A STT-Partition-Based Parallel Algorithm for Pattern Matching on GPU and CPU

Xudong Liu[1*], Yanbing Liu[2], Jian Li[1], Jing Yu[2], Jianlong Tan[2]

[1] School of Computer, Beijing University of Posts and Telecommunications, Beijing, China.
[2] Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China.

* Corresponding author. Tel.: +8618811597602; email: liuxudongbupt@163.com

Manuscript submitted September 5, 2014; accepted November 11, 2014.
doi: 10.17706/ijcce.2015.v4.387

Abstract: Pattern matching is an important process in various applications such as information and network security, bioinformatics, image processing, etc. Aho-Corasick (AC) is one of the most representative algorithms for multiple pattern matching. As the data becomes extraordinarily large, GPUs have been adopted to accelerate pattern matching because of their great power for parallel computing. However, if the automata of AC algorithm contains more than hundreds of thousands of nodes, its State Transition Table (STT) takes up quite large storage space which is beyond GPU memory. In this paper, we present an improved AC algorithm named as STT-partition-based parallel AC (SPAC) to reduce the storage space for GPU by separating the original STT into two parts, one is kept in GPU and the other is stored in CPU. GPU is in charge for the major filtering task in the first step and then the relatively small-scale filtered results are further processed on CPU. Experiments are carried out on three different datasets and results show that our method reduces the storage space by 45%~50% compared with state-of-the-art algorithms with comparable matching speed.

Key words: Aho-Corasick, GPGPU, parallel pattern matching, STT-partition.

1. Introduction

Nowadays, Network Intrusion Detection System (NIDS) has been widely used in Internet to keep the security of network operations. Among the representative network-based intrusion detection techniques, pattern matching is most commonly used for its excellent performance in both matching speed and matching accuracy.

Many pattern matching methods have been proposed in the past, most of which derive from the idea of Aho-Corasick (AC) [1], which is widely used in various pattern matching applications such as network intrusion detection, bio-sequence analysis, and image processing. Many popular NIDS and anti-virus systems, such as Snort [2] and ClamAV [3], apply AC algorithm in their multi-pattern matching engines since its worst performance is deterministic, linear to the length of the input stream and independent of the rule set size. Therefore it is impossible for an attacker to construct a worst-case traffic that can slow down the NIDS and let malicious traffic escape the inspection. With rule sets fast-growing, implementing AC algorithm with limited memory without sacrificing performance becomes a major bottleneck in NIDS design.

In the past few years, many approaches have been proposed to accelerate pattern matching based on hardware improvement. These approaches can be classified into two categories, logic architecture-based approaches [4]-[6] and memory architecture-based approaches [7]-[9]. In the first category, attacking patterns are synthesized into logic circuits that are typically implemented on field-programmable gate array (FPGA) to match multiple patterns in parallel. In the second category, memory architectures compile attack patterns into an automata and perform pattern matching by traversing the corresponding state transition table that is stored in a commodity memory.

Recently, GPUs have been adopted to accelerate pattern matching because of their enormous power for massive data parallel computing. User friendly programming environment has been developed recently such as CUDA [10] and OpenCL [11]. Using such environment, one can have more direct control over the cores of GPU and the memory hierarchy. Taking advantages of these tools, lots of innovative improvements for large-scale pattern matching have been made and many more are still to come.

The rest of the paper is organized as follows. Section 2 reviews the related work by analyzing representative works in pattern matching using AC algorithm on GPU. We pointed out some deficiencies existed in these algorithms. In Section 3, we propose a performance-driven SPAC algorithm on GPU and CPU, which makes full use of the CPU and GPU hardware features. In Section 4, we describe the implementation details of SPAC algorithm. Experiments and results are detailed and analyzed in Section 5. We conclude the paper in Section 6.

2. Related Work

In this Section, we first introduce the Aho-Corasick (AC) [1] algorithm and then describe two representative improved methods on GPU for pattern matching. Their contributions and problems are analyzed in the end.

2.1. Aho-Corasick (AC) Algorithm

The main idea of AC algorithm is to use failure transitions to backtrack a automata to recognize patterns starting at any location of an input stream. Given a current state and an input character, the AC automata first checks the information of the character, the AC automata first checks the information of the next state from the valid transition table to determine whether there is a valid transition for the input character. If not, the state jumps to the corresponding failure state according to the failure transition table. Then, the automata process the same input character until the character end in a valid transition. Fig. 1 shows an automata of the AC algorithm for matching patterns {era, her, heroic, oil}. The purple-circled nodes and white-circled nodes respectively represent the final states of the matched patterns and the middle states. The solid lines and the dotted lines respectively represent the valid transitions and the failure transitions. Any node without failure transitions has a failure transition pointing to the root node. As shown in Fig. 1, states 3, 6, 9 and 12 are the final states of the patterns "era", "her", "heroic", "oil". The pattern "her" is a prefix of the pattern "heroic", so there are several "shared node" like state 4, 5, 6, in the AC automata.

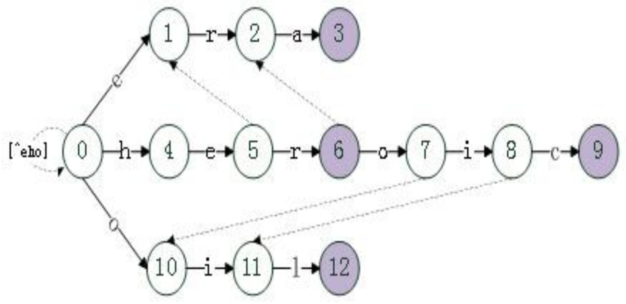


Fig. 1. AC automata of the patterns {era, her, heroic, oil}.

For example, consider the case that we want to match an input stream containing "hera" via the AC automata in Fig. 1. The AC automata starts from state 0 with the following travel path 0—>4—>5—>6, where the state 6 is the final state of the pattern "her". Because the state 6 has no valid transition for the next character "a", the AC automata first takes a failure transition to jump to the state 2 to process the previously failed character "a". As there exists a valid transition for "a" from the state 2 to the state 3, the AC automata reaches state 3 that is the final state of pattern "era." Finally, this AC automata matches the patterns "her" and "era" at the positions of "h", "e" respectively. In summary, the AC algorithm matches all patterns in O(n) time for processing an input stream of length n, regardless of the patterns.

2.2. Data Parallel AC Algorithm Implemented on GPUs

One kind of the data parallel approaches [12]-[15] aim to parallelize the AC algorithm, which is referred to as the data-parallel AC (DPAC) approach. The key point of DPAC approach is to divide an input stream into multiple chunks and allocates each chunk an individual thread to perform the AC algorithm separately. For example, Fig. 2 shows the general framework of the data-parallel AC (DPAC) algorithm, where the input stream is divided into two chunks and two threads are assigned to the two chunks to traverse the same AC automata. However, the obvious problem of DPAC approach is that it cannot detect a pattern occurring in the boundary of adjacent chunks. This problem is called the "boundary detection" problem. For example, in Fig. 2 the pattern "her" occurs in the boundary of chunk 1 and chunk 2 and cannot be identified by either thread 1 or thread 2. As DPAC focuses on the partition of the input data for parallel computing, it doesn't change the original AC scheme. Due to the nature of AC algorithm, single direction scanning for using the prefix information of the scanned data is not suitable for parallel execution.

Another kind of the data parallel approaches [16] for parallelizing the AC algorithm is referred to as the parallel failureless-AC (PFAC) algorithm. In this approach, the failure transitions are removed from all states. As shown in Fig. 3, for each substring starting from each byte of the input stream, the PFAC algorithm allocates an individual thread to identify the pattern beginning with the first input byte. If the first input byte in one thread doesn't match any pattern's beginning state, that thread will be terminated immediately without taking failure transitions to backtrack the automata.

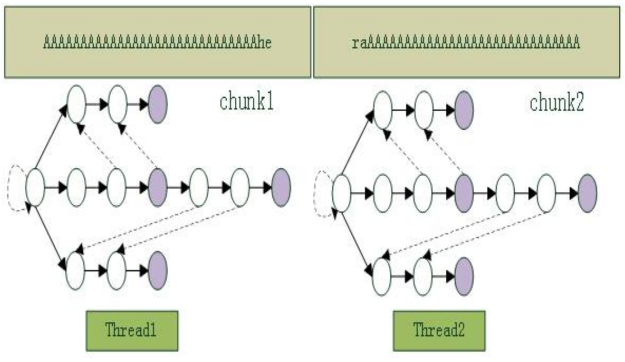

AAAAAAAAAAAAAAAAAAAAAAAAAAAAAhe
raAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
chunk1
chunk2
Thread1
Thread2

Fig. 2. The data-parallel AC (DPAC) algorithm.

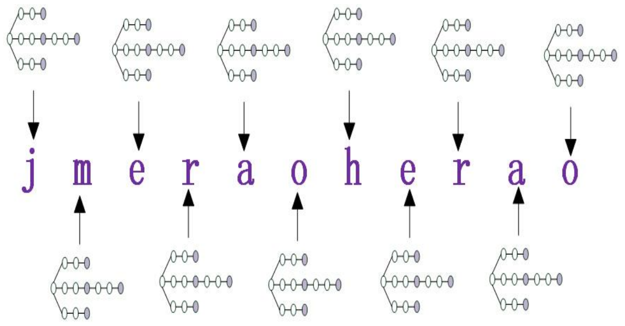


Fig. 3. The data-parallel Failureless AC (PFAC) algorithm.

Though all the approaches mentioned above are designed on GPU taking the advantage of parallel computing, they have all ignored the important features of the length distribution of the pattern strings as well as the size of the AC automata:

1) The pattern string length distribution: We have computed the length distribution of the pattern strings in the rule sets of both Snort and ClamAv. The results indicate that the length distribution is quite unbalanced, where the relatively short patterns have the largest proportion in all the patterns. In the rule sets of Snort, 90% of the patterns are shorter than 36 bytes, though the longest pattern is 122 bytes. A similar distribution occurs in the rule sets of ClamAv. 90% of the patterns are shorter than 70

bytes, though the longest one is 172 bytes. In GPU programming language OpenCL all threads within a wave (thread group) execute a set of instructions as SIMD [17] units. SIMD describes computers with multiple processing elements that perform the same operation on multiple data points simultaneously. Therefore, we need to consider the distribution of the pattern length, making the execution of all the threads in a wave as consistent as possible to better improve the parallel capability.

2) The size of the AC automata: The size of the AC automata point to the storage space of the AC automata's corresponding state transition table (STT). When using GPU for large-scale pattern matching, the matching speed is mainly limited by the memory storage capacity, which is not big enough to load both the STT and all the input data at one time. The trade-off strategy is to separate the input data into several chunks according to the memory capacity of GPU, and load each chunk into the memory for processing sequentially. Therefore, how to compress the storage space of the AC automata becomes one important aspect to improve the matching efficiency on GPU.

2.3. Stt-Partition-Based Parallel Ac Algorithm

In this section, we propose a novel algorithm that efficiently exploits the parallelism of the AC algorithm. We call the new algorithm STT-partition-based parallel AC (SPAC for short). Systematic illustration of the new algorithm is as follows.

2.4. State ID Reordering

As data parallel approach [16], in order to make full use of GPU's parallel computing capability, our design of the automata is based on the scheme of failureless-AC automata. Fig. 4 shows the original automata of the patterns {era, her, heroic, oil} with state IDs. In our approach, we use breadth-first search algorithm (BFS) to reorder the state IDs. The automata in Fig. 5 is given with reordering IDs. We can see that in the same layer, the state IDs range continuously. In this way, adjacent states in the automata will appear in the corresponding STT within short distance. The state ID reordering procedure is a pre-processing step for further compressing the AC automata.

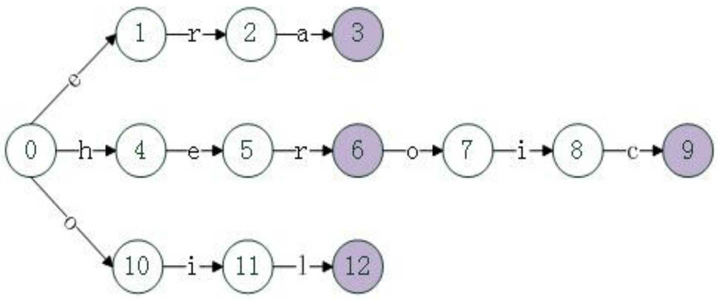


Fig. 4. Failureless-AC automata of the patterns {era, her, heroic, oil}.

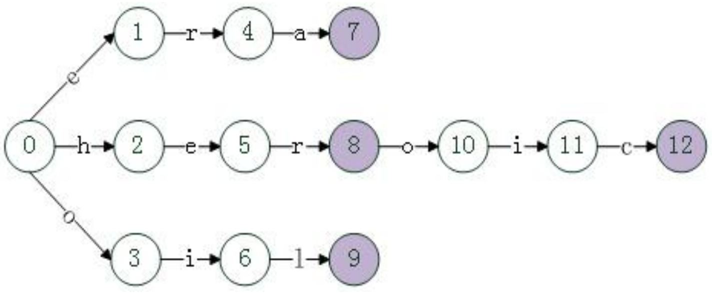
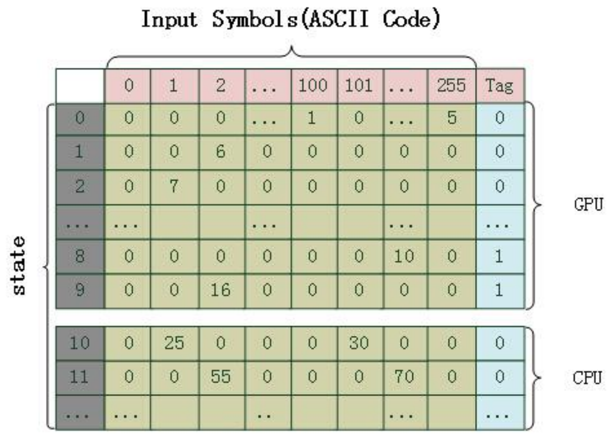


Fig. 5. State's ID reordering of Failureless-AC automata.

2.5. State ID Reordering

A 2-dimensional matrix, called State Transition Table (STT for short), is used to store the automata structure. Each row represents a state in the automata and each column represents a character. The entry STT[i][j] denotes the next state to jump to when given a state i and an input character j. Suppose the input data has 256 possible characters (256 characters in ASCII table), the corresponding STT needs 257 columns. Fig. 6 illustrates the STT structure for the failureless-AC automata.


Input Symbols(ASCII Code)
state
GPU
CPU

	0	1	2	...	100	101	...	255	Tag
0	0	0	0	...	1	0	...	5	0
1	0	0	6	0	0	0	0	0	0
2	0	7	0	0	0	0	0	0	0
...	...			...			...		...
8	0	0	0	0	0	0	10	0	1
9	0	0	16	0	0	0	0	0	1
10	0	25	0	0	0	30	0	0	0
11	0	0	55	0	0	0	70	0	0
...	...			..			...		...

Fig. 6. State transition table of Failureless-AC automata.

When given a large-scale input data, the automata may contains hundreds of thousands of nodes. Both the STT and the input data, as well as the processed results need to be stored in GPU's memory, which would greatly exceed GPU's memory. In our work, we focus on reducing the size of the STT store in GPU without affecting the matching performance.

From previous analysis we can conclude that in the actual pattern set, most of the pattern strings are very short and they can be matched within relatively short path in the failureless-AC automata. In previous procedure, we have reordered the state IDs and that the adjacent states in the automata will appear in the corresponding STT within short distance. In this way, a large number of the pattern strings can be matched by just referring to a relatively small part of the STT from the front. In our method, we partition the reordered STT into two parts: "upper part" and "bottom part", where "upper part" is stored in GPU for matching the majority data of the input string while the "bottom part" is stored in CPU of the same computer for matching the remaining part of the input string. This approach has several important features which reduce the storage space for STT in GPU and guarantee the matching efficiency.

1) In the actual pattern set, the proportion of long pattern strings is very small. Therefore, the "upper-part" of the STT stored on GPU can finish the majority part of the matching task. The size of the "upper-part" depends on the length distribution of the pattern strings. In our work, we use the average length of all the pattern strings as the partition depth of the automata. The "upper-part" of the STT will contain all the states within this depth in the automata. Thus the partial STT stored in GPU is just 45%~50% of the original size.
2) Since "upper-part" of the STT save the state thread for pattern matching will not experience as many transitions as before since they will stop in the STT partition boundary. Therefore, it makes all the threads in the same wave as consistent as possible, which improves the parallel performance.

In order to ensure complete match, the results of the GPU need to be further processed by CPU. After the processing of GPU, nearly 95% of the input text has been filtered since it is proved to contain patterns or absolutely not. So only less than 5% of the input text need further processing on CPU in extremely short time.

3. Implementation

The large-scale pattern matching algorithm SPAC takes full advantages of GPU and CPU. The algorithm will report a matched pattern at its starting position in the input text. If a pattern is a prefix of another longer one, they would be reported at the same position. In the following, we will explain our implementation of the algorithm in detail.

3.1. The Architecture of SPAC

Fig. 7 shows the architecture of SPAC, each GPU can manipulate a fixed-size text. Given a large-scale text, we split it into several small parts and sent them sequentially to GPU for processing. For each text part, GPU first matching the text according to the "upper-part" of the STT. In order to ensure complete match, the non-finished positions in the text will be further processed by CPU according to the "bottom-part" of the STT. Since the large-scale text need to be processed iteratively, operations on GPU and CPU can be executed parallel.

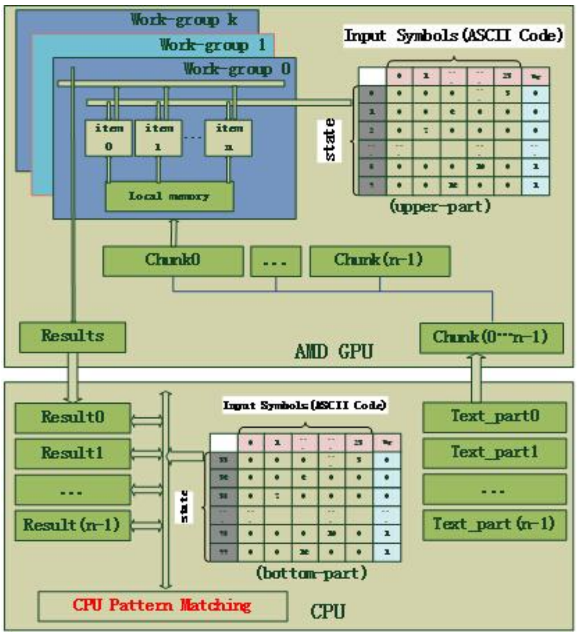


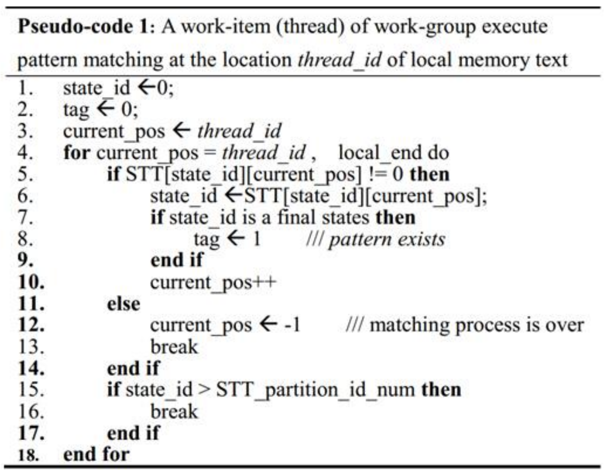
Fig. 7. The architecture of our proposed SPAC.

We also take full advantages of the hierarchical storage structure on GPU, including its global memory and local memory. As we split the input text into chunks, each work-group processes a chunk. In order to accelerate the scanning process, each work-item (or thread) will load the text chunk from global memory to its corresponding local memory.

3.2. Pattern Matching on GPU

GPU is suitable for parallel computing by keeping the consistency between different threads. As illustrated in Section 3(B), we proposed the STT partition method to guarantee the consistency and reduce the STT storage space on GPU.

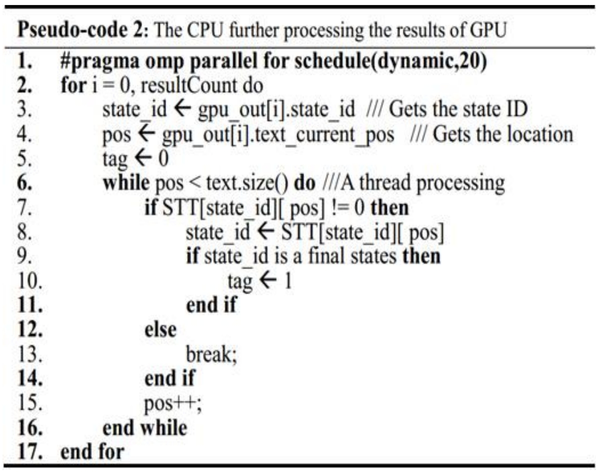
In our algorithms on GPU, each character is allocated to a thread. Fig. 8 is the pseudo code of a work-item (or thread) executing pattern matching at the text position *thread_id* in the local memory on GPU. In pseudo-code 1, *state_id* is referred to the current matching state (0: the root state). *t*ag marks the current matching results (0: no pattern matched; 1: matched). local_*end* represents the length of current processed chunk in the local memory. *STT_partition_id* is a STT partition ID indicating the corresponding relation between "upper-part" and "bottom-part" in STT. Line 5 is to determine whether the current state has next state transition. Whether the current state can be matched to a pattern is checked in line 7. If the current state ID exceeds *STT_partition_id*, this thread on GPU will end the matching process and wait for further processing by CPU".



Pseudo-code 1: A work-item (thread) of work-group execute pattern matching at the location *thread_id* of local memory text

```
1.   state_id ←0;
2.   tag ← 0;
3.   current_pos ← thread_id
4.   for current_pos = thread_id ,   local_end do
5.       if STT[state_id][current_pos] != 0 then
6.           state_id ←STT[state_id][current_pos];
7.           if state_id is a final states then
8.               tag ← 1        /// pattern exists
9.           end if
10.          current_pos++
11.      else
12.          current_pos ← -1        /// matching process is over
13.          break
14.      end if
15.      if state_id > STT_partition_id_num then
16.          break
17.      end if
18.  end for
```

Fig. 8. The algorithm of SPAC on GPU.



Pseudo-code 2: The CPU further processing the results of GPU

```
1.   #pragma omp parallel for schedule(dynamic,20)
2.   for i = 0, resultCount do
3.       state_id ← gpu_out[i].state_id  /// Gets the state ID
4.       pos ← gpu_out[i].text_current_pos   /// Gets the location
5.       tag ← 0
6.       while pos < text.size() do ///A thread processing
7.           if STT[state_id][ pos] != 0 then
8.               state_id ← STT[state_id][ pos]
9.               if state_id is a final states then
10.                  tag ← 1
11.              end if
12.          else
13.              break;
14.          end if
15.          pos++;
16.      end while
17.  end for
```

Fig. 9. The algorithm of SPAC on CPU.

3.3. Pattern Matching on CPU

CPU needs to finish two tasks: one is to send input text to GPU for coarse matching and the other one is to further process the matching results from GPU.

Fig. 9 shows the pseudo code of SPAC on CPU. The *gpu_out* is an array, which denotes the GPU's results needed to be further processed by CPU. The *resultCount* is the size of *gpu_out*. In pseudo-code 2, code from line 6 to line 16 describes the matching procedure of a thread on CPU. Line 7 is to determine whether the current state has next state transition. Whether the current state can be matched to a pattern is checked in line 9. To accelerate execution on CPU, we use Open MP for multithreading programming.

Table 1. Experimental Results of the Evaluated Algorithms AC, PFAC, SPAC on Various Datasets

Dataset	Test Data Size(MB)	#Pattern	Algorithm	STT Storage Space(MB)		Run-Time(ms)				Matching Speed (Gbps)
				GPU	CPU	GPU	CPU	I/O	Total	
URL blacklist	700	10,000	AC	-	122.81	-	2550	-	2550	2.14
			PFAC	122.81	-	83.10	-	13.80	96.90	56.44
			SPAC	**58.95**	**63.83**	**82.70**	**0.001**	**13.80**	**96.50**	**56.67**
Sythetic Dataset 1	800	10,000	AC	-	116.67	-	1910	-	1910	3.27
			PFAC	116.67	-	59.44	-	15.56	75	83.33
			SPAC	**50.17**	**66.01**	**56.34**	**0.035**	**15.56**	**71.94**	**86.88**
Sythetic Dataset 2	800	20,000	AC	-	223.98	-	2110	-	2110	2.96
			PFAC	223.98	-	82.38	-	17.66	100.04	62.48
			SPAC	**105.27**	**118.71**	**72.99**	**0.152**	**15.56**	**88.70**	**70.46**

4. Experiment and Evaluation

In the experiment settings, we compared our proposed STT-Partition-based Parallel AC (SPAC) algorithm with some representative algorithms, including classic Aho-Corasick designed on CUP and PFAC designed on

GPU. Our implementation is in C++ on Inter Core i7-3820 CPU with 32GB memory, and in OpenCL (version 1.2) on AMD Hawaii GPU with 1GB dynamically allocated memory.

The experiments are carried out on three different kinds of datasets, including the URL blacklist [18] benchmark and two synthetic datasets. Some details of the datasets are described as follows.

1) URL blacklist: 10,000 URLs (length > 7) are obtained from the URL blacklist website as the pattern strings. The URL blacklist is searched against about 8 million URLs (763MB) extracted from WLAN traffic.
2) Synthetic dataset 1: The input text is constructed by randomly generating 800MB characters and randomly inserting all the patterns in above URL blacklist dataset. The test pattern strings are the same as the ones in URL blacklist.
3) Synthetic dataset 2: The construction procedure is similar to synthetic dataset 1, differing in that we randomly insert each the pattern string to the generated text two times, finally forming the datasets containing 800MB input text and 20,000 pattern strings.

Table 1 shows the experimental results on the above datasets. Since our SPAC algorithm partitions the original STT, the partition depth here is set to the *average length* of all the pattern strings in each dataset. We compare the performance in the following two aspects.

4.1. Memory Space

It is clear to see in Table 1 that, on average, our approach only costs 45%~50% of PFAC's storage space for STT on GPU. In our experiments, the maximum size of dynamical memory on GPU is 1GB. With the increasing size of the pattern set, the scale of STT increases at the same speed. The second synthetic dataset is more than 800M and the corresponding STT is more than 200MB, which is two times of the first synthetic dataset. Faced with data in such scale or even larger scale, PFAC cannot load the whole STT and input data into GPU at one time. As algorithm requires loading the whole STT into GPU, it has no choice but to separate the input text into several small part and transmit each part to GPU sequentially. As a contrast, our SPAC just needs to load partial STT (45%~50% of the original size) into GPU, thus sparing larger space to load more input text at one time. In this way, our approach decreases the I/O times compared with PFAC on large-scale dataset. In Table 1, the total run time of our approach on synthetic dataset 2 is shorter than that of PFAC, which has strongly proved our conclusion.

4.2. Run-Time

In this section, we compare the total run time of each algorithms. For the classic Aho-Corasick algorithm, the run time refers to the execution time on CPU only. For PFAC, the run time contains both the processing time on GPU and I/O time for data transmission between GPU and CPU. The run time of our proposed SPAC includes the processing time on both GPU and CPU, as well as I/O time. The comparison in Table 1 shows that our approach is faster than the original AC algorithm designed on CUP by 26 times on average. The performance is comparable with PFAC on the URL blacklist dataset and synthetic dataset 1. Meanwhile, the running time of our SPAC is 11.3% shorter than that of PFAC on synthetic dataset 1 because of the decrease of I/O time analyzed in above section. From the point of run-time, our approach is comparable with AC and PFAC for relatively small-scale testing data and will outperform these two approaches for large-scale testing data.

5. Conclusion

In this paper, we proposed a STT-Partition-based Parallel AC (SPAC) algorithm on GPU and CPU. It can effectively reduce the storage space on GPU by separating the original state transition table into two parts, one small part is kept in GPU for major filtering tasks and the other part is stored in CPU for further processing the relatively small-scale filtered results. Experimental performance indicates that our approach costs only 45%~50% of the space compared with the representative data parallel approach on GPU. Furthermore, our total matching time is comparable with that of state-of-the-art approaches, which is about 26 times faster than the original AC algorithm on CPU. We proposed a STT-Partition-based Parallel AC (SPAC) algorithm that partitions the automata and adopted the average length of pattern strings in rule sets as partition depth. Therefore, our future work will be studying the impact of the partition depth of the automata on the algorithm performance.

References

[1] Aho, A. V., & Corasick, M. J. (1975 June). Efficient string matching: An aid to bibliographic search. *Communications of the ACM*, *18(6),* 333–340.

[2] Snort is an open source network intrusion prevention and detection system. Retrieved from http://www.snort.org/.

[3] ClamAV is an open source (GPL) antivirus engine. Retrieved from http://www.clamav.net/lang/en/.

[4] Sidhu, R., & Prasanna, V. K. (2001, March 29-April 2), Fast regular expression matching using FPGAs. *Proceedings of Field-Programmable Custom Computing Machines* (pp. 227-238). Rohnert Park, CA, USA.

[5] Hutchings, B. L., Franklin, R., & Carver, D. (2002). Assisting network intrusion detection with reconfigurable hardware. *Proceedings of Field-Programmable Custom Computing Machines* (pp. 111-120). Boston, Massachusetts, USA.

[6] Moscola, J., Lockwood, J., Loui, R. P., & Pachos, M. (2003, April 9-11). Implementation of a content-scanning module for an internet firewall. *Proceedings of Field-Programmable Custom Computing Machines* (pp. 31-38). Boston, Massachusetts, USA.

[7] Jung, H. J., Baker, Z. K., & Prasanna, V. K. (2006 April). Performance of fpga implementation of bit-split architecture for intrusion detection systems. *Proceedings of Parallel and Distributed Processing Symposium* (pp. 25-29). Rhodes Island, Greece.

[8] Dharmapurikar, S., & Lockwood, J. (October 26-28). Fast and scalable pattern matching for content filtering. *Proceedings of Architecture for Networking and Communications Systems* (pp. 183-192). *Princeton,* NJ, USA.

[9] Kumar, S., Dharmapurikar, S., Yu, F., Crowley, P., & Turner, J. (2006, October). Algorithm to accelerate multiple regular expressions matching for deep packet inspection. *Proceedings of ACM SIGCOMM: Vol. 36* (pp. 339-350). Pisa, Italy.

[10] CUDA™ is a parallel computing platform and programming model invented by NVIDIA. Retrieved from http://www.nvidia.com/object/cuda_home_new.html#sthash.aWNVFWZF.dpuf.

[11] OpenCL™ is the first truly open and royalty-free programming standard for general-purpose computations on heterogeneous systems. Retrieved from http://developer.amd.com/tools-and-sdks/opencl-zone/.

[12] Tumeo, A., Villa, O., & Sciuto, D. (2010, May). Efficient pattern matching on gpus for intrusion detection systems. *Proceedings of Seventh ACM Int'l Conf. Computing Frontiers* (pp. 87-88).

[13] Tumeo, A., Secchi, S., & Villa, O. (2011, February 24-25). Experiences with string matching on the fermi architecture. *Proceedings of Architecture of Computing Systems* (pp. 26-37). Como, Italy.

[14] Kopek, C. V., Fulp, E. W., & Wheeler, P. S. (2007, October 29-31). Distributed data parallel techniques for content-matching intrusion detection systems. *Proceedings of Military Communications Conference* (pp. 1-7). Orlando, FL, USA.

[15] Vasiliadis, G., & Ioannidis, S. (2010, September 15-17), Gravity: a massively parallel antivirus engine.

Proceedings of 13th International Symposium Research in Attacks, Intrusions and Defenses 2010 (pp. 79-96). Ottawa, Ontario, Canada.

[16] Lin, C. H., Chien, L. S., & Chang, S. C. (2013, October). Accelerating pattern matching using a novel parallel algorithm on gpus. *IEEE Trans. on Computers*, *62(10)*, 1906-1916.

[17] Intro. of SIMD in Wikipedia. Retrieved from http://en.wikipedia.org/wiki/SIMD.

[18] URL blacklist dataset link. Retrieved from http://urlblacklist.com/.

Xudong Liu was born in 1989 and will receive M.S. degree in 2015, in computer science and technology from Beijing University of Posts and Telecommunications, located in China. He received his B.S. degree in 2012, in Software Engineering of North University, located in China. His research interests include algorithm on strings and automata, information security, heterogeneous computing, etc.

Yanbing Liu was born in 1981 and received both his Ph.D. degree and M.S. degree in computer science from Institute of Computing Technology (ICT), Chinese Academy of Sciences (CAS) in 2012 and 2006 respectively. Prior to joining ICT, he received his B.S. degree in mathematics from Wuhan University of Technology in 2003. Now he works as an associate professor for Institute of Information Engineering (IIE), Chinese Academy of Sciences (CAS). His research interests include algorithm on strings and automata, graph algorithms, algorithm engineering, etc.

Jian Li is a Ph.D. holder and an associate professor of Beijing University of Posts and Telecommunications, his research interests include quantum information, quantum computation, quantum communication security, electronic commerce and artificial intelligence.

Jing Yu is working as a research assistant in Institute of Information Engineering (IIE), Chinese Academy of Sciences (CAS). She received her B.S. degree in automation science from MinZu University, China in 2011 and her M.S. degree in pattern recognition and intelligent systems from the School of Automation Science and Electrical Engineering, Beihang University, China in 2014. She used to work as an intern in Microsoft Research Asia, Intel Labs China, and Samsung Electronics (China) R&D Center. Her research interests include graph pattern matching, multimedia information retrieval, computer vision and digital image processing.

Jianlong Tan was born in 1974 and received his Ph.D. degree in computer science from Institute of Computing Technology (ICT), Chinese Academy of Sciences (CAS) in 2003. Prior to joining ICT, he received both his M.S. degree and B.S. degree in computer science from Xiangtan University in 2000 and 1997 respectively. Now he works as a professor for Institute of Information Engineering (IIE), Chinese Academy of Sciences (CAS). His research interests include string matching algorithms, network information processing, hardware design, etc.