Peningkatan Kinerja Modul Pencocokan Pola dalam Sistem Deteksi Intrusi Snort Menggunakan GPU

*Note: Sub-titles are not captured in Xplore and should not be used

Afrizal Fikri
School of Electrical Engineering and Informatics
Institut Teknologi Bandung
Bandung, Indonesia
afrizalf96@gmail.com

Achmad Imam Kistijantoro
School of Electrical Engineering and Informatics
Institut Teknologi Bandung
Bandung, Indonesia
imam@informatika.org

***Abstract*—The advancement of technology has been giving contributions to the rapid growth of the use of digital data. In this digital era, lots of physical data have been transformed into the digital ones. Among those, a lot of confidential data also take place. Thus, security assurance become important as well. One of the entry point of those malicious usages come from server side. Ensuring authority on server utilizing network intrusion detection system (NIDS) could take enormous resource and time. One of the important part of NIDS is string matching part inside the analyzer. Based on this rationale, this paper aims to improve matching speed of detection in order to maximize overall system throughput using GPU instead of CPU. Based on the experiment and testing, the proposed solution has a significantly better run time compared to the state of the art linear solution while maintaining the accuracy.**

***Index Terms*—pattern matching; intrusion detection; GPU; parallel computation; CUDA**

I. Introduction

This document is a model and instructions for LaTeX. Please observe the conference page limits.

II. String Matching

In general, there are two kinds of string matching: single pattern string matching and multi pattern string matching. We will focus into multi pattern string matching. There are several famous algorithms for multi pattern string matching: Aho-Corasick, Commentz-Walter, and Wu-Manber algorithm. This paper will try to implement variation of Aho-Corasick algorithm.

A. Aho-Corasick Algorithm

Aho-Corasick algorithm is kind of string matching algorithm works by matching one string to a dictionary. This dictionary contains all pattern to be matched in form state machine.

In this algorithm, matching operation is performed by traversing state machine by characters in input string. If any of the final state reached, then input string matched one of the patterns. Matching continued until input string is terminated.

Identify applicable funding agency here. If none, delete this.

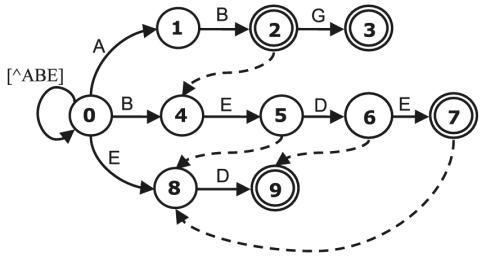

Fig. 1. Example state machine of Aho-Corasick dictionary.

Aho-Corasick use failure function to perform backtrack after stop at one of the final states. Failure function will redirect this final state to longest pattern prefix which match suffix of the currently matched pattern. Failure function reduce the need of repeating another same sequence which just matched by. By this method, all of containing patterns can be matched by one pass and reducing complexity from $O(mn)$ to $O(n)$.

In this algorithm, memory lookup become bottleneck. Every traversal operation will fetch one entry from table. Since memory accesses are more expensive than computations, this algorithm is memory bound.

B. Data Parallel Aho-Corasick

In order to extend Aho-Corasick into multithreading, we need to divide the load of matching for each threads. One of the method is by partitioning input into several chunks and then each threads will match this sequence into dictionary. But as we encounter a pattern span multiple chunks at once, it is not recognized on either threads. This problem known as boundary detection problem. So, we need to extend matching by length of the longest pattern. Memory lookup thus increase to $O((n/s+m)*s) = O(n+ms)$ for s is amount of chunks.

C. Parallel Failureless Aho-Corasick

Another method to adapt multithreading Aho-Corasick is to spread threads to all characters byte. Each threads will

Fig. 2. Pattern **AB** not recognized by both thread 3 and 4

matching any pattern begin with each coresponding character and terminate if any of final state or no valid transition occurs [**?**]. The consequence is eavery character will match only one pattern at most. Thus, no need to make any failure function. And also boundary detection problem will not occur by using this approach.

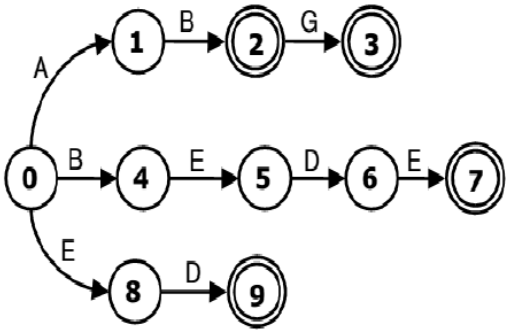

Fig. 3. State machine without failure function

III. Related Works

A. Utilization of Double Buffering Scheme, Texture Memory and Pinned Memory

In this paper, GPU-based matching implementation had been proposed using Aho-Corasick algorithm. Beside utilization of GPU multithreading, there are few adjustment to import Aho-Corasick to GPGPU. State machine built using transition table on 2D table on texture memory. Based on experiment, this implementation improving performance by 19% [**?**]. For the memory transfer scheme, double buffering scheme was proposed. Packets will be batched in one buffer. Whenever the buffer gets full, all packets are transferred to the GPU in one operation. Transfer from device to host also done in same way. All result will be batched in the second buffer and get transferred when gets full. Another optimization trick used is using pinned memory. Both of the buffer in host will be pinned to reduce swappiness. And pinned memory also enable DMA transfer and reducing computation needs and latency. This design can improve system performance until 3.2 times higher than implementation using multithreading on CPU.

B. Maximizing GPU Utilization by Pipelining

By profiling, it is concluded that bottleneck occured on 3 components: packet acquisition, multi pattern string matching, and option rule matching by regular expression. Moreover, inside string matching component alone,

C. Failureless Aho-Corasick by Maximizing Shared Memory Usage

Salah-satu algoritma yang sering digunakan dalam deteksi pola adalah Aho-Corasick yang bersifat *multi-pattern string matching*. Dalam penelitian yang dilakukan [**?**], dilakukan optimasi pada implementasi algoritma Aho-Corasick pada GPU. Implementasi memanfaatkan fitur-fitur pada GPU seperti *memory coalescing*, penghematan transaksi pada *global memory*, dan akses *shared memory* yang bebas *bank-conflict*.

Desain akan menggunakan pendekatan algoritma AC tanpa *failure function* yaitu PFAC (*Parallel Failureless Aho-Corasick*). Pendekatan ini dapat memaksimalkan penggunaan *thread* pada GPU tanpa menyebabkan *overlapping*. Transaksi ke *global memory* dapat dikurangi dengan memuat transisi *state* kosong ke *shared memory* dan penggunaan *texture memory* yang memanfaatkan *cache*.

Hasil pengujian dengan CPU Intel Core i7-950 dan GPU NVIDIA GTX 580 menunjukkan peningkatan yang signifikan. Optimasi dengan PFAC pada *multithread* CPU mencapai 7 kali lipat dari AC. Sedangkan PFAC pada GPU dapat mencapai 10 kali lipat daripada AC pada CPU. Masukan didapatkan dari dataset *traffic* kompetisi DEFCON.

IV. Proposed Solution

A. Matching Algorithm

B. State Machine Implementation

C. Thread *Allocation*

D. GPU Optimation

V. Implementation on Snort

A. MPSE Interface

B. Modules

VI. Evaluation

A. Environment

B. Performance Evaluation

VII. Conclusion

VIII. Acknowledgment