

# Accelerating Regular Expression Matching Using Hierarchical Parallel Machines on GPU

Cheng-Hung Lin\*, Chen-Hsiung Liu\*\*, and Shih-Chieh Chang\*\*

\* National Taiwan Normal University, Taipei, Taiwan

\*\* National Tsing Hua University, Hsinchu, Taiwan

**Abstract**—Due to the conciseness and flexibility, regular expressions have been widely adopted in Network Intrusion Detection Systems to represent network attack patterns. However, the expressive power of regular expressions accompanies the intensive computation and memory consumption which leads to severe performance degradation. Recently, graphics processing units have been adopted to accelerate exact string pattern due to their cost-effective and enormous power for massive data parallel computing. Nevertheless, so far as the authors are aware, no previous work can deal with several complex regular expressions which have been commonly used in current NIDSs and been proven to have the problem of state explosion.

In order to accelerate regular expression matching and resolve the problem of state explosion, we propose a GPU-based approach which applies hierarchical parallel machines to fast recognize suspicious packets which have regular expression patterns. The experimental results show that the proposed machine achieves up to 117 Gbps and 81 Gbps in processing simple and complex regular expressions, respectively. The experimental results demonstrate that the proposed parallel approach not only resolves the problem of state explosion, but also achieves much more acceleration on both simple and complex regular expressions than other GPU approaches.

**Keywords**—regular expression, pattern matching, graphics processing units

## I. INTRODUCTION

Network Intrusion Detection Systems (NIDS) have been widely employed to protect computer systems from network attacks by matching input streams against thousands of predefined attack patterns. Regular expression has been adopted in many NIDS systems, such as Snort [1], Bro [2], and Linux L7-filter [3], to represent certain attack patterns because regular expression can provide more concise and flexible expressions than exact string expressions

To accelerate regular expression matching, many hardware approaches are being proposed that can be classified into logic-based [4][5][6][7] and memory-based approaches [8][9][10][11][12]. The logic-based approaches which are mainly implemented on the Field-Programmable Gate Array (FPGA), map each regular expression into circuit modules in FPGA. Logic-based approaches are known to be efficient in processing regular expression patterns [4][5] because multiple logic modules can perform their operations simultaneously.

On the other hand, memory-based approaches compile attack patterns into finite state machines and employ commodity memories to store the corresponding state transition tables. Since state transition tables can be easily updated on commodity memories, memory-based

approaches are flexible to accommodate new attack patterns. Memory-based approaches have been known to suffer the memory explosion problem for certain types of complex regular expressions. To resolve the memory problem, the rewriting technique [13] converts a regular expression to a new regular expression with smaller DFA. Another research D<sup>2</sup>FA [20] proposes to reduce the number of state transitions for a regular expression by introducing a new transition called a “default transition.”

Recently, several works [21][22][23][24] have attempted to use Graphic Processor Units (GPU) to accelerate exact and regular expression pattern matching because GPUs provides tremendous computational ability and very high memory bandwidth to process massive input streams and attack patterns. A modified Wu-Manber algorithm [21] and a modified suffix tree algorithm [22] are implemented on GPU to accelerate exact string matching while a traditional DFA approach [23] and a new state machine XFA [24] are proposed to accelerate regular expression matching on GPU. However, all of these approaches do not consider the complex regular expressions which can incur state explosion.

In this paper, we first explore the parallelism of regular expression matching and discuss the problem of memory explosion. And then, we propose a GPU-based approach which applies hierarchical parallel machines to accelerate regular expression matching and resolve the problem of state explosion. The experimental results show that the proposed parallel algorithm achieves up to 100 Gbps and 81 Gbps in processing simple and complex regular expressions, respectively. The results show that the proposed algorithm has significant improvement on performance than other GPU approaches.

## II. COMPLEX REGULAR EXPRESSION PATTERNS

As mentioned above, in this paper, we attempt to reduce all types of complex regular expression into two specific types of complex regular expressions. In this section, we first review these two types of complex regular expressions and explore the reasons that merging such complex regular expressions may lead to large DFAs.

The first type of complex regular expressions is an expression having multiple string sub-patterns divided by the star closure, “\*”. For example, the pattern in Linux L7-filter, “\*.membername.\*session.\*player”, has three string sub-patterns, “membername”, “session”, and “player”. We illustrate the reason of the memory explosion using the following example. Consider to compile two regular expressions, “\*.RETA.\*PASS” and “\*.CWD.\*ROOT”. Figure 1 illustrates the composite DFA which attempts to match these two regular expression patterns where partial edges have been

The second type of complex regular expression is the pattern that has constraint repetitions, such as the pattern for detecting FTP STAT overflow attempt, “\*STAT[^\n]{100}”. The constraint repetitions, “[^\n]{100}”, represents one hundred of non-line-feed characters. Merging such a complex regular expression may also lead to memory explosion. For example, using Flex [15] to compile the single pattern, “.\*STAT[^\x0a]{10}” results in a DFA containing 137 states. If we merge this regular expression pattern with the other two patterns “.\*USER [^\x0a]{10}” and “.\*PASS[^\x0a]{10}”, the number of states increases sharply to 2,498.

### III. HIERARCHICAL PARALLEL MACHINES

For example, in Section 2, the first type of complex regular expressions can be partitioned into two parts, exact string patterns and meta-characters. We can find that the matching of these two types of complex regular expressions can be translated to find the appearance order between exact string patterns. For example, matching the regular expression “.\*RETA.\*PASS” is equivalent to check whether there is a pattern “RETA” appearing before “PASS” with zero or more instances of any character between “RETA” and “PASS”. Similarly, matching “.\*uid.{0,10}gid” is equivalent to check whether “uid” appears before “gid” with at least 0 but no more than 10 instances of any character between “uid” and “gid”.

```

graph LR
    A[Input streams] --> B[Master Machine]
    B --> C[Inter-result]
    C --> D[Slave Machine]
    D --> E[Match vector]
  
```

### 3.1 Master Machine

The PFAC algorithm is proposed to accelerate exact string matching by parallelizing the Aho-Corasick (AC) algorithm [14] on GPUs. Using PFAC algorithm has two steps. The first step is to compile multiple string patterns into a PFAC state machine. For example, Figure 4 shows the PFAC state machine for the four patterns, “RETA”, “ROOT”, “PASS”, and “CWD”, where solid lines represent valid transitions. Except for valid transitions, all other invalid transitions leading to trap states are omitted.

In the second step, the PFAC algorithm assigns an individual thread for each byte of an input stream to inspect any pattern starting at the thread's starting position by traversing the same PFAC state machine. As shown in Figure 5, the first thread inspects the input stream from the first character 'A' while the second thread starts from the second character 'R' and the third thread starts from the third character 'E'. A pictorial demonstration for all other threads is shown in Figure 5.

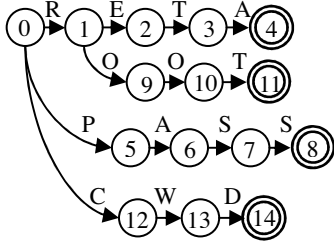


Figure 4. PFAC state machine

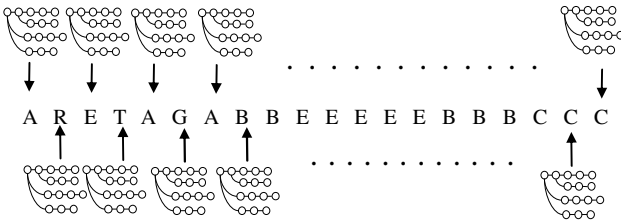


Figure 5. Allocate each byte of the input stream a thread to traverse the PFAC state machine.

During the traversal of a PFAC state machine, if a thread finds a match, the thread reports which pattern is matched, and the starting location corresponding to the thread. When a thread encounters an invalid transition in a PFAC state machine, the thread terminates its computation. For example in Figure 5, traversing the PFAC machine in Figure 4, only the second thread finds a match and reports two information including the match of "RETA" and the starting location of the thread, the second byte. All other threads terminate when they are led to trap states. For example, the first input character taken by the first thread is "A" which has no valid transition for the initial state. Therefore, the first thread terminates immediately after reading the first character. The idea of applying multiple threads to traverse the same PFAC state machine has important implications for efficiency. First, each thread of the PFAC algorithm is only responsible for matching any pattern located at the thread's starting location. Second, although the PFAC algorithm applies huge number of threads to perform pattern matching in parallel, most of threads have a high probability of terminating early. Therefore, the PFAC algorithm can achieve significant improvements on performance than the traditional AC algorithm.

By extending the PFAC algorithm, we can easily compile the regular expressions containing character classes with repetitions into a PFAC state machine. Because each thread of the PFAC algorithm is only responsible for identifying the pattern located at the thread starting position, the additional states caused by the overlapped matching are totally removed. For example, Figure 6 shows the state machine for matching the three regular expression patterns, `"*STAT [^\n]{10}"`, `"*USER [^\n]{10}"`, and `"*PASS [^\n]{10}"`.

`[^\n]{10}"`, `"*USER [^\n]{10}"`, and `"*PASS [^\n]{10}"`. Compared to the traditional DFA state machine as shown in Figure 2, the number of states is significantly reduced from exponential to linear size with respect to the length of constraint repetitions.

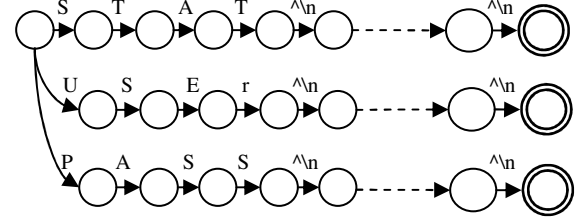


Figure 6. A PFAC state machine for the three complex regular expressions, `"*STAT [^\n]{10}"`, `"*USER [^\n]{10}"`, and `"*PASS [^\n]{10}"`.

### 3.2 Slave Machine

Since the master machine can resolve the second type of regular expressions, we now describe how to design the slave machine to consider the first type of regular expressions. Note that the outputs of the master machines include starting locations of master threads and their corresponding matched exact patterns which are translated to encoded symbols. Therefore, the inputs to the slave machine are a set of encoded symbols and their corresponding starting locations. The purpose of the slave machine is to determine the order relationship between encoded symbols using their starting locations.

We demonstrate the construction of the slave machine using the same example of matching two regular expressions `"*RETA.*PASS"` and `"*CWD.*ROOT"`. As shown in Figure 4, the master machine is used to match the four sub-patterns, "RETA", "PASS", "CWD", and "ROOT" whose encoded symbols are labeled as  $\alpha$ ,  $\gamma$ ,  $\delta$ , and  $\beta$ , respectively.

The regular expressions `"*RETA.*PASS"` and `"*CWD.*ROOT"` denote that "RETA" must appear before "PASS" and "CWD" must appear before "ROOT". In other words, a slave machine checks whether  $\alpha$  appears before  $\gamma$  and  $\delta$  appears before  $\beta$ . The construction of a slave machine consists of three steps. In the first step, we build the initial machine for each order relation of encoded symbols. The second step is to add a self-loop transition to each internal node. Figure 7 shows the slave machine of `" $\alpha\gamma$ "` and `" $\delta\beta$ "`. We add self-loop transitions labeled as `" $\wedge\gamma$ "` and `" $\wedge\beta$ "` to state 1 and 3, respectively. These transitions mean that if the next input is not  $\gamma$  and  $\beta$ , the machine stays in state 1 and state 3, respectively. In other words, the self-loop transition represents the meta-character `"."` in the original regular expressions.

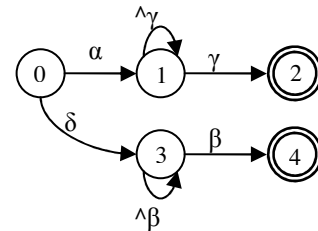
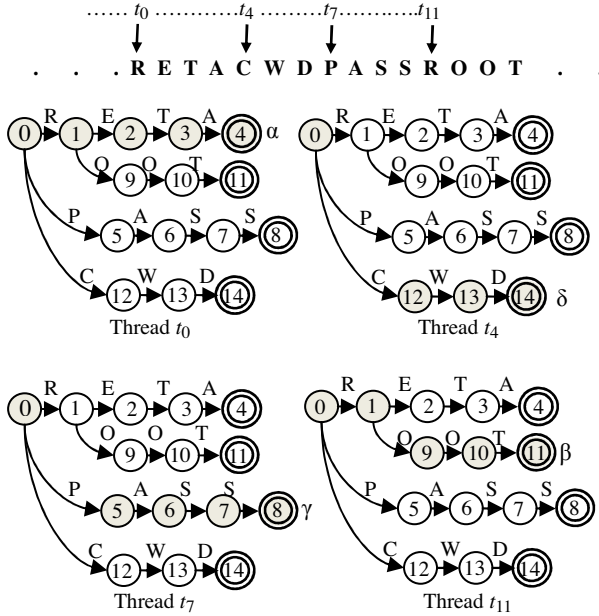


Figure 7. Slave machine of `" $\alpha\gamma$ "` and `" $\delta\beta$ "`.

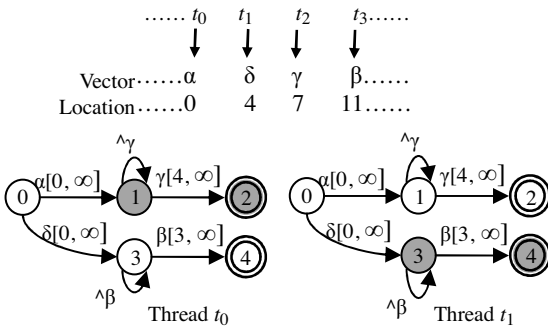
### 3.3 Complete Example

We now use an example to summarize the operation of the master and slave machines. Consider an input stream “RETACWDPASSROOT”. In Figure 8, threads  $t_0$ ,  $t_4$ ,  $t_7$ , and  $t_{11}$  are allocated to character ‘R’, ‘C’, ‘P’, and ‘R’, respectively. After taking the inputs “RETA”, thread  $t_0$  reaches state 4 and outputs  $\alpha$ , which indicates the pattern “RETA” is matched. Because there is no valid transition for “R” in state 4, thread  $t_0$  terminates at state 4. Similarly, thread  $t_4$  reaches final state 14 and outputs  $\delta$ . simultaneously, threads  $t_7$  and  $t_{11}$  matches the patterns “PASS” and “ROOT” and output  $\gamma$  and  $\beta$ , respectively. Finally, the master machine outputs the sequence of encoded symbols and their starting locations,  $(\alpha, 0)$ ,  $(\delta, 4)$ ,  $(\gamma, 7)$ ,  $(\beta, 11)$ .



**Figure 8. Example of master machine where the patterns “RETA”, “CWD”, “PASS”, and “ROOT” are identified by the threads  $t_0$ ,  $t_4$ ,  $t_7$ , and  $t_{11}$ , respectively.**

Similar to the master machine, we also apply multiple threads on the slave machine to identify order relationships of the output sequence. As shown in Figure 9, thread  $t_0$  allocated to  $\alpha$  matches the sequence “ $\alpha\gamma$ ” which indicates the regular expression “ $.*RETA.*PASS$ ” is matched while thread  $t_1$  allocated to  $\delta$  matches the sequence “ $\delta\beta$ ” which indicates the regular expression “ $.*CWD.*ROOT$ ” is matched.



**Figure 9. Using slave machine to identify the output sequence “ $\alpha\delta\gamma\beta$ ”.**

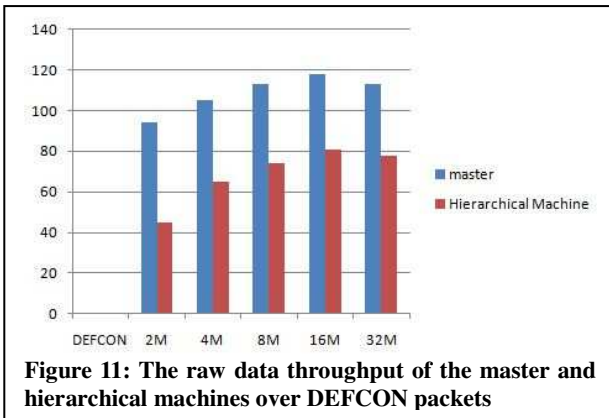
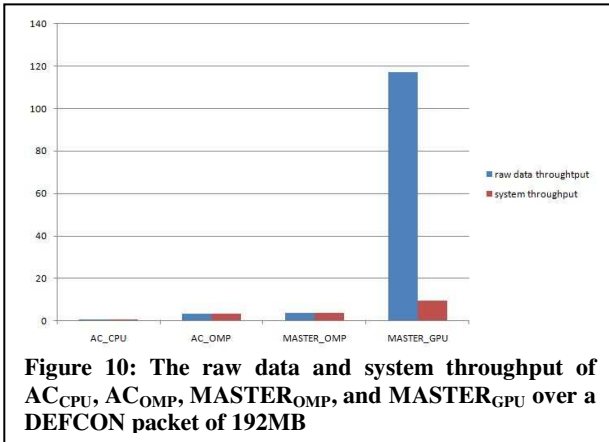
### IV. EXPERIMENTAL RESULTS

We have implemented the proposed algorithm on Nvidia® GPUs which offers a parallel programming model via the CUDA™ (Compute Unified Device Architecture) [25]. The experimental configurations are divided into *host* and *device*. The host is equipped with an Intel® Core™ i7-950 running the Linux X86\_64 operating system with 12GB DDR3 memory on an ASUS P6T-SE motherboard while the device is equipped with an Nvidia® GeForce® GTX480 GPU in the same Core™ i7 system with Nvidia driver version 260.19.29 and the CUDA 3.2 version. The network attack patterns consists of 990 simple regular expressions and 88 complex regular expressions extracted from the Bro V1.5.1 rule sets published in 2010. The regular expression matching engine is tested using DEFCON [28] packets. The DEFCON packets which contain large amounts of real attack patterns are widely used to test commercial NIDS systems.

We extend our previous work [26], the PFAC library [27] to construct the master and slave machines. The master machine alone can perform simple regular expression matching. The simple regular expressions contain exact string patterns and the second type of complex regular expressions, such as “ $.*STAT[^n]{10}$ ”. In order to compare the performance of the master machine, we implement three CPU versions and one GPU version where  $AC_{CPU}$  denotes the implementation of the AC algorithm on the Core™ i7 using single thread,  $AC_{OMP}$  denotes the implementation of the AC algorithm on the Core™ i7 with OpenMP [29] library,  $MASTER_{CPU}$  denotes the implementation of the master machine on the Core™ i7 with OpenMP library, and  $MASTER_{GPU}$  denotes the implementation of the master machine on the GTX480. Figure 10 shows the comparisons of the master machine and the other three CPU versions for processing a DEFCON packet of 192Mbytes. In Figure 10, the raw data throughput denotes the data throughput without considering the data transfer time via PCIe while the system throughput denotes the data throughput with considering the bandwidth of PCIe transmission. Figure 10 shows that  $MASTER_{GPU}$  achieves maximum raw data throughput of 117 Gbps while  $AC_{CPU}$ ,  $AC_{OMP}$ , and  $MASTER_{OMP}$  achieves 0.72, 3.22, and 3.43 Gbps. Compared to  $AC_{CPU}$ ,  $AC_{OMP}$ , and  $MASTER_{OMP}$ ,  $MASTER_{GPU}$  achieves 160x, 36x, and 34x times speedup, respectively. Moreover, because the PCIe interface only provides the effective bandwidth of 6~6.41 GB/s, the system throughput would be much smaller due to PCIe overheads. However, even considering the PCIe overheads, the  $MASTER_{GPU}$  still has 2~2.5x times improvements compared to  $AC_{OMP}$  and  $MASTER_{OMP}$ .

Furthermore, we evaluate the implementation of the hierarchical parallel machines which consists of two state machines running sequentially. Because none of other approaches adopt the same architecture as ours, we only provide the results of the hierarchical parallel machines. Figure 11 shows that the raw data throughput of the master and the hierarchical machines over DEFCON packets. The hierarchical machines achieve up to 81Gbps for processing 16MBytes DEFCON packets. Due to the

memory limitation of GTX480, the maximum size of the input stream can be processed at a time is 32Mbytes while the master machine alone can support up to 192Mbytes. We would like to mention that multiple GPUs would be a good solution to improve throughput in the future works.



## V. CONCLUSIONS

In this paper, we have explored the parallelism of complex regular expression matching and proposed a new architecture which consists of hierarchical parallel machines performed on GPU to accelerate regular expression matching and resolve the problem of memory explosion. The experimental results show that the proposed approach achieves a significant speedup both on processing simple and complex regular expressions.

## REFERENCES

- [1] M. Roesch. Snort- lightweight Intrusion Detection for networks. In Proceedings of LISA99, the 15th Systems Administration Conference, 1999.
- [2] Bro, <http://www.bro-ids.org/>
- [3] Linux L7-filter, <http://l7-filter.sourceforge.net/>
- [4] R. Sidhu and V. K. Prasanna, "Fast regular expression matching using FPGAs," in *Proc. 9th Ann. IEEE Symp. Field-Program. Custom Comput. Mach. (FCCM)*, 2001, pp. 227-238.
- [5] B.L. Hutchings, R. Franklin, and D. Carver, "Assisting Network Intrusion Detection with Reconfigurable Hardware," in *Proc.10th Ann. IEEE Symp. Field-Program. Custom Comput. Mach. (FCCM)*, 2002, pp. 111-120.
- [6] C. R. Clark and D. E. Schimmel, "Scalable Pattern Matching for High Speed Networks," in *Proc. 12th Ann. IEEE Symp. Field-Program. Custom Comput. Mach. (FCCM)*, 2004, pp. 249-257.
- [7] J. Moscola, J. Lockwood, R. P. Loui, and M. Pachos, "Implementation of a Content-Scanning Module for an Internet Firewall," in *Proc. 11th Ann. IEEE Symp. Field-Program. Custom Comput. Mach. (FCCM)*, 2003, pp. 31-38.
- [8] M. Aldwairi\*, T. Conte, and P. Franzon, "Configurable String Matching Hardware for Speeding up Intrusion Detection," in *ACM SIGARCH Computer Architecture News*, 2005, pp. 99-107.
- [9] S. Dharmapurikar and J. Lockwood, "Fast and Scalable Pattern Matching for Content Filtering," in *Proc. of Symp. Architectures Netw. Commun. Syst. (ANCS)*, 2005, pp. 183-192.
- [10] Y. H. Cho and W. H. Mangione-Smith, "A Pattern Matching Co-processor for Network Security," in *Proc. 42nd Des. Autom. Conf. (DAC)*, 2005, pp. 234-239.
- [11] L. Tan and T. Sherwood, "A high throughput string matching architecture for intrusion detection and prevention," in *proc. 32nd Ann. Int. Symp. on Comp. Architecture, (ISCA)*, 2005, pp. 112-122.
- [12] H. J. Jung, Z. K. Baker, and V. K. Prasanna, "Performance of FPGA Implementation of Bit-split Architecture for Intrusion Detection Systems," in *20th Int. Parallel and Distributed Processing Symp. (IPDPS)*, 2006.
- [13] F. Yu, Z. Chen, Y. Diao, T. V. Lakshman, and R. H. Katz, "Fast and Memory-Efficient Regular Expression Matching for Deep packet Inspection," in *Proc. ACM/IEEE Symp. Architectures Netw. Commun. Syst. (ANCS)*, 2006, pp. 93-102.
- [14] A. V. Aho and M. J. Corasick. Efficient String Matching: An Aid to Bibliographic Search. In *Communications of the ACM*, 18(6):333-340, 1975.
- [15] Flex, <http://flex.sourceforge.net/>
- [16] PCRE, <http://www.pcre.org/>
- [17] M. Aldwairi, T. Conte, and P. Franzon, "Configurable String Matching Hardware for Speeding up Intrusion Detection," in *Proc. ACM SIGARCH Computer Architecture News*, 33(1):99-107, 2005.
- [18] F. Yu, R. H. Katz, and T. V. Lakshman, "Gigabit Rate Packet Pattern-Matching Using TCAM," in *Proc. the 12th IEEE International Conference on Network Protocols (ICNP'04)*, 2004.
- [19] N. Tuck, T. Sherwood, B. Calder, and G. Varghese. "Deterministic Memory-Efficient String Matching Algorithms for Intrusion Detection," in *Proc. 23rd Conference of IEEE Communication Society (INFOCOMM)*, Mar, 2004.
- [20] S. Kumar, S. Dharmapurikar, F. Yu, P. Crowley, and J. Turner, "Algorithms to Accelerate Multiple Regular Expressions Matching for Deep Packet Inspection," in *ACM SIGCOMM Computer Communication Review*, ACM Press, vol.36, Issue. 4, Oct. 2006, pp. 339-350.
- [21] N. F. Huang, H. W. Hung, S. H. Lai, Y. M. Chu, and W. Y. Tsai, "A gpu-based multiple-pattern matching algorithm for network intrusion detection systems," in *Proc. 22nd International Conference on Advanced Information Networking and Applications (AINA)*, 2008, pp. 62-67.
- [22] M. C. Schatz and C. Trapnell, "Fast Exact String Matching on the GPU," Technical report.
- [23] G. Vasiladias, M. Polychronakis, S. Antonatos, E. P. Markatos and S. Ioannidis, "Regular Expression Matching on Graphics Hardware for Intrusion Detection," in *Proc. 12th International Symposium on Recent Advances in Intrusion Detection*, 2009.
- [24] R. Smith, N. Goyal, J. Ormont, K. Sankaralingam, C. Estan, "Evaluating GPUs for network packet signature matching," in *Proc. of the International Symposium on Performance Analysis of Systems and Software, ISPASS (2009)*.
- [25] CUDA, [http://www.nvidia.com.tw/object/cuda\\_home\\_tw.html](http://www.nvidia.com.tw/object/cuda_home_tw.html)
- [26] Cheng-Hung Lin, Sheng-Yu Tsai, Chen-Hsiung Liu, Shih-Chieh Chang, and Jyuo-Min Shyu, "Accelerating String Matching Using Multi-threaded Algorithm on GPU," in *Proc. IEEE GLOBAL COMMUNICATIONS CONFERENCE (GLOBECOM)*, 2010.
- [27] PFAC library, <http://code.google.com/p/pfac/>
- [28] DEFCON, <http://cctf.shmoo.com>
- [29] OpenMP, <http://openmp.org/wp/>