the performance of a string matching process. The performance was largely dependent on the combination of traffic and rule sets. While the performance with shfl functions was lower than that without shfl functions when the active ratio of GPU threads was low, the performance with shfl functions was 12.0% higher than that without shfl functions when the active ratio of GPU thread was high. Thus, we have confirmed the effectiveness of the proposed string matching method using shfl functions.

### TABLE 2 ACTIVE RATIO OF GPU THREADS AT THE FOURTH MATCHING

<table>
<thead>
<tr>
<th>Function</th>
<th>Normal trace</th>
<th>Artificial trace</th>
</tr>
</thead>
<tbody>
<tr>
<td>Snort set</td>
<td>81.7%</td>
<td>85.2%</td>
</tr>
<tr>
<td>HTTP set</td>
<td>20.0%</td>
<td>34.0%</td>
</tr>
</tbody>
</table>

![Fig. 6. Active ratio of GPU threads](image-url)
Fig. 7. Throughput of search processing using the HTTP sets and the normal trace

Fig. 8. Throughput of search processing using the Snort sets and the normal trace

Fig. 9. Throughput of search processing using the HTTP set and the artificial trace

Fig. 10. Throughput of search processing using the Snort sets and the artificial trace

REFERENCES


