Accelerating Pattern Matching with CUDA Persistent Threads

Chun-Liang Lee⁵,a, Guan-Zhang Chen⁶

⁵Department of Computer Science and Information Engineering, School of Electrical and Computer Engineering, College of Engineering, Chang Gung University, Kwei-Shan, Tao-Yuan 33302, Taiwan
⁶Stroke Center and Department of Neurology, Chang Gung Memorial Hospital, Linkou, New Taipei 20401, Taiwan

*Corresponding Author: cllee@mail.cgu.edu.tw

Abstract

Network security is important not only for personal computers but also for industrial control systems. Network intrusion detection systems (NIDS) protect Internet-enabled devices from cyber attacks by performing deep packet inspection (DPI). The key component of DPI is pattern matching, which scans packet payloads to determine if packets contain malicious contents. In this paper, we propose a method to accelerate pattern matching using CUDA persistent threads. Compared to the traditional method that launch a kernel for every batch of packets, the proposed method only launches one kernel for all packets, and thus can reduce the time cost of launching kernels. Experimental results indicate that the proposed method outperforms the traditional method by a factor of 2 in terms of throughput.

Keywords: network security, deep packet inspection, pattern matching algorithm, network intrusion detection system.

1. Introduction

As the number of Internet-enabled devices increases, the damage caused by cyber attacks continues to increase dramatically. Cyber attacks target not only personal computers but also industrial control systems. One traditional defense against cyber attacks is to deploy firewalls that monitor and filter incoming and outgoing network traffic based on security rules. A security rule defines how a firewall should handle packets with specific header fields, including source/destination IP addresses, source/destination ports, and protocol. However, not all attacks can be detected with firewalls because attacks may reside in packet payloads, which are not inspected by firewalls. Thus, network intrusion detection systems (NIDS)⁶ have been developed to provide advanced security protection. According to a pre-defined set of patterns, a NIDS performs deep packet inspection (DPI), which scans the payload of a packet to determine if the packet contains any patterns. Pattern matching holds a large proportion of system execution time (2,3). As network speeds continue to increase, pattern matching algorithms must be fast enough to keep up with network speeds; otherwise, NIDS can become a bottleneck in the network.

Since pattern matching play an important role in NIDS, a number of pattern matching algorithms can be found in the literature. These algorithms can be divided into two categories: hardware-based and software-based. Hardware-based algorithms⁴–⁶ utilize specific hardware components such as field programmable gate arrays (FPGAs), application-specific integrated circuits (ASICs), and content addressable memory (CAM). In contrast, software-based algorithms⁷–¹² utilize off-the-shelf processors, such as central processing units (CPUs) and graphics processing units (GPUs). In general, hardware-based algorithms offer high matching speeds, but are costly and inflexible to develop. In contrast, software-based algorithms are highly flexible and programmable. As the computing power of CPUs and GPUs continues to increase dramatically, it is expected that software-based algorithms can achieve matching speeds comparable to those of hardware-based algorithms. Thus, in this paper, we focus on software-based algorithms.
The rest of this paper is organized as follows. In Section 2, we summarize the related work in the literature. Our proposed method is described in detail in Section 3. Experimental results are presented and discussed in Section 4. Finally, Section 5 concludes the paper.

2. Related Work

2.1 Pattern Matching Algorithms with CPU/GPU Cooperation

The computing power offered by GPUs has increased rapidly in the last decade. Compared to CPUs, GPUs provide higher parallel computing capability, and have been used in many research areas. In our previous work\(^9\), we proposed a hybrid CPU/GPU pattern-matching algorithm (HPMA) that distributes workload between a CPU and GPU. Since the method proposed in this paper is based on the HPMA, we briefly review the HPMA here. In the HPMA, the pattern matching task is divided into two parts: pre-filtering and full pattern matching. Every incoming packet is initially processed by a pre-filtering algorithm, which was designed to be able to identify non-malicious packets using the CPU. Since the purpose of the pre-filtering algorithm is not to perform pattern matching, which is much more complicated, the pre-filtering algorithm requires very small memory size (about 20 KB), and provides fast processing speeds. Packets that are not non-malicious are called suspicious packets, since these packets may (but not necessarily) contain patterns. Suspicious packets are transferred to the GPU for full pattern matching.

2.2 Traditional CUDA Programming Style

Fig. 1 gives a pseudo code which illustrates the traditional programming style of CUDA. The host (i.e., the CPU and its memory) first allocates memory space in the device (i.e., the GPU and its memory). Then, data are transferred from the host to the device. The computing task is executed by launching a kernel. Once the GPU completes the task, results are transferred back to the host. Finally, memory space allocated in both the host and the device should be released. This pseudo code shows us the abstract structure of a CUDA program. Because the device memory space is limited, data may be divided into batches, and processed batch by batch by the GPU. If so, the CPU needs to launch a kernel for every batch of data.

Allocate memory space for data in host;
Allocate memory space in device;
while (there are more data to process)
{
    Initialize input data in host;
    Copy input data to device;
    Launch kernel on GPU;
    Copy results back to host;
}
Release allocated memory space in host and device;

Fig. 1. Traditional CUDA programming style.

3. Pattern Matching Using CUDA Persistent Threads

The traditional CUDA programming style mentioned previously is simple and can fulfill the requirements of most applications. However, if the amount of data to be processed by the GPU is large, and the application looks for the best performance in speed, the cost of launching kernels may become the performance bottleneck. Due to the fast advancement in network technology, it is necessary to minimize possible overhead in pattern matching for providing line speed processing. In this paper, we address this issue with CUDA persistent threads.

Fig. 2 shows the CUDA programming style using persistent threads. It is obvious that this programming style is more complicated than the traditional programming style. First, memory spaces required for the host and device are allocated as in the traditional programming style. Note that the host allocates extra memory space for flags, which are used to synchronize the host and the device. Thus, the allocated memory space should be mapped into the CUDA address space. In the pseudo code, two flags, dataReadyFlag and resultsReadyFlag, are used to indicate if the data to be transferred to the device are ready, and the results generated by the device are ready to be transferred back to the host, respectively. The value of dataReadyFlag is set to false before launching the kernel, so that the kernel does not start processing until the host transfers all data to the device. After launching the kernel, a while loop is used to process all data batch by batch. In the while loop, the first statement sets the value of resultsReadyFlag to false. The purpose of resultsReadyFlag is to inform the host that the device finishes its task, and the results are ready to be transferred back to the host. Then, the host initializes all
Perform the computing task;
if (this is the master block/thread) {
    while (there are no blocks still running)
        dataReadyFlag = false;
    resultsReadyFlag = true;
}  

The second part of the pseudo code explains how the device operates. Since the kernel is launched only once, a while loop is used to process all batches of data transferred from the host. Recall that the host sets dataReadyFlag to true when it has transferred a batch of data to the device. A while loop is used to wait for the data. Then, the device can perform its task, and writes results to the memory space allocated by the host. Since all threads are executed in parallel in the device, we must wait for all threads to finish before notifying the host to transfer the results back. Here, we choose the thread with the smallest index (i.e., threadIdx.x+blockDim.x*blockIdx.x = 0) as the master thread. Once there are no blocks still running, the master thread is the only thread to set the flags. Clearly, dataReadyFlag is set to false for waiting for the next batch of data, and resultsReadyFlag is set to true so that the host can transfer the results back, as mentioned previously.

### 4. Experimental Results and Discussion

Table 1 shows the hardware configuration used in our experiments. The simulation parameters are listed in Table 2. We used three concurrent processes to execute the pre-filtering algorithm. The operating system was 64-bit Ubuntu 20.04. The pattern set from Snort was used for performance evaluation. The number of patterns is 1,288. According to the required intrusive packet percentage, a certain number of packets were randomly chosen as intrusive packets by inserting a pattern randomly chosen from the pattern set at a random position in a packet payload.

A comparison of the original HPMA (denoted as HPMA) and HPMA with persistent threads (denoted as HPMA-PT) throughputs for different intrusive packet

---

### Table 1. Hardware configuration for the experiments.

<table>
<thead>
<tr>
<th>Component</th>
<th>Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Main board</td>
<td>SuperMicro/X10DRG-Q</td>
</tr>
<tr>
<td>CPU</td>
<td>Brand/Model: Intel/Xeon E5-2620 v4</td>
</tr>
<tr>
<td>Frequency</td>
<td>2.1 GHz</td>
</tr>
<tr>
<td>Main memory</td>
<td>Type: DDR4</td>
</tr>
<tr>
<td>Size</td>
<td>32 GB</td>
</tr>
<tr>
<td>Frequency</td>
<td>2,400 MHz</td>
</tr>
<tr>
<td>GPU</td>
<td>Brand/Model: EVGA/NVIDIA GTX 1080</td>
</tr>
<tr>
<td>Number of cores</td>
<td>2,560</td>
</tr>
<tr>
<td>Memory</td>
<td>8 GB GDDR5</td>
</tr>
<tr>
<td>NIC</td>
<td>Brand/Model: Intel/XL710 40 GbE QSFP+</td>
</tr>
</tbody>
</table>

### Table 2. Simulation parameters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Packet length</td>
<td>1,500 bytes</td>
</tr>
<tr>
<td>Blocks per grid</td>
<td>15</td>
</tr>
<tr>
<td>Threads per block</td>
<td>512</td>
</tr>
<tr>
<td>Concurrent processes</td>
<td>3</td>
</tr>
</tbody>
</table>
Fig. 3. Throughput value plotted against intrusive packet percentage.

percentages are presented in Fig. 3. We can see that HPMA-PT significantly outperforms HPMA, achieving 10.13% to 100.71% higher throughputs. HPMA throughput decreases as intrusive packet percentage increases. This is because when the intrusive packet percentage increases, more packets are sent to the GPU for full pattern matching. The time spent on the GPU is an additional cost since all packets are first inspected by the pre-filtering algorithm. In contrast, HPMA-PT throughput shows a different trend for intrusive packet percentages between 0% and 80%. This is because with persistent threads, the processing speed of the GPU is much higher than without persistent threads. Recall that the pre-filtering algorithm transfers packets to the GPU batch by batch. Suppose that the GPU is fast enough to inspect one batch of packets and transfers the results back to the CPU before the next batch is ready. From the perspective of the pre-filtering algorithm, the task of processing a packet is complete whether the packet should be sent to the GPU or not. In other words, the processing speed of the pre-filtering algorithm is equal to the processing speed of the whole system. As the percentage of intrusive packets increases, more packets are determined as malicious packets by the pre-filtering algorithm and sent to the GPU for full pattern matching. For a malicious packet, the pre-filtering algorithm generally does not need to examine the entire payload. Therefore, the higher the intrusive packet percentage, the higher the throughput. If the intrusive packet percentage is so high that the GPU cannot process one batch of packets in time, the pre-filtering algorithm will stop once the next batch is ready. This explains why HPMA-PT’s throughput starts to drop when 90% or 100% of packets are intrusive.

5. Conclusion

In this paper, we proposed a method to accelerating the matching speed of HPMA with persistent threads. With persistent threads, the overhead caused by launching kernels can be reduced significantly. Experimental results show that our proposed method can achieve up to 100.71% faster matching speed than the HPMA.

Acknowledgment

This work was supported in part by the High Speed Intelligent Communication (HSIC) Research Center of Chang Gung University, Taiwan, and by grants from the Ministry of Science and Technology of Taiwan (MOST 108-2221-E-182-010 and 110-2221-E-182-010) and Chang Gung Memorial Hospital (BMRP 942).

References


