Fast and Memory Efficient NFA Pattern Matching using GPU

Yeim-Kuan Chang and Yu-Hao Tseng
Department of Computer Science and Information Engineering
National Cheng Kung University
Tainan, Taiwan
Email:ykchang@mail.ncku.edu.tw

Abstract—**Network intrusion detection system (NIDS) is mainly designed to monitor the malicious packets spreading on the Internet. With pre-defined virus signatures called patterns, NIDS can find out whether these pre-defined patterns exist in the packet's payload. GPU can be useful to effectively accelerate pattern matching process due to abundant parallel hardware threads. In this paper, we propose a constrained NFA (CNFA) scheme to store complex regular expressions in limited memory of GPU effectively. CNFA is constructed from the original NFA based on the subset construction algorithm that converts NFA to DFA. Compared to original NFA and DFA, CNFA imposes a constraint that each state can only have at most two transitions (self-loop and non-self-loop) for each character. Based on our experimental results, CNFA can achieve the performance of about 100 Gbps for one of the tested rule sets on GPU. Also, CNFA only needs 18% of memory needed in iNFAnt. In addition, CNFA can be used for more complex rule sets that is not possible to be implemented in iNFAnt.**

Keywords- Deep Packet Inspection; DFA; NFA; GPU; Pattern Matching; Regular expression.

I. INTRODUCTION

Due to the popularity of the Internet, Internet traffic increases exponentially. Network security has become a significant issue because more malicious attacks, such as malwares and viruses, have spread on the Internet. Traditional protections, such as firewalls, have been inadequate to protect our computers. Instead, Network Intrusion Detection System (NIDS) has been diffusely used to maintain the security of network activities. The main task of NIDS is to examine the payload of each input packet to find out whether or not the packet contains suspicious contents based on the pre-defined rules. If there are some suspicious contents contained in the input packet, NIDS reports all occurrences of these suspicious contents associated with the matched rules.

In computer science, pattern matching algorithms are used to check a given text of tokens for the presence of the constituents of some patterns. In other words, we often utilize the idea of pattern matching to develop the NIDS. According to different forms of rules, pattern matching is divided into string and regular expression matching. For regular expression matching, most people implement finite state machine, such as non-deterministic (NFA) and deterministic (DFA) finite automata, to perform the matching operations. We can first translate a regular expression into a parse tree and use one of these algorithms, which contain Thompson [10] and Glushkov [3] algorithms, to build a NFA on the basis of the parse tree. We can even transform the NFA into an equivalent DFA. With the rapid expansion of networks, traditional software-based approaches are not able to satisfy these demands. A lot of researches attempt to improve the performance of pattern matching on different devices. For example, Baker and Prasanna [11] proposed variants version of KMP algorithm and implement on FPGA. Zha and Sahni [12] and Lin et al. [13] proposed a Parallel Failureless-AC (PFAC) algorithm that uses Computer Unified Device Architecture (CUDA) to implement modified version of AC algorithm on GPU which NVIDIA corporation produces. PFAC is the simple version of AC algorithm but suitable for GPU. PFAC effectively streamlines AC algorithm by utilizing massive threads of GPU. The main idea of PFAC is to assign each thread to process input stream at the corresponded position of stream. Each thread accesses the same goto function whose initial state has no self-loop. In the above hardware, GPU has high scalability and low overhead. When we choose CUDA as our programing language, implementing General-purpose computing on graphics processing units (GPGPU) becomes an easy work.

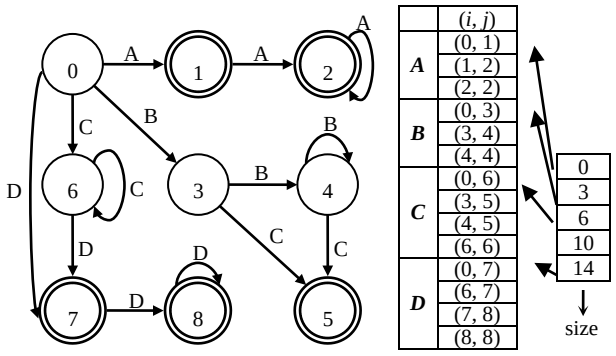
In this paper, we propose our scheme which decreases memory consumption of the original NFA and utilizes SIMT (Single Instruction Multiple Thread) of GPU to accelerate NFA's searching procedure to get the better performance. In our experiment, we can reach performance around 6 Gbps at worst case.

The rest of the paper is organized as follows. Section II describes the related work. In Section III, gives a detailed description of the proposed scheme. Section IV outlines the implementation on GPU. Section V presents the experimental results that are compared to the existing GPU implementation called iNFAnt [1]. Finally, our conclusions are stated in Section VI.

II. RELATED WORK

In this paper, we focus on regular expressions that can specify a finite set of strings mainly used in pattern matching. Traditional software-based approaches are unable to meet the performance requirement of the NIDS. Several researches have attempted to improve the performance by using GPU. There is a GPU-based parallel regular expression matching engine, iNFAnt [1]. iNFAnt adopts NFA to support a very large complex rule set that are otherwise hard to solve. iNFAnt is explicitly designed and developed for running on

 ISBN: 978-1-61208-503-6



(a) NFA transition graph. (b) Transition table with an ancillary table.

Figure 1. Character-first representation in iNFAnt for pattern "(A+|B+C|C*D+)" and Σ = {A, B, C, D}.

GPU consisting of a large number of threads. This parallelism is exploited to handle NFA and to process multiple packets at once, thus get better performance.

Compared to traditional NFA, iNFAnt adopts a character-first representation for state transition graph. The character-first representation keeps a list of transitions that will be triggered by each of the 256 characters. This list can grow very large so it must be stored in global memory of GPU, together with an ancillary data structure that records the index in character-first transition table for the first transition of each character in order to perform easy random lookups. Figure 1 illustrates an example of iNFAnt.

III. PROPOSED SCHEME

Pattern matching for regular expressions (RegEx) patterns is performed by searching a finite state automata (FSA) built from these patterns. Researchers are familiar with non-deterministic (NFA) and deterministic (DFA) finite automata. Automata theory confirms that both NFA and DFA are true in terms of expressiveness, but their practical properties like memory requirement and number of active states are much different. According to automata theory, each state in NFA may transit to zero or more states after processing an input character. Furthermore, each state in NFA may also transit to zero or more states without processing any input character, which we call ε-transition. Different from NFA, each state in DFA transits to only one state for an input symbol. DFA is faster than NFA because only one active state exists in any cycle. However, DFA is less memory-efficient than NFA because it requires a lot of memory to store the transition table. NFA suffers from a higher cost to traverse many states per input character but it requires much less memory than DFA. For some complicated regular expressions, it may not be possible to build the corresponding DFAs because the required memory may exceed the amount of memory that a computer can support. In order to store large RegEx sets in a scant amount of memory, we choose the architecture of NFA.

Our proposed scheme is inspired by the subset construction algorithm that converts NFA to DFA. As we know that all NFAs can be converted into equivalent DFAs by using the subset construction algorithm [2]. It is well

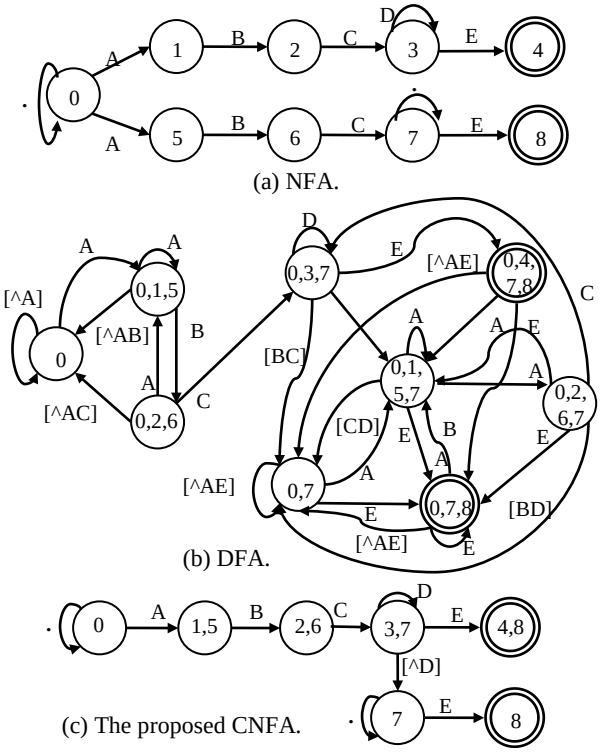


Figure 2. Finite State Automata for {ABCD*E, ABC.*E}.

known that complex syntaxes like ".*" and "[^\r\n]*" in RegEx lead to so-called state space explosion in DFA because a state in DFA is associated with a subset of states in NFA and most subsets contain the NFA states that are originated from these complex syntaxes. One of the advantages of subset construction algorithm is that many different states generated from the same prefix of different rules are merged and so some different transitions labeled with the same character are merged into the same transition. Consider two rules "ABCD*E" and "ABC.*E" whose NFA and DFA are shown in Figure 2. Transitions (state 0 → state 1) and (state 0 → state 5) in NFA are transformed into the transition (state {0} → state {0, 1, 5}) in DFA. The reason is that these two patterns have the same prefix "A" and thus reduces some redundant states and transitions in NFA. Obviously, the DFA in Figure 2(b) is more complex than the NFA in Figure 2(a). States 0 and 7 in NFA exist in most subsets of DFA due to ".*" syntax. This is the disadvantage that makes the number of DFA states blow up. Besides, we also observe that states 1 and 5 (or 2 and 6) in NFA exist in the same subsets of DFA. Reducing redundant states can mitigate the process of updating active states in NFA. And less transitions in NFA can lead to a better throughput. In order to utilize the observations described above, we propose a new NFA called constrained NFA (CNFA) that considers the regular expressions containing the syntax of ".*" and counters.

To show the proposed CNFA and original NFA, we use a simple rule set example containing two rules, "ABCD*E" and "ABC.*E", in Figure 2 and explain the difference between them. Assume that character set only includes A, B,

TABLE 1. DEFINITIONS OF CNFA CONSTRUCTING ALGORITHM.

notation	Description
S_i ($S_{i'}$)	State i in NFA (State i' in CNFA)
$\Phi(i')$	NFA states associated with state i' in CNFA
Σ	The character set
GroupNum	# of tuples in *StateClosureGroup*
StateClosureGroup	NFA states reachable from one of the NFA states in $\Phi(i')$ are divided into *GroupNum* subsets based on repetitions. These *GroupNum* subsets are put into *StateClosureGroup* which is a *GroupNum*-tuple list.

C, D and E. The proposed CNFA is shown in Figure 2(c). It is obvious that the proposed CNFA has fewer states than NFA. Because we merge some states in NFA, we can get multiple matched rules in final states of the proposed CNFA.

We will first illustrate the procedure of converting NFA to CNFA. Table 1 lists some definitions of our CNFA constructing algorithm to help us understand the converting procedure. Figure 3 shows the pseudo code of the proposed CNFA construction algorithm. Figure 4 show the NFA and CNFA for rule set {ABCD*E, ABC.*E, CDEA+C, CDE.*C} and Σ = {A, B, C, D, E}. Table 2 shows the complete list of Si' and the associated Φ(i') Figure 4(b). For example, S0 is the initial state in NFA, S5' denotes that the state 5 in CNFA is equivalent to a set of corresponding NFA states where Φ(5') = {S3, S7}. According to line 1 of the pseudo code, we add the initial state of the proposed CNFA. Because NFA has no ε-transitions, Φ(0') is set to {S0}. From lines 2-15, we process each CNFA state Si' and add accessible transitions labeled with the character α from Si' to Stmp' that is the temporary state of CNFA during the conversion. Unlike the subset construction algorithm [2], we divided the process of computing accessible transitions from a state to its next state into two parts, the self-loop transition and the non-self-loop transition for each state in Φ(i'). In lines 4-8, we add an accessible non-self-loop transition Si' to Stmp' labeled with character α. In lines 9-13, the same codes are performed for self-loop transition. In line 4, function Extract-NonSelfLoop-States(Φ(i'), α) collects all the states j from non-self-loop transitions i → j with label α in the original NFA for i ∈ Φ(i'). In other words, it computes a set of NFA states that can be reached from one of the NFA states in Φ(i') by a non-self-loop transition with label α. Similarly, function Extract-SelfLoop-States(Φ(i'), α) collects all the states j (i.e., Φ(tmp')) from self-loop transitions i → j with label α in the original NFA for i ∈ Φ(i'). As a result, we obtain the full table of the proposed CNFA except the information of final states. Therefore, in line 16-20, we mark the final states of CNFA based the information of final states in the original NFA. Here we will give a graphic example of converting NFA to CNFA in Figure 4 by using rule set containing rules "ABCD*E", "ABC.*E", "CDEA+C" and "CDE.*C". Notice that NFA is constructed according to Glushkov algorithm [3]. In this paper, we will also use the character-first format to store the transition table of the proposed CNFA.

A. Compression by Default State

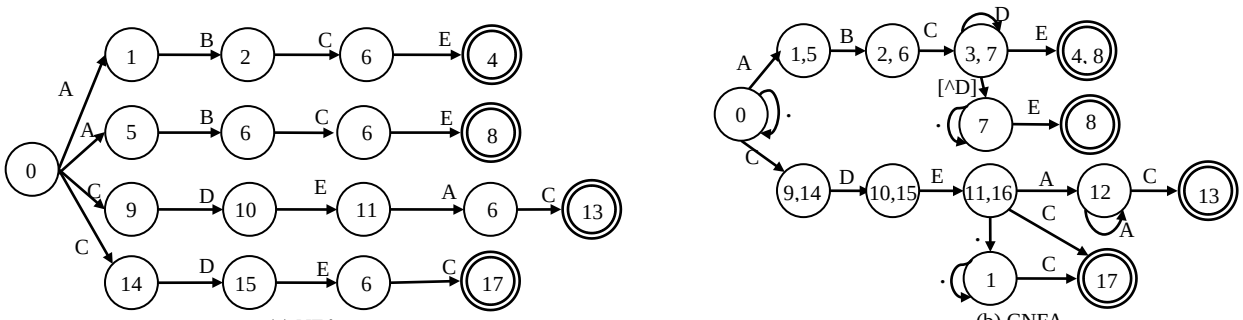
Before describing the details of the proposed compression schemes, we analyze the number of transitions for the NFA states based on Snort534 [4]. We observe that most characters have a large constant number of transitions. Every character appears to be the transition symbol for at least 200 transitions. In other words, a source state has more different transitions to the same destination state. This appearance takes place due to overlapping syntaxes such as "." and "[^\r\n]". As a result, based on character-first data structure for transition table, more memory is required. In order to reduce the number of transitions per symbol for character-first for transition table, we record default transition for each state by using a simple traditional state-first transition table. Take the CNFA in Figure 4(b) as an example. The character-first transition table is shown in Figure 5(a). We build a default state table (DST) in Figure 5(b) including four fields, the default next state of each state, default character and don't care character of the default transition, and the compliment flag. If don't-care character bit is 0, the default transition follows the default character. Otherwise, the default transition is unconditional, i.e., the default transition is taken no matter what the input character is. If the complement flag is set to 1 and the default character is C, the default transition will follow the symbol ^C (i.e., any character other than C).

```
Convert_NFA_to_CNFA (NFA)
01 CNFA = {0'} and Φ(0') = {S0}; tmp' = 1;
02 for each non-processed state i' in CNFA {
03    for each α in Σ {
04       Φ(tmp') = Extract-NonSelfLoop-States(Φ(i'), α);
05       if (Stmp' ∉ CNFA) CNFA = CNFA + Stmp'; tmp'++;
06       add a transition Si' to Stmp' labeled with α;
07       Φ(tmp') = Extract-SelfLoop-States(Φ(i'), α);
08       if (Stmp' ∉ CNFA) CNFA = CNFA + Stmp'; tmp'++;
09       add a transition Si' to Stmp' labeled with α;   }
10 }
11 for each state i' in CNFA
12    if (any Si ∈ Φ(i') is a final state in the NFA)
13       Set Si' as final state in CNFA;
14 return CNFA;
```

Figure 3. Pseudo code of constructing CNFA.

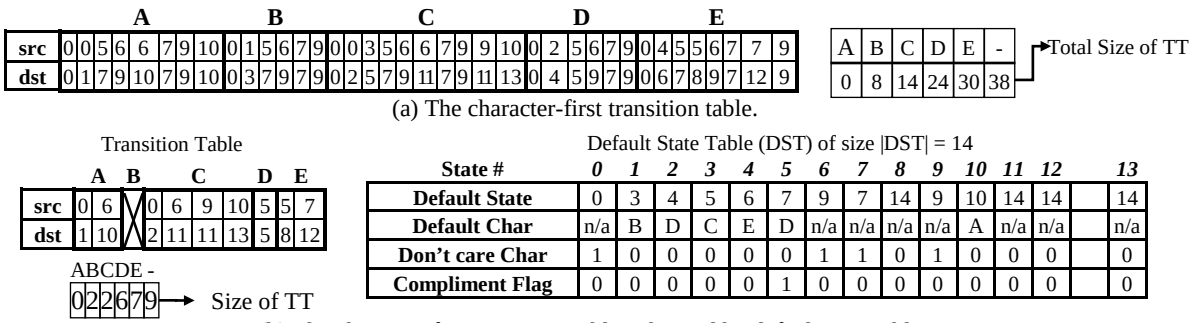
B. Counter

As far as we know, counter is used to solve repetitions of RegEx. In terms of data structure, we store the information of repetition in a table which keeps a list of counter pairs (min, max) for every state. Therefore, the proposed CNFA construction algorithm has to be modified. As shown in Figure 6(a), the reachable NFA states, *Φ(tmp')*, from any state in *Φ(i')* returned by functions *Extract-SelfLoop-States*() and *Extract-NonSelfLoop-States*() need further process to consider the repetition conditions. What we do is to divide *Φ(tmp')* into several groups based on the repetition conditions performed by function *UpdateCNFA*(). Lines 2-5 of *UpdateCNFA*() in Figure 6(b) are responsible for grouping based on the repetition conditions. *GroupRepState*() function divides Φ(tmp') into several groups when Φ(i') of current Si' does not contain a repetition state of NFA. *IdentifyLeaveRep*() function that is similar to *GroupRepState*() divides *Φ(tmp')* into two groups, one for the CNFA state leaving the repetition and the other for the CNFA state continuing the repetition when current Si is a repetition state.



(a) NFA. (b) CNFA.

Figure 4. NFA and CNFA for Rule Set = {ABCD*E, ABC.*E, CDEA+C, CDE.*C} and Σ = {A, B, C, D, E}.



(b) The character-first transition table enhanced by default state table.

Figure 5. The complete data structure for Figure 4(b).

IV. GPU IMPLEMENTATIONG

In this section, we will emphasize how to parallelize the general purpose procedure. First, all threads in the same block initialize and update the active state list together. Second, we also exploit parallelism that GPU offers to accelerate. We select valid transitions for the current symbol and all threads in the same block averagely sharing the workloads. Similarly, we use parallelism to speed up the process that access Default State Table when the design architecture contains Default State.

According to different requirements, we first consider that transition table, the default state table and then counters should be stored in suitable type of memory space. Because these tables need larger memory space and every task reads the same data structure, we choose global memory and texture memory that can be read by all threads on GPU. How large is texture memory is dependent on the size of global memory and texture memory is cached on chip. In some situations, texture memory will provide higher effective bandwidth by reducing memory requests to off-chip DRAM. So we store our tables in texture memory.

Furthermore, we have to find out memory space suitable for input streams. If input streams assigned in a block can be stored in enough shared memory, we will choose this storing mode. Because shared memory is faster than other memory spaces except register. And the famous problem of using shared memory is bank conflict. In order to get higher bandwidth, shared memory is divided into memory modules which are the same memory size when parallelizing memory accesses. The memory module is named as bank and different banks can be accessed at the same time. When all 16 threads of half-warp access the same memory address in the same bank, shared memory adopt the broadcast mode to respond requirements of half-warp. So we don't have bank conflict because all threads on the GPU read a character of the input stream from shared memory at every cycle.

V. EXPERIMENT RESULTS

Our experiments are based on three rule sets and we compare throughput and memory consumption with iNFAnt [1]. Moreover, we show the comparison of throughput and memory consumption with different compressed schemes. And also we show that the influence on throughput by workload per block.

All experiments were performed using a 4-core Intel Core i5-650 machine running at 3.2 GHz with 8 GB of RAM. GPU tests were implemented on the same platform equipped with one graphic card which is NVIDIA GeForce GTX 770. The GPU has 2 GB of RAM and 8 multiprocessors clocked at 1.11 GHz, and its compute capability is version 3.0. Though the GPU supports PCI-E 3.0, the motherboard on our PC supports only PCI-E 2.0. Finally, we install Ubuntu 12.04.4 LTS x64 on the PC.

In our experiments, we use 2 rule sets which is taken from iNFAnt [1] and an additional rule set to finish our experiments. The first rule set, *Snort534*, taken from [4] is composed of 534 regular expressions. *Snort534* can be partitioned into subsets that share an initial part while the tails differ. The second rule set, *L7-filter*, is from the L7 traffic classifier [5] and consists of 115 regular expressions. *L7-filter* is a very complex and irregular rule set where no

```
NFA_to_CNFA(NFA_in)
01 NFA_out = {}
02 Set the initial state S_0' in NFA_out and let Φ(0') = {S_0};
03 for each non-processed state i' in NFA_out {
04     for each α in Σ {
05         Φ(tmp') = ExtractSelfLoopStates(Φ(i'), α);
06         UpdateNFA_out(NFA_out, i', Φ(tmp'), α);
07         Φ(tmp') = ExtractNonSelfLoopStates(Φ(i'), α);
08         UpdateNFA_out(NFA_out, Φ(tmp'), α); }
09 }
10 for each state i' in NFA_out
12     if (any S_i ∈ Φ(i') is a final state in the NFA_in)
13         Set S_i',Φ(i') as one final state of the NFA_out;
14 return NFA_out;
```

(a) Algorithm Convert_NFA_to_CNFA_counter.

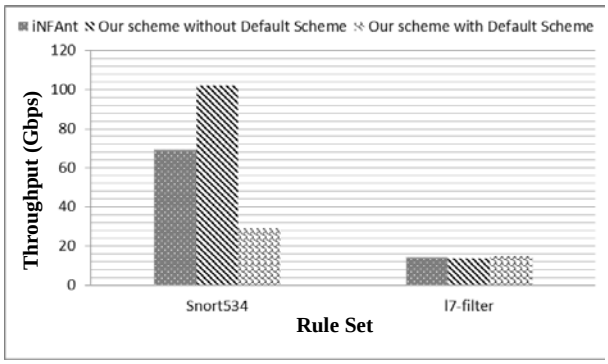
```
UpdateNFA_out(NFA_out, i', Φ(tmp'), α)
01 StateClosureGroup ={};
02 if (S_i',Φ(i') is a repetition state)
03    StateClosureGroup = IdentifyLeaveRep(Φ(tmp'));
04 else
05    StateClosureGroup = GroupRepState(Φ(tmp'));
06 for each set j in StateClosureGroup {
07     set temp state S_tmp',Φ(tmp'), where Φ(tmp') = set j;
08     if (S_i',Φ(i') is a repetition state )
09        for each state S_x in Φ(tmp') {
10           find state S_y' in NFA_out such that S_x ∈Φ(y') {
11              add a α-transition from S_i' to S_y' ;
12              Φ(tmp') =Φ(tmp')-{S_x};}
13        if Φ(tmp')is not empty {
14           create a new state S_tmp',Φ(tmp') in NFA_out;
15           add a α-transition from S_i' to S_tmp'; }
16     else
17        if (S_tmp',Φ(tmp') does not exust in NFA_out)
18           create a new state S_tmp',Φ(tmp') in NFA_out
19        add a α-transition from S_i' to S_tmp';
20 }
```

(b) Algorithm *UpdateNFA$_{out}$*.

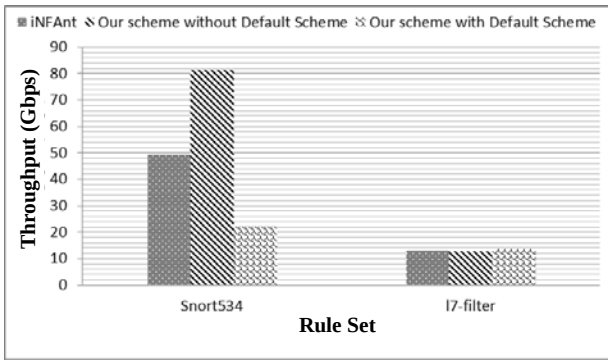
Figure 6. Pseudo code of the proposed CNFA construction.

special properties or common prefixes can be exploited. Because of these same rule sets, these comparisons between iNFAnt and the proposed CNFA deserve to be a reference. Finally, we also take an additional rule set which is Emerging Threats [6] Open optimized for *Suricata* [7] because previous two rule sets don't have complex regular expressions with repetitions. Table 4 shows feature of these rule sets.

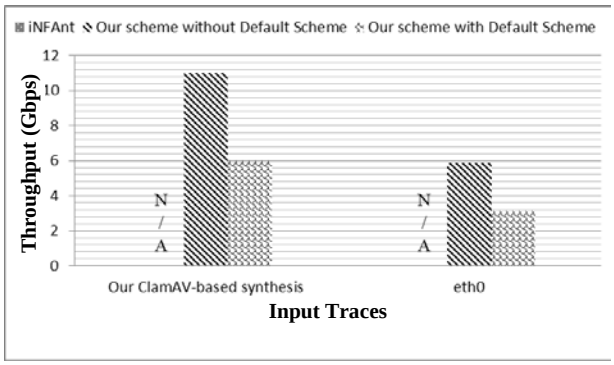
We compare the memory consumption of the proposed CNFA with iNFAnt [1] by using different rule sets. In addition, we also show the difference between these compressed schemes we proposed. Table 5 shows the comparison results for *Snort534*, *L7-filter*, and *Suricata*. Because there is no syntax of repetition in *Snort534*, *L7-filter*, the comparison has no experimental data of Counter scheme. We find out that CNFA decreases around 60% of memory consumption needed by iNFAnt for *Snort534*, but less than 1% of memory needed by iNFAnt for *L7-filter*. The reason is that *Snort534* has many common prefixes between different rules but *L7-filter* is a complex and irregular rule set where no special properties or common prefixes can be utilized. Compared to *Snort534* and *L7-filter*, rule set *Suricata* has complex repetitions. Due to complex repetition conditions, the complete data structures of iNFAnt [1], as well as CNFA without Counter scheme cannot be built for *Suricata*. So Table 5(c) shows only experimental data of CNFA with Counter scheme and CNFA with Counter scheme and the default state table.



(a) The synthesis trace.



(b) The eth0-Hex trace.



(c) Performance of Suricata with different input traces.

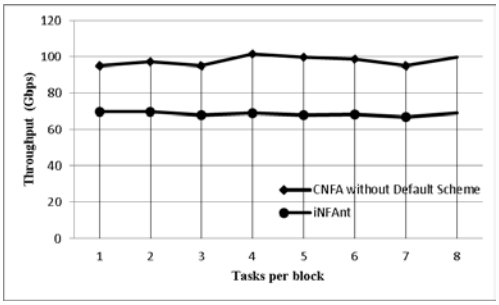
Figure 7. Performance with different input traces.

In Figure 7, we show the throughputs of iNFAnt compared with CNFA and CNFA with Default State Table. The difference between these two figures is that one uses our ClamAV-based [8] synthesis and the other utilizes eth0 taken from Defcon [9]. In Figure 7(c), we use *Suricata* to build our finite state machine, but the data structure of iNFAnt can't be built due to insufficient memory on the device. So we only show the performance of CNFA with counter scheme.

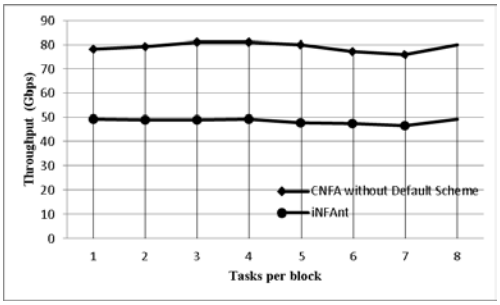
Finally, we test iNFAnt [1] and our proposed CNFA by controlling the number of tasks per block. We test the two input traces with rule set Snort534. Figure 8 shows the throughputs. We discover that increasing the number of tasks doesn't evidently accelerate the searching speed of iNFAnt [1]. The proposed CNFA gets most performance gain when the number of tasks is four.

TABLE 4. DETAILS OF RULE SETS

Snort534 [4]	*L7-filter* [5]	*Suricata* [6]
534	115	1195
common prefixes	complex, irregular, and no common prefixes	similar to snort and complex repetition



(a) The synthesis trace.



(b) The eth0-Hex trace.

Figure 8. Performances of different number of tasks.

TABLE 5. MEMORY (KB) CONSUMED by iNFANT, CNFA, AND CNFA WITH DEFAULT STATE (denoted by $CNFA_{de}$).

	iNFAnt	**CNFA**	**$CNFA_{de}$**
# of states	14,566	9,696	9,696
# of transitions	160,657	59,869	3,225
TT	627.6	233.9	12.6
DST	N/A	N/A	74
Match Table	56.9	2.1	2.1
Match List Flag	N/A	37.9	37
Total	685.5	274.9	125.7
Ratio over iNFAnt	x1.0	x0.40	x0.18

(a) *Snort534 with no repetition syntax.*

	iNFAnt	**CNFA**	**$CNFA_{de}$**
# of states	6,123	6,006	6,006
# of transitions	1,400,594	1,397,104	1,397,104
TT	5471.1	5427.2	914.4
DST	N/A	N/A	295.8
Matched Table	23.9	23.5	23.5
Matched List Flag	N/A	3.0	3.0
Total	5496.0	5454.7	1236.8
Ratio over iNFAnt	x1.0	x0.99	x0.23

(b) *L7-filter with no repetition syntax.*

	CNFA	**$CNFA_{de}$**
# of states	27,574	27,574
# of transitions	423,501	41,592
TT	1653.8	162.5
DST	N/A	353.3
Repetition Table	107.7	107.7
Matched Table	7.4	7.4
Matched List Flag	107.7	107.7
Total	1876.6	738.5

(c) *Suricata with repetition syntax.*

VI. CONCLUSION

In this paper, we proposed a scheme on GPU with efficient utilization of memory space to avoid the so-called state space explosion. The main for searching is that we assign each block on GPU to process appropriate amount of tasks. And we utilize massive amount of threads to accelerate the process of finding possible transitions to next states. Compared to iNFAnt [1], our scheme does not increase the complexity of NFA searching but accelerate the searching procedure because it can decrease the number of states for most rule sets. And we assign each thread to be responsible for number of tasks and avoid latency of block switch.

By utilizing the same rule set, our method can reach 101.94 Gbps for one of the tested rule sets. With our compression scheme, we need 18% of iNFAnt's memory usage with the same rule set. Besides, we proposed the architecture for counters to slow down state space explosion which is caused by repetitions. The proposed CNFA scheme obviously slows down our performance and we can construct the complete search data structure for more complex rule sets that is not possible for iNFAnt.

REFERENCES

[1] N. Cascarano, P. Rplando, F. Risso, and R, Sisto, "iNFAnt: NFA Pattern Matching on GPGPU Devices," ACM SIGCOMM Computer Communication Review, vol. 40 Num. 5, pp. 21-26, 2010.

[2] J. C. Marrtin, "Introduction to Languages and the Theory of Computation." McGraw Hill, pp.108, 2010

[3] V-M. Glushkov, "The abstract theory of automata." Russian Mathematical Surveys 16-5, pp.1–53, 1961.

[4] M. Becchi, C. Wiseman, and P. Crowley, "Evaluating Regular Expression Matching Engines on Network and General Purpose Processors," the 5th ACM/IEEE Symposium on Architectures for Networking and Communications Systems, 2009. pp. 30-39

[5] L7-filter, "Application Layer Packet Classifier for Linux". [Online]. Available. http://l7-filter.sourceforge.net/ 2013.06.05

[6] Emerging Threats. [Online]. http://emergingthreats.net/

[7] Suricata. [Online]. http://suricata-ids.org/ 2016.06.20

[8] ClamAV. [Online]. http://www.clamav.net/ 2016.05.03

[9] Defcon. [Online]. Available : http://www.defcon.org/ 2016.4.7

[10] K. Thompson, "Programming Techniques: Regular expression search algorithm." Communications of the ACM Volume 11 Issue 6, pp. 419-422, 1968.

[11] Z. K. Baker, and V. K. Prasanna, "Time and Area Efficient Pattern Matching on FPGAs," Proceedings of the 2004 ACM/SIGDA 12th international symposium on Field programmable gate arrays, pp.223-232, 2004.

[12] X. Zha, and S. Sahni, "GPU-to-GPU and Host-to-Host Multipattern String Matching On A GPU," IEEE Transactions on Computers, vol.62, pp.1156-1169, 2013.

[13] C.-H. Lin, C.-H. Liu, L.-S. Chien, and S.C. Chang, "Accelerating Pattern Matching Using a Novel Parallel Algorithm on GPUs," IEEE Transactions on Computers, 62 (10), pp.1906-1916, 2013.

 ISBN: 978-1-61208-503-6