JAIST Repository
https://dspace.jaist.ac.jp/

Title	Accelerating bit-based finite automaton on a GPGPU device
Author(s)	Vu, Kien Chi
Citation	
Issue Date	2020-09
Type	Thesis or Dissertation
Text version	author
URL	http://hdl.handle.net/10119/16859
Rights	
Description	Supervisor: 井口 寧, 先端科学技術研究科, 修士(情報科学)

JAIST
JAPAN ADVANCED INSTITUTE OF SCIENCE AND TECHNOLOGY

Japan Advanced Institute of Science and Technology

Master's Thesis

Accelerating bit-based finite automaton on a GPGPU device

Vu Chi Kien

Supervisor: Prof. Yasushi Inoguchi

School of Information Science
Japan Advanced Institute of Science and Technology
(Master of Science)

September, 2020

Abstract

In the Internet era, the amount of data over networks is growing rapidly day by day, and people are getting easier to access information and knowledge. On the other hand, a number of cybercrimes are on the rise due to the availability of the data. Criminals try attacking network systems to gain unauthorized information. Therefore, a critical problem is how to secure the data on the Internet. The most common security mechanism is a firewall, which is used to control network traffic by defining the rules and filtering data packets based on those rules. However, the growth of sophisticated attacks requires complicated rulesets to prevent, which will slow down the filtering process, causes the bandwidth bottleneck, and decrease the performance of the network system. BFA is a research, which tried to address that problem by introducing the regular expression matching algorithm to take advantage of parallel computations on multi-core platforms. Unfortunately, the task of each parallel process is quite hard, so the method is not suitable to use with large rulesets. In our research, we focus on improving the BFA method by proposing enhancement of the matching algorithm to utilize parallel processing by using Single Instruction, Multiple Thread model on a GPGPU device. Our method implemented on GPGPU reduces matching time dramatically 50 to 215X compared with the previous one on the host, depending on the complicated level of the ruleset.

Contents

1 Introduction **1**
- 1.1 Problem statements 1
- 1.2 Research objective 1
- 1.3 Approach 2
- 1.4 Goal of the research 2
- 1.5 Research Scope 3
- 1.6 Thesis Organization 3

2 Background **4**
- 2.1 Introduction 4
- 2.2 Regular expression matching 4
  - 2.2.1 Regular expression 4
  - 2.2.2 Non-deterministic finite automaton 5
- 2.3 GPGPU Architecture 7
- 2.4 Summary 9

3 Related Works **10**
- 3.1 Introduction 10
- 3.2 NFA pattern matching on multi-core processors 10
  - 3.2.1 SR-NFA on Multi-core systems 10
  - 3.2.2 BFA - bit-based finite automation 11
- 3.3 NFA pattern matching on GPUs 13
  - 3.3.1 iNFAnt - NFA implemented on GPGPUs 13
  - 3.3.2 iNFAnt2 14
  - 3.3.3 Virtual-NFA - GPU-based NFA implementation 15
  - 3.3.4 Throughput optimizations of NFA processing on GPUs 16
- 3.4 Summary 17

4 Proposed Method and Implementation **18**
- 4.1 Introduction 18
- 4.2 Problems of the original work 18

4.3 The modification . . . 21
4.4 Complexity analysis . . . 23
4.5 Implementation . . . 24
4.5.1 Pre-processing phase . . . 25
4.5.2 Matching phase . . . 26
4.6 Summary . . . 26

5 Evaluation **27**
5.1 Introduction . . . 27
5.2 Experimental environment . . . 27
5.3 Comparing with the original BFA method . . . 28
5.3.1 Experiment results . . . 28
5.3.2 Comparing theoretical and experimental performance . 29
5.4 Comparing with iNFAnt2 . . . 31
5.5 Summary . . . 35

6 Conclusion and future work **36**
6.1 Conclusion . . . 36
6.2 Future work . . . 36

List of Figures

2.1 Simple state machine model of an oven 5
2.2 A NFA accepting regular expressions of *abcd+*, *ab[c-z]e+*, *bc+da* 7
2.3 Pascal GP100 Streaming Multiprocessor (SM) unit architecture (quoted from [1]) 8
2.4 Logical GPGPU architecture (quoted from [2]) 9

3.1 An instance of bit-based finite automaton (BFA) machine for "ab.*cd" (quoted from [3]) 11
3.2 Example of BFA transition after receving "c" character 12
3.3 Symbol-first representation (quoted from [4]) 13
3.4 State transition algorithm 14

4.1 Processing flow of BFA 19
4.2 Modification to processing flow 21
4.3 Modified parallelize BFA algorithm on a GPGPU device . . . 22
4.4 An overview of the matching system 25

5.1 The change in speedup ratio over number of states 29
5.2 The change in matching time over number of states 30
5.3 Comparison between iNFAnt2 and Modified BFA using GPGPU device 32
5.4 Comparison between iNFAnt2 and Modified BFA using GPGPU device (Cont.) 33
5.5 The change in matching time over number of states 34

List of Tables

2.1 Common metacharacters are used in regular expression 5

5.1 The detail of rulesets used for the evaluation 27
5.2 Matching time comparison with the original method on CPU . 28
5.3 Matching time comparison with the original method on GPGPU 29

List of abbreviations

CPU central processing unit

GPU graphics processing unit

GPGPU general-purpose computing on graphics processing unit

NFA nondeterministic finite automaton

BFA bit-based finite automaton

CUDA Compute Unified Device Architecture

SM Streaming Multiprocessor

Chapter 1

Introduction

1.1 Problem statements

Regular expression matching is a technique that can detect the same language expressions using given patterns. Firewalls use that technique to determine data packets over the network, whether it contains malicious code or not. However, with the development of technology, there are more and more sophisticated attacks. Therefore, the matching system requires a large number of patterns to check data. As a consequence, since the firewall needs to check many patterns, it takes up much storage space and computational resources. A bottleneck may occurs and causes the decreasing of the system performance.

There is a research named bit-based finite automaton (BFA) [3], which used the nondeterministic finite automaton (NFA) to match strings. BFA can handle complex rulesets and state transitions by taking advantage of parallel computation on multi-core platform.

The weakness of BFA is that when the ruleset is getting more complex, the task of each central processing unit (CPU) core will be heavy and make the matching process become slower.

This research focuses on solve that problem of BFA.

1.2 Research objective

We are trying to increase the throughput of matching engine by researching some studies which are good at making use of parallel computation. BFA is our choice since it shows some potential for increasing throughput of matching engine. The previous section shows the problem of BFA, which is the task explosion for each core if the ruleset is more complicated. The size of

state matrices will be huge when the number of states increases. Using a core to perform many matrix multiplications is not suitable.

In this research, we aim to reduce the matching time by split and balance the work of each core, increase the performance of the matrix multiplication process.

1.3 Approach

Our approach is moving the matching phase from CPU to general-purpose computing on graphics processing unit (GPGPU). Since CPU and GPGPU are independent of each other, the filtering packet process will not block other processes on the CPU, and it will help increase the performance of the network system. The CPU will receive packets from the Internet and send them to GPGPU, GPGPU performs the matching phase and send the results back to CPU. In the end, CPU will decide to drop or forward the packets.

Besides that, we improve the matching algorithm to be suitable with GPGPU architecture, which has a large number of cores but the clock speed of each core is very low. The enhanced algorithm split the matrix multiplication process into many tiny independent tasks that fit processing units on the GPGPU device to utilize parallel processing.

1.4 Goal of the research

The Internet network speed has been increase significantly over time from 10 Mbps to 100 Gbps. Therefore, in order to protect a network system, the ideal firewall need to filter the packet in the same speed with the network speed. However, the filtering packet process speed depends on the demand for security level. For example, a home network will use a simple ruleset to filter. On the other hand, a business network need to use complex ruleset to protect the network. The previous survey [5] showed that with the ruleset containing 7,700 states, the throughput of regular expression matching engine using NFA on GPUs is under 50 Mbps (0.9 Mbps on CPUs). According to the result, most of the matching engines are not suitable for applying to reality. We hope that our study can break the limit of previous studies, so that the matching engine using regular expression can be put into practice.

1.5 Research Scope

Our main target focuses on improving non-deterministic finite automaton representation for being suitable with the parallel architecture of GPGPU devices, which can handle a large amount of packets data using complex rulesets in order to perform regular expression matching.

1.6 Thesis Organization

This section provides an overview of our thesis structure.

In this chapter, we introduce our research, which consists of the problems we aim to solve, the scope of the study. Furthermore, we brief on the objectives and our approach to deal with the issues.

In Chapter 2, we show the basic knowledge about the techniques and devices which we use in this research.

Chapter 3 mentions some related works which implemented NFA machine on GPGPU devices to examine network packets. We use them as competitors with our work. The last research in this chapter, which is bit-based finite automaton [3], is a foundation for our study.

In Chapter 4, we go into detail about the problems of the original work and our solution to increasing the performance of the system. After that, the analysis and implementation will be shown to explain our work.

Chapter 5 describes the environment which we use to implement and evaluate our study. Besides that, we show the comparison results with the competitor and explain that results.

Finally, we show our achievements in the summary part and a plan for future work in the last chapter.

Chapter 2

Background

2.1 Introduction

Regular expression matching is a basic mechanism of deep packet inspection. It is used to examine network packets transferring over the Internet in order to avoid cyber attacks. Non-deterministic finite automaton is a memory efficient method to implement regular expression matching due to the flexibility of the machine. However, that characteristic brings significant challenges to have a good matching engine implementation on GPGPU since the divergences occur when a machine has many transitions triggered by an input. Furthermore, since the machine has to check the pattern storing in memory, memory accessing is an important matter which we need to consider. Base on the survey [5], the matching performance of the engine implemented on GPGPU when using large complex ruleset does not exceed 0.3 Gbps.

2.2 Regular expression matching

2.2.1 Regular expression

Regular expression was defined in [6], which illustrates strings or ordered pairs of strings. They are often used to represent patterns for matching text by using a sequence of symbols. Each symbol has its literally or special meaning.

Table 2.1 shows common metacharacters that often be used in the regular expression and corresponding meanings. Metacharacters make the regular expression more flexible since a metacharacter has a specific meaning and can match with one or more characters. For example, "[a-e]" match all lower case letters from "a" to "e".

Table 2.1: Common metacharacters are used in regular expression

Type	Metacharacter	Description	Example
Character class	.	matches any single character	
	[]	matches a single character contained in the brackets	$[abc]$ matches a or b or c
	[^]	matches a single character not contained in the brackets	$[\hat{}ab]$ matches any character other than a or b
Quantification	$*$	matches the preceding element zero or more times	$a * b$ matches b, ab, aab, etc
	?	matches the preceding element zero or one time	$ab?c$ matches ac and abc
	$+$	matches the preceding element one or more times	$a + b$ matches ab, aab, $aaab$, etc
	$\{m, n\}$	matches the preceding element from m to n times	$a\{1, 3\}b$ matches ab, aab and $aaab$
	$\{m, \}$	matches the preceding element at least m times	$a\{1, \}b$ matches ab, aab, $aaab$, etc
	$\{m, n\}$	matches the preceding element at most n times	$a\{, 3\}b$ only matches b, ab, aab and $aaab$
Position	^	matches the starting position within the string or line	$\hat{}hat$ matches hat but only at the start of the string or line
	$	matches the ending position of the string or line	$cat\$$ matches cat but only at the end of the string or line
Others	$\vert$	OR relationship	$abc\vert def$ matches abc or def

In the network field, a regular expression can be used to represent the unique characteristics of application protocols, malicious strings, data contained inside network packets. Therefore, regular expression matching is widely used as firewall mechanism, specially deep packet inspection engines and applications.
For instance, a regular expression "[\x22 \x27] \s*or \s* \d+ \x3d" will filter the packet containing a SQL injection pattern like "$'or\ 1 = 1$".

2.2.2 Non-deterministic finite automaton

A finite automaton [7] is a computational model consisting of a finite number of states and transitions between pairs of states. A transition represents a change from one state to another in response to some conditions.

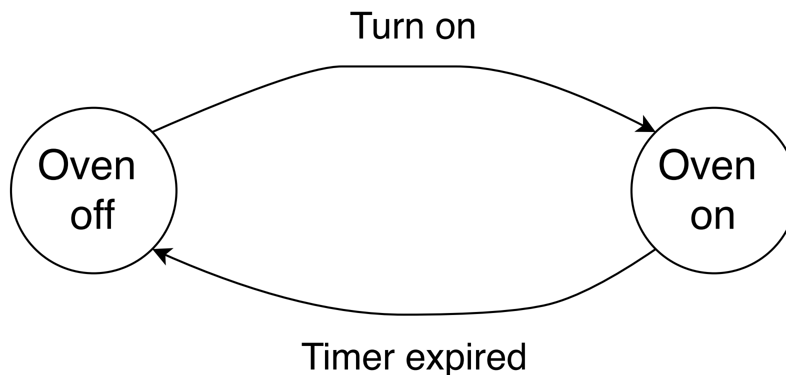


Figure 2.1: Simple state machine model of an oven

Figure 2.1 shows an instance of a state machine model, when turning on the oven, it will change from *off* state to *on* state. That is a transition caused

by a condition "turn on". When the timer expired, the state of the oven will change from *on* to *off*.

There are two kinds of finite automaton, which are nondeterministic finite machine and deterministic finite machine. A deterministic finite machine is a machine that can determine exactly a destination state from a source state and a condition. On the other hand, in a non-deterministic finite automaton model, when a condition triggers transitions, a source state can move to none, one or many destination states. Our research focuses on solving the problem of the non-deterministic computation model.

As the definition from [8], NFA is a 5-tuple $M = (Q, \Sigma, \delta, q, F)$, where

- a set of states Q
- a set of symbols Σ
- a transition function $\delta : Q \times \Sigma \rightarrow P(Q)$, $P(Q)$ is a notation for the power set of Q [1]
- a start (or initial) state q, which is an element of Q
- a set of accept states F (a subset of Q)

NFA has a further generalization, called NFA with ε-moves (NFA-ε), which accepts an empty string ε as valid input. However, since NFA-ε is equivalent to NFA (NFA-ε can transform to NFA, and vice versa [9]), we only work with NFA in this research.

The nondeterministic finite automaton M can accept string s if for any symbol in s, M can move to the states which belong to the set of accept states F If the above conditions are not satisfied, M rejects the string s.

The figure 2.2 shows an instance of NFA, which is created from three regular expression rules. If "a" is an input symbol, it can trigger changes: (0) *to* $(0), (0)$ *to* $(1), (0)$ *to* $(2), (11)$ *to* (14). If the machine reaches an accept state such as $(10), (11), (12)$, a match is found between a input string and a pattern.

[1] The power set of Q is set of all subsets of Q

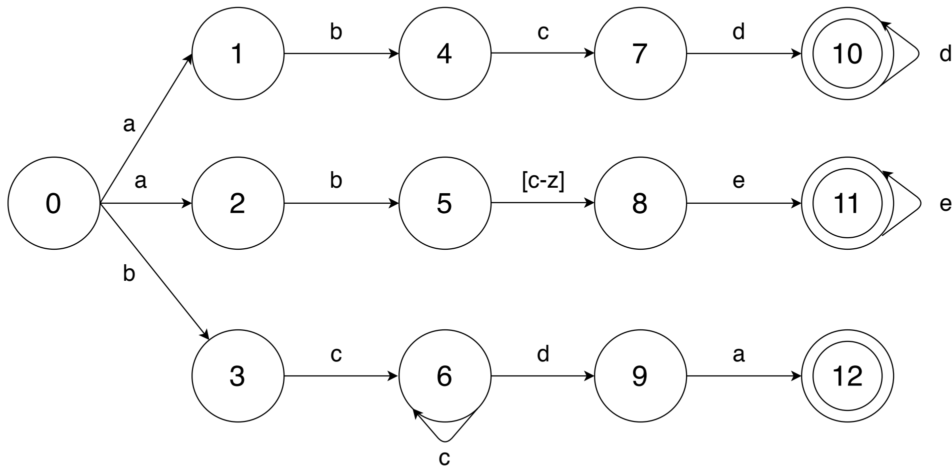


Figure 2.2: A NFA accepting regular expressions of *abcd+*, *ab[c-z]e+*, *bc+da*

2.3 GPGPU Architecture

A graphics processing unit (GPU) is a particular electronic circuit, which is designed for accelerating the creation of images in a frame buffer that images are used for output to display equipment such as a monitor, mobile phone's screen,... In order to deal with enormous elements of graphic images, GPU uses a large number of cores working at the same time for computation and rendering. Its parallel architecture helps GPU more useful than the CPU.

CPU focuses on reducing the latency when process a task, so CPU uses a small number of cores that have a high clock speed. On the other hand, GPU uses a huge number of low clock speed cores in order to process many tasks in parallel, which will increase the throughput of the system.

Ordinarily, the GPU aims at rendering graphics; a specialized GPU was created to handle tasks that were formerly the domain of CPU, named GPGPU. GPGPU uses Single instruction, multiple thread (SIMT) computation model, which means an instruction can run by many threads at the same time. This model is beneficial for data parallelism, which is separating data into many small parts and process them using the same set of instructions.

Details of the physical architecture of a GPGPU instance are shown in figure 2.3. In this GP100 instance, there are 60 Streaming Multiprocessor (SM) containing many computational resources. A steaming multiprocessor can handle many tasks using multi-warp[2]. Each warp contains scheduler,

[2]Warp is a set of 32 threads

instruction buffer, register file, computation units; therefore they can work independently with each other. Warp scheduler manages all the tasks of a warp in order to make a schedule for multi-thread. Hence, the number of parallel flows which a device can handle is equal with the number of its warps.

Figure 2.3: Pascal GP100 SM unit architecture (quoted from [1])

There are many software platforms that are used to program for GPGPUs such as Nvidia Compute Unified Device Architecture (CUDA) (specific for NVIDIA GPUs), OpenMP (Open Multi-Processing), OpenCL (Open Computing Language). In this research, we use CUDA to implement our program since it is optimized for Nvidia devices. Figure 2.4 shows a logical design example which user can use to interact with the hardware. A kernel is a function that GPGPU will process. Users can assign tasks to resources using "grid". A "grid" can divide into many blocks, and each block contains many threads. Depending on user needs, thread blocks can be created as 1D, 2D, or 3D logical grids. In the figure, the first kernel uses a 2 dimensions grid, which contains six blocks organized, each block has 15 threads. There is some logical type of memory:

- Global memory: the largest and slowest memory type which can be accessed by all of threads

- Constant memory and texture memory: specific memory types serve to store constant variables or read-only variables. They have smaller space but faster than global memory
- Shared memory: the highest speed shared memory between threads of a block, which has the smallest storage space
- Local memory: A memory space that resides in global memory, it exists only inside a thread view and cannot be accessed by other threads.

Since the GPGPU memory is independent with the host memory, data needs to be initialed and move from host to device before initialing a kernel. It will cost more additional time than only using the CPU.

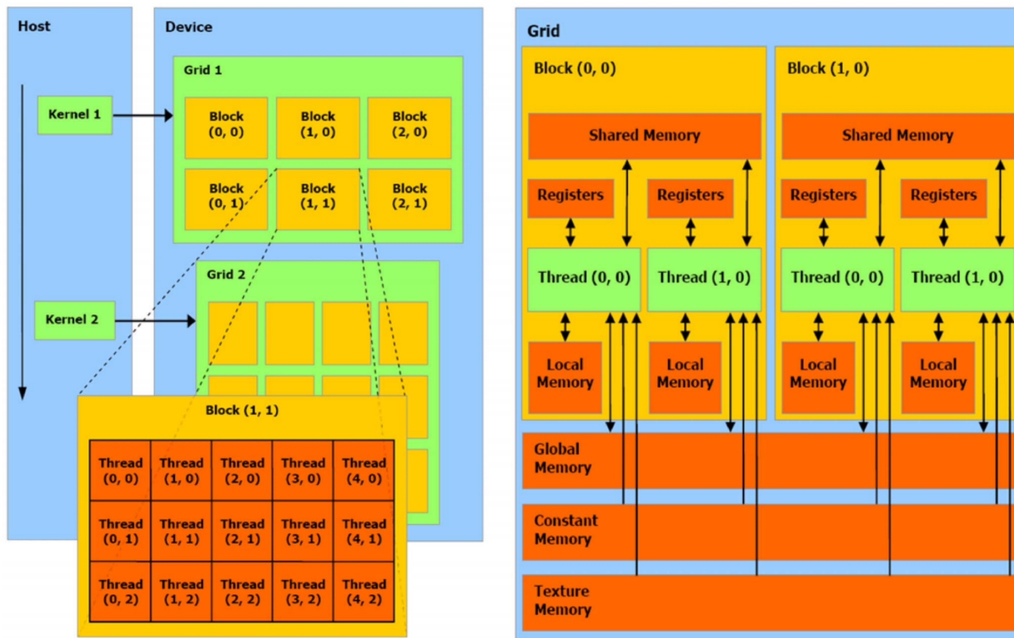


Figure 2.4: Logical GPGPU architecture (quoted from [2])

2.4 Summary

In this chapter, we introduced about some basic knowledge which we used in our study. It contains definition and application of the regular expression matching, or more specifically, the non-deterministic finite automaton. Besides that, we mentioned about the architecture of GPGPU, which is a device we used to implement and evaluate our research.

Chapter 3

Related Works

3.1 Introduction

In this chapter, we mention some well-known studies which have successfully increased performance of the matching engine using regular expression.There are many studies which have tried to address the problem of nondeterministic finite automaton in many ways such as parallel computation, representation format, $\cdots$. Each study has achieved its improvements.

3.2 NFA pattern matching on multi-core processors

3.2.1 SR-NFA on Multi-core systems

SR-NFA [10] is a study based on modular nondeterministic finite automaton [11]. The pattern is represented by two matrices. A matrix is used to represented ε-transitions between pairs of states. Another records the trigger symbols accepts by all of the states.

To handle complex ruleset, the authors separate a complicated SR-NFA into many simple SR-NFA segments. In addition, the ruleset is classified into two categories:

- A simple set: a set of rules which can be created deterministic finite automaton without state explosion
- A complex set: a set of rules which cannot be created deterministic finite automaton due to enormous number of states

Both sets have complied to many SR-NFA segments. Depend on an input symbol, a small number of segments corresponding to the type of the symbol will be used to check.

Using Snort ruleset containing 1134 rules, SR-NFA obtains around 2.2 Gbps throughput.

3.2.2 BFA - bit-based finite automation

This method was proposed by Zhe Fu et al. in 2019 [3], which is designed to accelerate the matching process of non-deterministic automaton on multi-core platform. All of state transitions in the NFA is represented by bit vectors. Then the state transition procedure transforms into a Boolean vector multiplies by Boolean matrices.

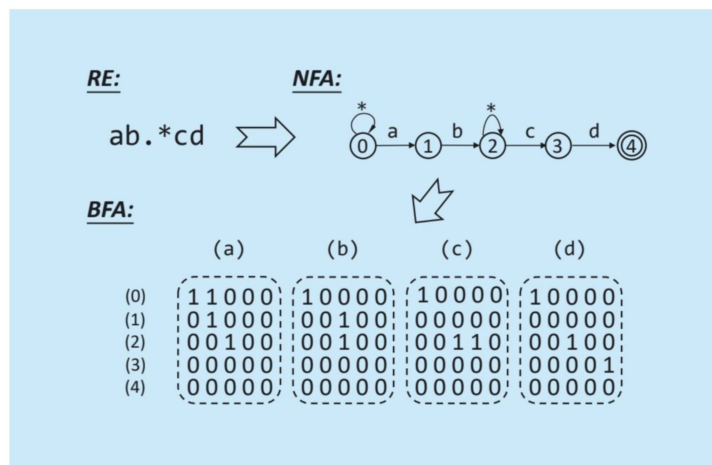


Figure 3.1: An instance of BFA machine for “ab.*cd” (quoted from [3])

After generating NFA graph from the regular expression, it will be encoded into $s \times s$ Boolean matrices (s is the number of states). A number of matrices is equal with the number of transition symbols, normally BFA will have 256 matrices corresponding to 256 ASCII characters. Each row in a matrix represents a state in NFA, and a transition will be represented by a bit 1. For example, in figure 3.1, “ab.*cd” has been transformed to NFA diagram, then it is encoded into 4 Boolean matrices corresponding to 4 trigger symbols (a, b, c, d). The machine starts from the state (0); if it receives character “a”, the machine can move to (1) and stay at (0), therefore the matrix “a” will have bit 1 at first and second positions, other positions will be 0.

$$\begin{bmatrix}1 & 0 & 1 & 0 & 0\end{bmatrix} \times \begin{bmatrix}1 & 0 & 0 & 0 & 0\\0 & 0 & 0 & 0 & 0\\0 & 0 & 1 & 1 & 0\\0 & 0 & 0 & 0 & 0\\0 & 0 & 0 & 0 & 0\end{bmatrix}$$
$$= \begin{bmatrix}1 & 0 & 1 & 1 & 0\end{bmatrix}$$

Figure 3.2: Example of BFA transition after receving “c” character

An instance of transition process is depicted in figure 3.2. For example, a current state vector is [10100] which means that states (0),(2) are activated. When the machine receives “c” as an input symbol, the new state vector will be calculated by taking the current state vector multiplies with Boolean matrix of “c”. The result is activated states triggered by “c”, which are (0),(2),(3).
The multiplication process can be accelerated using the algorithm from [12], which is scanning bit 1s in the current state vector (first position and third position in the figure), then choosing rows in the matrix corresponding to the positions of bit 1s (first row and third row) and “OR” them together. A result vector will be the result of the multiplication.

Algorithm 1 describes the state transition algorithm(multiply a vector by a matrix algorithm) that is used in BFA. The algorithm takes an idea from [12], which is a fast Boolean multiplication method. The input of the algorithm contains bit vector v_i, input character k, and transition tables B (normally 256 Boolean matrices corresponding to 256 ASCII characters). The first step is finding all the positions of bit 1 inside the bit vector v_i and store them in *set* variable. Then, a “for” loop is created to process all the position in *pos_set*, v_{i+1} is calculated by taking corresponding row in matrix of character k and perform AND operation with v_{i+1}, the final value of v_{i+1} will be the result of multiplication between vector v_i and matrix of character k.

Algorithm 1: Optimized vector multiplies with matrix

Input : bit vector v_i, input character k, transition tables B

Output: bit vector v_{i+1}

1 $pos_set \leftarrow$ FindBit1Position(v_i)

2 $v_{i+1} \leftarrow \{0\}$

3 **for *each*** pos ***in*** pos_set **do**

4 | $v_{i+1} \leftarrow v_{i+1} \vee B(pos, k)$

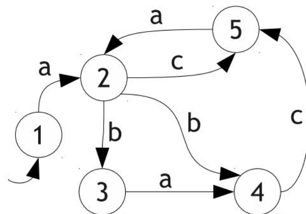
5 **end**

6 **return** v_{i+1}

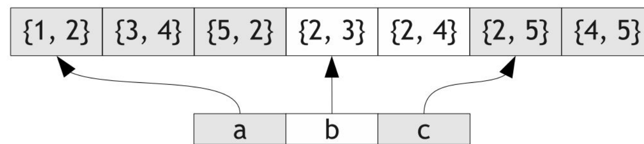
3.3 NFA pattern matching on GPUs

3.3.1 iNFAnt - NFA implemented on GPGPUs

Niccolo' Cascarano et al. proposed iNFAnt, an implementation of NFA on GPGPU devices to filter network packet. In this research, a transition (an edge of NFA graph) will be store in a pair "{source, destination}". The system store lists of pairs sorted by the trigger symbol. Figure 3.3 shows an example of iNFAnt representation, symbol "a" will trigger 3 transitions (1→2),(3→4),(5→2); therefore "a" points to a list containing 3 pairs (1,2),(3,4),(5,2).



(a) NFA transition graph.



(b) Transition vector.

Figure 3.3: Symbol-first representation (quoted from [4])

A current active state set and a future state set are represented by bit vectors stored in shared memory on the GPGPU device. Each bit denotes

```
1: current_sv ← initial_sv
2: while ¬input.empty do
3:    c ← input.first
4:    input ← input.tail
5:    future_sv ← current_sv ∧ persistent_sv
6:    while a transition on c is pending do
7:       src ← transition source
8:       dst ← transition destination
9:       if current_sv[src] is set then
10:         atomicSet(future_sv, dst)
11:      end if
12:   end while
13:   current_sv ← future_sv
14: end while
15: return current_sv
```

Figure 3.4: State transition algorithm

a state, and an active state will set as 1; otherwise, it will be 0. Figure 3.4 describe depict transition algorithm in detail. A "while" loop is created to process all the input character sequentially until "input" is empty. A persistent vector denotes all the persistent states which are not reset after activating. Line 6 to line 12 will be done by exploiting parallelism. When examining a character, each transition pair will be assigned to a thread. If a bit "source" is set in the current vector, then the bit "destination" will be set in the future vector using atomic operation[1] to avoid memory conflict. After examining all the transition pairs of a trigger symbol, the future state vector will be set as the new current vector for the next symbol. When the input stream is empty, the algorithm returns the current state vector as the final state of the machine.

3.3.2 iNFAnt2

iNFAnt2 which is an enhanced version of iNFAnt, is proposed by Jack Wadden et al. [13]. This version contains some modifications:

- Mark accept states using negative ID numbers.
- Can process multi-byte character set.

[1] An atomic operation guarantees that a race condition will not happen when many parallel threads try to read and write memory.

- Store lists of NFA transition in the GPGPU texture memory which is good for read-only data.

With the purpose of improve the performance of iNFAnt2, the authors used some new capabilities of instruction-level profiling of NVIDIA Maxwell architecture. Furthermore, iNFAnt2 can report the ID of the rule of each detected match.

Since this work is well-known research in regular expression matching implemented on GPGPU device field, we use it as the competitor with our work to evaluate the performance of NFA matching engine.

3.3.3 Virtual-NFA - GPU-based NFA implementation

Virtual-NFA [14] is a study of Yuan Zu et al. The authors detected a drawback of iNFAnt [4], which is that the number of transitions needed to be examined for one trigger symbol normally much larger than the number of states which actually activate. Hence, it has been shown that most of the threads will take responding for source states which are not activated. It wastes so many computational resources in the original iNFAnt design.

To reduce the number of wasted examination, the authors create an array to store all the active states at a time. The idea is that all each thread will examine an element of active state array, it can reduce wasted computation than examine all the transitions of the character. However, a race condition may occur when threads try to write the new active state into the active state array. To solve this issue, they defined a concept of *compatible groups* to store states of the machine, which satisfy that members belonging to one group cannot activate simultaneously. A compatible group takes responsibility for an active state. Therefore, when threads set new active states to the array, they will not access the same element since the new states cannot belong to the same group.

In reality, a number of compatible groups can be large than number of threads; therefore, the authors proposed *virtual-NFA*, which is transformed from an original NFA. All the states in virtual-NFA belong to super compatible groups, a super compatible group is created from many small compatible groups. As a consequence, some of the members in a super group can be activated simultaneously. Therefore they used the term *virtual states* to represent distinct statuses of the super group.

According to their experiment results, when using Snort rulesets containing the number of rules in a range from 16 to 36, the study can improve the matching speed by 29 to 46 times compared with the original iNFAnt.

3.3.4 Throughput optimizations of NFA processing on GPUs

Hongyuan Liu et al. have been noticed that the problems of NFA implementation on GPUs are the movement of data and compute utilization. Hence, their recent study [15] introduced some ways to improve the throughput of the NFA matching process using GPUs, which are shown below:

- Using new data structure to store NFA pattern. A state is represented by a node structure containing:
  - An 256-bit array of matchset, each bit will be an ASCII character, a bit is set if it triggers a transition.
  - Four outgoing edges in a 64-bit integer
  - 8-bit array of attributes: 3 bits are used to record start state, accept state and always-active state; other 2 bits are used for compression.

  This node data structure is mapping one-one to threads. Each thread will iterate all the input symbols; if there is any match with its structure, it will write the active state into a shared array variable to inform other threads in the same thread block.

- Compressing matchsets (reducing number of checking the array of trigger symbols): when the arrays have special attributes such as containing a continuous set of bit 1s or continuous set of bit 0s; such set will be marked by the first element and the last element. When a thread examines a matchset which has that attribute, it can examine in that range instead of checking all the bits.

- Since it is challenging to have enough threads for assigning node data structure, the authors examine the behavior of states and determine which states have high activity frequency and which states have low activity frequency. Based on that behaviour, high-frequency states will be mapped one-one to threads, the low-frequency states will be stored in a list, and a thread takes responsibility for one or many elements in the list (depending on the available computational resource).

Overall, this study archived better performance compared with iNFAnt [4] (around 23.5 times speed up) and Virtual-NFA [14] (around 5 times speed up).

3.4 Summary

This chapter discussed about some previou studies which tried to address the problem of nondeterministic finite automaton. According to the experiment results, we can see the improvements made by the above mentioned studies. They took advantages of parallel computation, as well as some special characteristics of nondeterministic finite automaton to increase the matching performance. Their proposed methods can boost the throughput of matching engine to 50 Mbps on GPUs (0.9 Mbps on CPUs) .However, these results are not good enough to implement into practice. Therefore, we research to find another way to boost regular expression matching process. We realized that BFA has some potential to improve the performance of matching system. Therefore, we proposed the modification and discuss it in the next chapter.

Chapter 4

Proposed Method and Implementation

4.1 Introduction

In this chapter, we present the details of problem that the original study has. After that, we show our modification to address that issue and how we implement the solution on a GPGPGU device. Besides that, we analyze the complexity of both original and modified algorithms to verify the truth of the solution.

4.2 Problems of the original work

Figure 4.1 shows the processing flow of the BFA method. The algorithm declares a initial vector v_0, $c_0, c_1, c_2, \cdots, c_{z-1}$ are denoted the input characters. After receiving the input, they will be separated into many small parts corresponding to the number of cores n on multi-core platform. Each core will calculate $\frac{z}{n}$ matrix multiplications in sequence. When all of the cores finish computing the multiplications, n result matrices are collected, and a core will perform the multiplications between the initial vector v_0 and the matrices. Since matrix multiplication does not have the commutative property, all of them will be calculated in order of the input stream.

The detail of the algorithm is shown in algorithm 2. The algorithm inputs are a BFA tuple $B = (\mathcal{Q}, \Sigma, \mathcal{B}, v_0, v_f)$ and input stream. The input stream C will be separated into n parts. From line 2 to line 7, n "for" loops are created to compute the matrix multiplications in parallel. n cores will perform matrix multiplications at the same time, M_i are temporary variables initialized as identity matrices to store multiplications results. From line 9

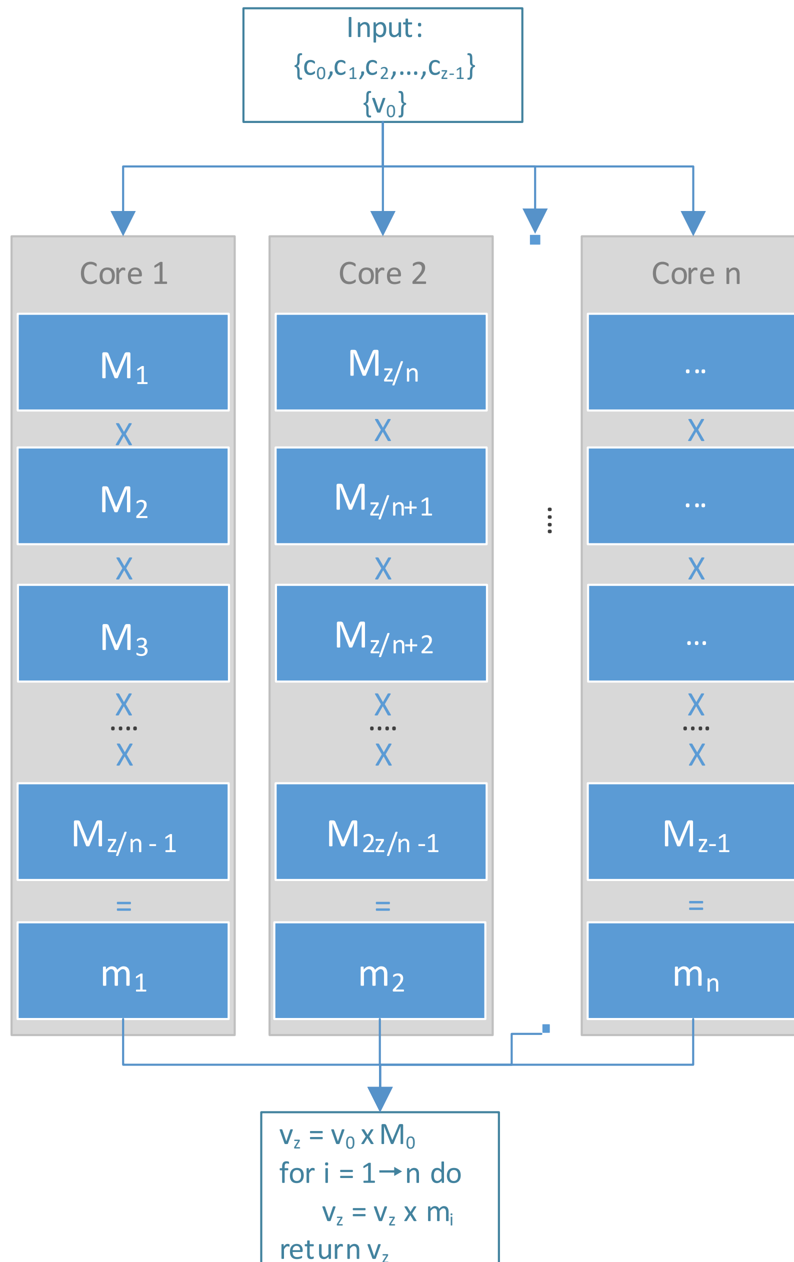


Figure 4.1: Processing flow of BFA

Algorithm 2: Parallel state transition on multi-core platform

Input : BFA B = Q, Σ, B, v_0, v_f, input data C = $c_0c_1c_2...c_{z-1}$, number of threads n

Output: bit vector v_z

```
1  //perform optimized BMM on each core
2  for each i in [1 ··· n] parallel do
3  |   m_i ← I_{s×s} //initialize m_i as an identity matrix
4  |   index ← z/n × (i − 1) //index of the first symbol for a thread
5  |   for each k in [0 ··· z/n] do
6  |   |   m_i ← m_i × M_{index+k}
7  |   end
8  end
9  //get the final v_z
10 v_z ← v_0 × M_0
11 for each i in [1 ··· n] do
12 |   v_z ← v_z × m_i
13 end
14 return v_z
```

to line 11, the initial vector v_0 will multiply with each matrix M_i and store in the variable v_n. After that, the algorithm returns the final state vector v_n, which describes the state of the machine after receiving all the characters.

The matrix multiplication, which is described in Algorithm 3, is performed by iterating over all of the rows in the first matrix m . For each *row*, the algorithm 1 will be use to calculate multiplication between a temporary vector *tmp* and the second matrix M_1. Each multiplication result vector will be a row in the result matrix m.

However, we notice that when the size of ruleset increases (denoted by s), the size of the matrix increase in exponential function (s^2). It causes slow multiplication time since the complexity of matrix multiplication is $\mathcal{O}(s^3)$.

Algorithm 3: Matrix multiplication

```
   Input  : matrices m, M_1
   Output: matrix m
 1 for each row in m do
 2     pos_set ← FindBit1Position(row)
 3     tmp ← {0}
 4     for each pos in pos_set do
 5         tmp ← M_1[pos] ∨ tmp
 6     end
 7     m[row] ← tmp
 8 end
 9 return m
```

4.3 The modification

Figure 4.2: Modification to processing flow

We realize that the Boolean matrix multiplication can be separated into many independent tasks. Figure 4.2 shows the processing flow for the transitions phase. Matrix C_0,matrix C_1,$\cdots$,matrix C_z denote the Boolean matrices of input characters $C_0, C_1, \cdots, C_z$. An initial vector v_0 multiplies with each input character's matrix sequentially, and a multiplication can be calculated in parallel by many threads.

To be more convenient for computation, we cast the Boolean matrices to the integer matrices. Figure 4.3 describes the detail of our modified algorithm. We create n threads, which equal to the number of integer elements in the state vector. A "for" loop is used to process the input stream (line 1). Each element will be assigned to a thread. Firstly, all the threads will examine their number to find positions of bit 1 inside an initial state vector and store them in a shared variable *pos_set* at line 2. After that, the machine takes the row row_{pos} in a matrix corresponding to each position in the *pos_set* and performing the "OR" operation between the row and a temporary state vector *temp*. The final value of the temporary state vector will be the initial state vector for the next input character. We use *syncThreads()* function on line 3 and line 8 to avoid conflict when accessing shared variables.

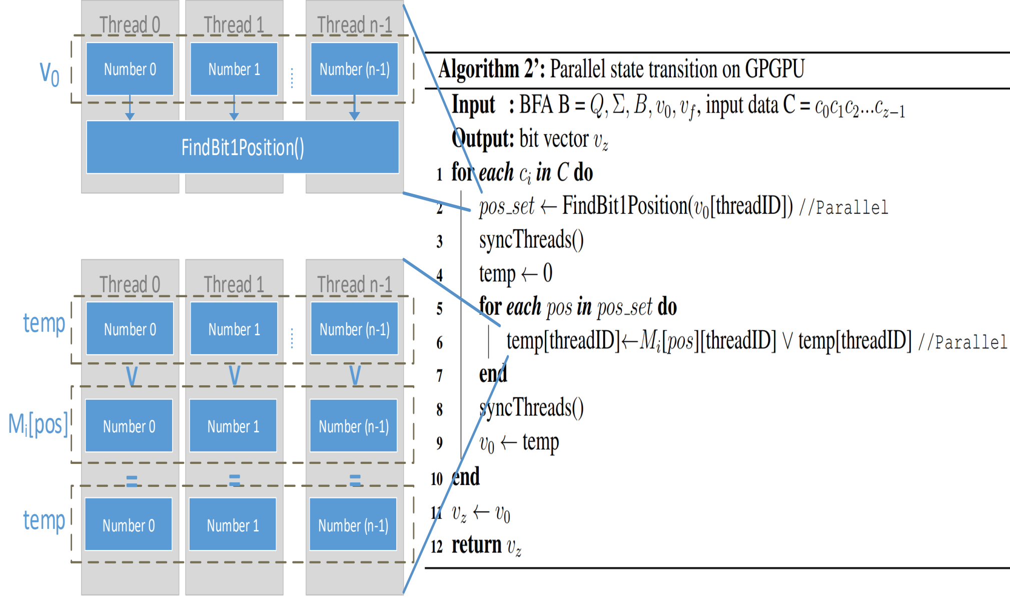


Figure 4.3: Modified parallelize BFA algorithm on a GPGPU device

The algorithm 4 shows details of how to find positions of bit 1s in an integer. Input is a number, and output is a set of positions where bits set to 1. We use the GPGPU device function "__ffs" to find the position of the least significant bit 1. The "while" loop in line 3 is used to examine all the bit of the number. After founding the first position, we keep it in a temporary variable tmp and perform shift right operation to remove the first bit 1 from the number. The shared variable $index$ is used to synchronize memory access between threads; when a thread found one position, it will store the position $tmp - 1$ at $index^{th}$ element of pos_set, then $index$ is increased by 1 using "atomicAdd" function, which is a CUDA device function used to guarantee that there is only one thread can modify memory content at a time.

Algorithm 4: Finding bit 1 positions

```
1  !htbp   Input  : integer number num
           Output: position set pos_set
2  int pos, index ← 0
3  int tmp ← 0
4  while num != 0 do
5  |   \\Find the position of the least significant bit set to 1
6  |   pos ← __ffs(num)
7  |   tmp ← pos + tmp
8  |   num ← num >> pos
9  |   atomicAdd(&index, 1)
10 |   pos_set[index - 1] ← tmp - 1
11 end
12 return pos_set
```

4.4 Complexity analysis

In this section, we will show the proof that our work is better than the original one.

We denote:

- s: number of states
- n: number of threads
- z: number of characters which need to be examined

Both the original work and our method use the result from [12], which proved that product of two $s \times s$ Boolean matrices uses an expected number of operations of $\mathcal{O}(s^2)$. For the worst case, it will need cs^3 operations (c is a constant), however it rarely occurs. Therefore, $s \times s$ Boolean matrix multiplication can be calculated using $\mathcal{O}(s^{2+\varepsilon})$ elementary operations[1] for any $\varepsilon > 0$, asymptotically in s.

Firstly, we analyze the original BFA method. A Boolean matrix size is s^2, hence the running time which needs to compute a matrix multiplication is $\mathcal{O}(s^{2+\varepsilon})$ with ($\varepsilon > 0$). There are z characters, then the complexity is $\mathcal{O}(s^{2+\varepsilon} \times z)$. n threads perform the multiplications in parallel. Therefore the running time of the first phase, in theory, will be:

$$\mathcal{O}(\frac{s^{2+\varepsilon} \times z}{n})$$

[1] Elementary operations are comparisons, additions, multiplications.

After performing all the matrix multiplications, n results will be multiplied by a state vector. Each vector-matrix multiplication needs $\mathcal{O}(s^{1+\varepsilon})$ (inheriting from matrix multiplication result), then the complexity is $\mathcal{O}(s^{1+\varepsilon} \times n)$. Therefore, the running time of BFA method is:

$$\mathcal{O}(\frac{s^{2+\varepsilon} \times z}{n} + s^{1+\varepsilon} \times n) \qquad with\, \varepsilon > 0 \tag{4.1}$$

Secondly, we consider the modified BFA method. Since we only use the multiplications between a vector and a matrix, it takes $\mathcal{O}(s^{1+\varepsilon} \times z)$ running time to transition z characters. However, as the parallel calculation of n threads, the complexity of whole process will decrease:

$$\mathcal{O}(\frac{s^{1+\varepsilon} \times z}{n}) \qquad with\, \varepsilon > 0 \tag{4.2}$$

From equation (4.1) and equation (4.2), it is clear that the running time of our modified BFA method is shorter than the original one when s, z, n are increasing. Using those equations, we calculate the approximate speedup ratio by equation (4.3)[2]. Algorithm complexity represents a rough number of instructions that need to execute a program based on the value of the input. Clock frequency shows the reverse of the time for a core executes one cycle. Therefore, the speedup ratio approximately equals to the ratio between both algorithm complexity times by the difference of clock frequency between CPU core and GPGPU core (some other factors will be omitted by the division).

$$\begin{aligned} Speedup\ ratio &= \frac{(4.1)}{(4.2)} \times \frac{Clock\ frequency\ of\ GPGPU\ core}{Clock\ frequency\ of\ CPU\ core} \\ &= \frac{\frac{s \times z}{n_{cpu}} + n_{cpu}}{\frac{z}{n_{gpgpu}}} \times \frac{Clock\ frequency\ of\ GPGPU\ core}{Clock\ frequency\ of\ CPU\ core} \end{aligned} \tag{4.3}$$

4.5 Implementation

As the equations from the previous section, we realize that when increasing the number of threads, the running time will reduce. That is the reason why we choose a GPGPU device, which can create many parallel threads to implement our method. The implementation is based on the open-source of the regular expression processor designed by M. Becchi [16].

[2]Since there are many factors that affect the performance of programs, the speedup ratio is for reference only

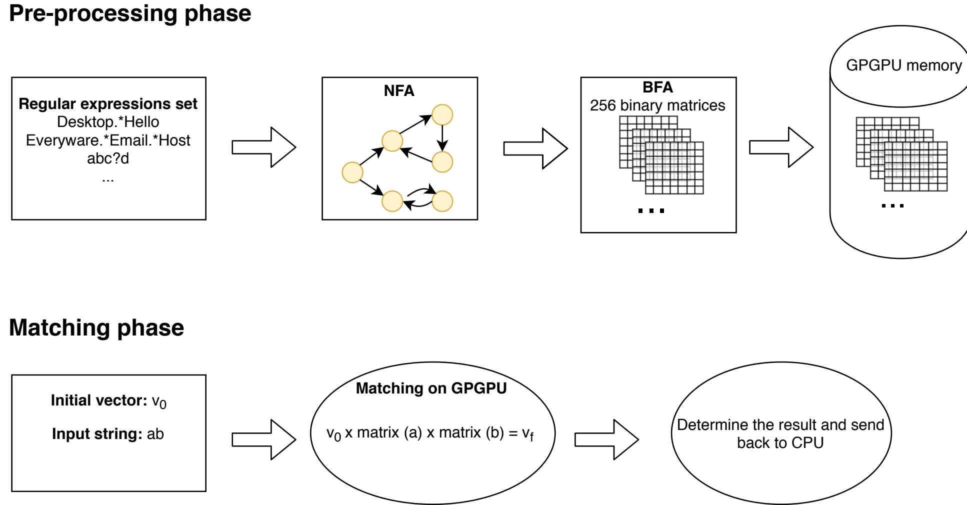


Figure 4.4: An overview of the matching system

Figure 4.4 shows two main phases in our system. A pre-processing phase is used to generate pattern structure from rulesets and store it on GPGPU device's memory; regular expression set will be used to generate NFA machine, then the machine is encoded to BFA form containing 256 binary matrices on CPU and BFA matrices will move to store in GPGPU memory in order to serve for matching processing. Another phase is the matching phase, which will match the input stream with a given pattern to detect malicious code.

4.5.1 Pre-processing phase

Regarding the pre-processing phase, here are the steps to prepare for the matching phase:

- Construct NFA from regular expression data sets: s = number of total states.
- Create transition square matrices size $(s \times s)$, each row of matrix is a bit vector.
- Padding *p bit 0s* to make size of each row is the blocks of *32 bits.*
- Cast each block becomes integer to create $\frac{s+p}{32}$ integer numbers.

- Transfer *256* integer matrices size ($s \times \frac{s+p}{32}$) to GPGPU memory.

4.5.2 Matching phase

In the matching phase, since the number of computation threads created by the GPGPU device is vast, GPGPU can process many packets at the same time. We decide to send packets in a batch to cut down the data transferring time between the host and the GPU device, and the number of thread blocks equals the number of packets in the batch, it means that packets and thread blocks are mapping 1 to 1. The input stream will be forwarded from CPU to GPGPU after the CPU receive data from the Internet. The GPGPU device will perform the matching process by calculating the state vector for each character and comparing the final state vector with the given accept vector to determine whether there is any match. After that, GPGPU sends results back to CPU and CPU will decide to forward or drop packets.
In order to reduce the global memory accessing time, we store state vector and indexes of the position set in the shared memory of GPGPU device, BFA pattern, and other variables are stored in global memory.

4.6 Summary

In this chapter, we already explained the problems of BFA on CPUs, which is assigning hard work for each core. The solution is reducing the task and making use of massive threads on a GPGPU device. The analysis has proved that our solution has a lower complexity than the original BFA. The implementation of both the pre-processing phase and the matching phase are also presented in this chapter. Besides that, based on the complexity, we show the equation for calculating the speedup ratio to determine the theoretical performance of the solution. The speedup ratio depends on the number of states in BFA and the device used to experiment. Based on the given rule-sets, we estimate that the speedup ratio ranges in 10 to 160. We will verify it in the next chapter.

Chapter 5

Evaluation

5.1 Introduction

In this chapter, we perform some experiments to evaluate our study. The environment used to set up will be shown. After that, we present the details of test cases for comparing with the original work and the competitor implemented on GPGPUs. The result and comparison will be analyzed.

5.2 Experimental environment

To evaluate our work, we set up our system on K20 server which has specification as follow:

- CPU: Intel(R) Xeon(R) CPU E5-2687W @ 3.10GHz
- RAM: 64 GB
- GPU: NVIDIA Tesla K20m-PCIE-5Gb (Base clock: 706MHz)

The rulesets used to test are Snort34 and Dotstar which we take from [3]. The complexity level depends on number of transition between states. The more transaction ruleset has, the more complex it is.

	Snort34[3]	**Dotstar**[3]
Number of states	883	11,191
Complexity	Low	High

Table 5.1: The detail of rulesets used for the evaluation

5.3 Comparing with the original BFA method

5.3.1 Experiment results

In order to compare with the original BFA method, we set up two test cases [17].

The first test is comparing with the original BFA implementation on CPU with our work on GPGPU. The input is a packet which has 16,000 bytes. The transfer time is determined by measuring the time from GPGPU device receives the input data until it sends the matching result back to CPU.

<table>
<tr><th></th><th>BFA (CPU)</th><th>Modified BFA (GPGPU)</th><th colspan="2" rowspan="2"></th></tr>
<tr><th>Transferring time (s)</th><td>0</td><td>0.001</td></tr>
<tr><th rowspan="4">Matching time (s)</th><td>0.033</td><td>0.041</td><td>883</td><th rowspan="4">Number of states</th></tr>
<tr><td>2.272</td><td>0.045</td><td>5181</td></tr>
<tr><td>5.160</td><td>0.047</td><td>8996</td></tr>
<tr><td>10.529</td><td>0.049</td><td>11194</td></tr>
</table>

Table 5.2: Matching time comparison with the original method on CPU

The result is shown on table 5.2. When using the ruleset containing 883 states, the speed of the modified BFA is slower than the original one since this ruleset is the simplest set. Although our work has an additional time which is the transfer time, the matching time is much better than the original one when using complex rulesets. The gap of matching time is more larger when the ruleset is more complex.

We measured the matching time in microsecond (μs). However, according to the experiment results, we realize that the error rate is high. The digit of millisecond (ms) is the same between several experimental runs. Therefore, to increase the precision of the result, we use second (s) as the unit of measurement with 3 digits after decimal point.

The second test is comparing both implementations on a GPGPU device. Since the original method is not suitable for GPGPU architecture, and we want to keep the basic idea of BFA, the test is set up simply with a small input data containing 64 bytes.
Table 5.3 shows the result of the test. Due to the low clock speed of GPGPU cores, when we force a core to compute a matrix multiplication, the processing time takes much longer than the modified one. The results are measured in millisecond to show the different between them.

		BFA	Modified BFA
Matching time	Snort34	871.192 ms	0.465 ms
	Dotstar	141,118.937 ms	1.046 ms

Table 5.3: Matching time comparison with the original method on GPGPU

5.3.2 Comparing theoretical and experimental performance

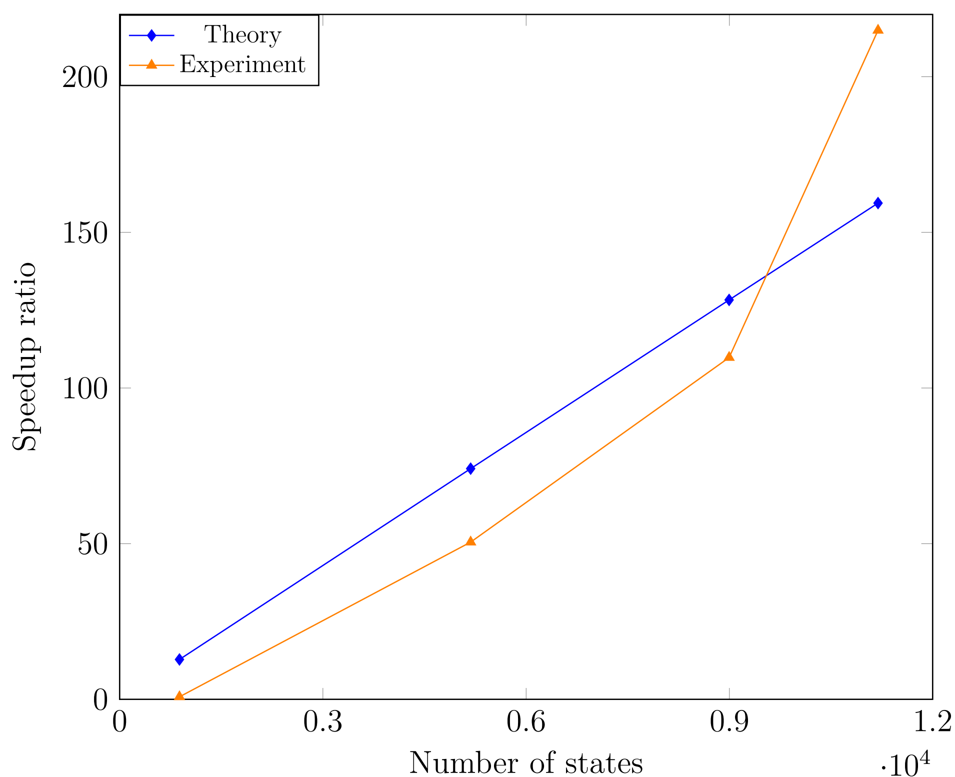


Figure 5.1: The change in speedup ratio over number of states

The figure 5.1 describe the different in speedup ratio between theoretical result calculated by equation (4.3) and experiment result. The number of CPU cores n_{cpu} is 16; the number of GPGPU cores n_{gpgpu} is 32 since although the number of threads used to process is large, they are grouped into many warps, all of the warps are executed by a SM using scheduling mechanism. The graph show the wide gap between theory and reality when changing number of states. The gap ranges from 1.4 to 16 times. There are many factors that affect the execution time of the program such as:

- Memory speed: the random-access memory speed is 1600 MT/s[1], and the speed of K20 memory is 5200 MT/s.
- Number of instructions: the complexity of a algorithm represent the difference in number of step to execute program based on the size of input. However, there are many instructions which do not depend on the size of input, they is not mentioned in the complexity equation. Therefore, the complexity cannot show exactly the difference between two algorithms.
- Built-in instructions: Built-in instructions is set of instructions that created for specific device to serve a purpose. In our work and previous work, we used some built-in instructions to process bits in integers. Hence the differences between CPU and GPGPU instruction set architecture cause the different in running time.
- Parallel model: Although the number of threads is given, it is not clear that how many threads run in parallel or they run in multi-threading, which are managed by schedulers.

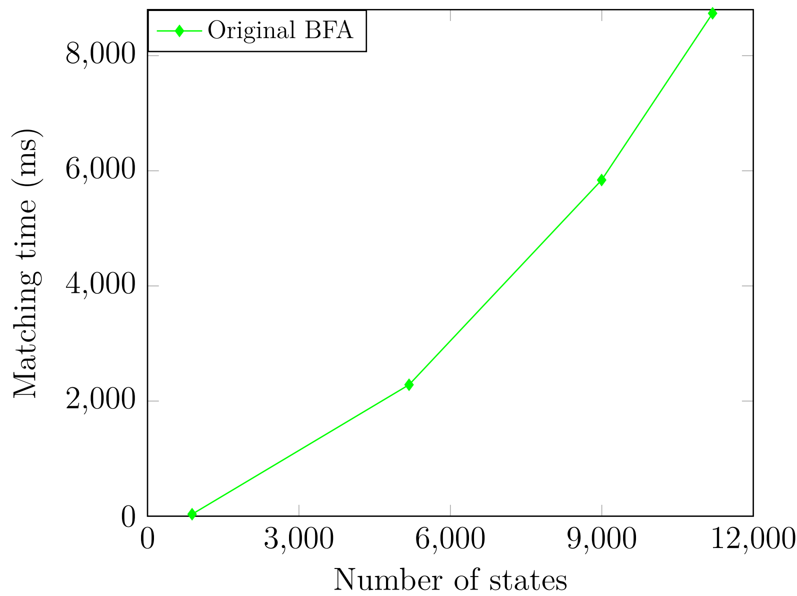


Figure 5.2: The change in matching time over number of states

We verify our complexity equation of the original BFA in chapter 4 by figure 5.2. In equation 4.1, it is easy to see that the complexity of running

[1]MT/s stands for megatransfers per second

time depends on the number of states that follow an exponential function. We vary the number of states from 883 to 11,191. The figure shows the matching time is larger after each tick; therefore, it follows an exponential function as the equation.

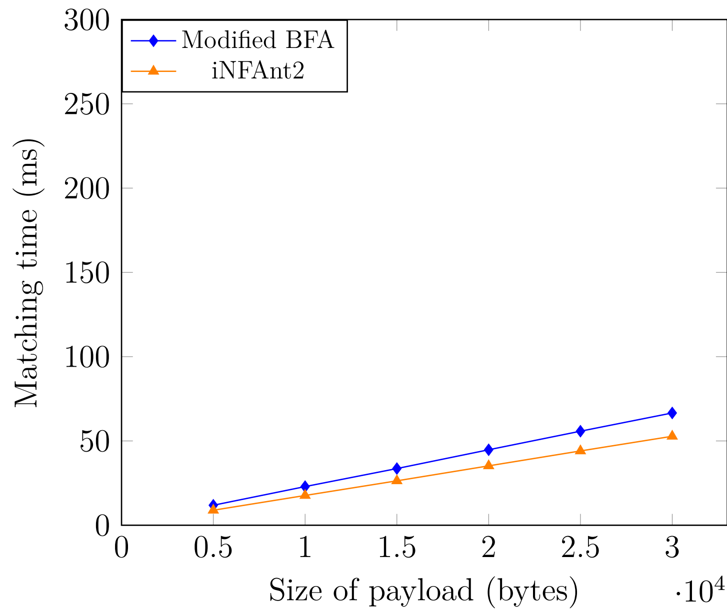
5.4 Comparing with iNFAnt2

iNFAnt2 is a well-known method used for regular expression matching on a GPGPU device. Therefore, we choose it as the competitor to our work. There are two test cases.

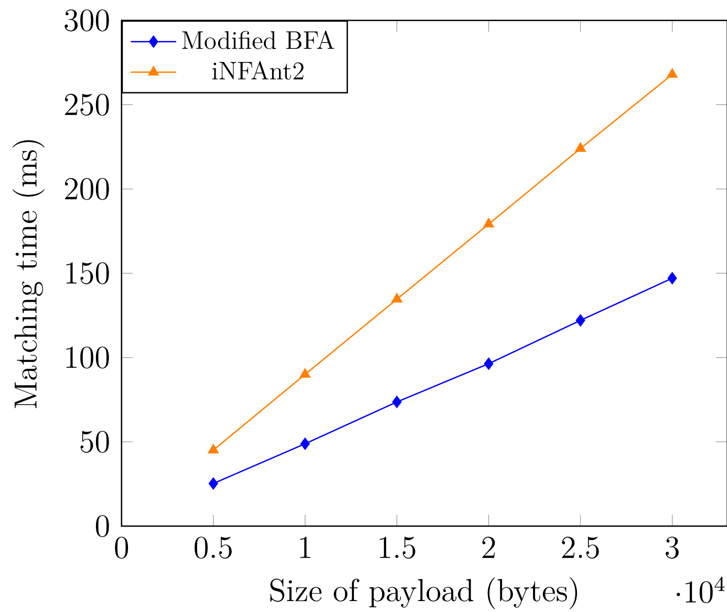
The first one is the matching time comparison when varying the size of the packet (input data for a thread block) from 5,000 *bytes* to 30,000 *bytes* per packet, the number of packets is 100. Figure 5.3 shows the experiment results with Snort34 and Dotstar rulesets. When using simple ruleset (Snort34), the matching time of iNFAnt2 is shorter than the modified BFA since the number of each character's transitions is small, the modified BFA will create fewer threads. As a consequence, the number of parallel computations will less than iNFAnt2. However, in the case of using Dotstar ruleset, which has high complexity, our work archive a significantly better result due to the task of each thread in iNFAnt2 is harder. The matching time of the modified BFA when using Dotstar ruleset is from 25.269ms to 147.062ms; iNFAnt2 is from 45.092ms to 267.949ms.

The second test case is varying the number of thread blocks (number of packets in a batch) when the size of the packet is fixed at 20,000 *bytes*. The result is depicted in figure 5.4. Similar to the first experiment, our work obtains a better result when using a high complexity pattern and a worse result when using the simple one. There are some periods in which the matching time does not change so much, due to the parallel computation threshold of the GPGPU device (K20). When the number of thread blocks is below the threshold, the device can process in parallel, so the time will not increase. After the number of thread blocks exceeds the device's threshold, some packets need to wait until enough computational resources are ready to be re-assigned; hence the processing time will increase.

To find the lower bound that our study will have better performance than the competitor, we use the experiment's results to draw figure 5.5. As depicted in the figure, approximately 4,000 states is a threshold which makes our work better in term of matching time.

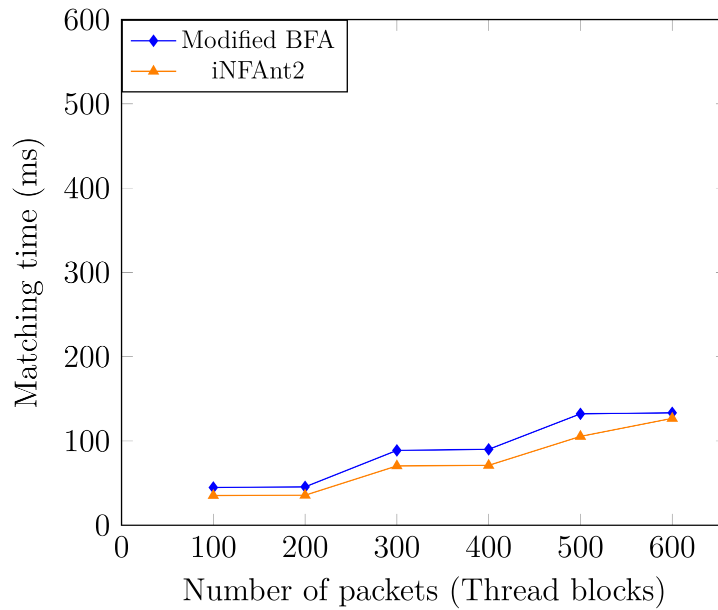


(a) Snort34 ruleset

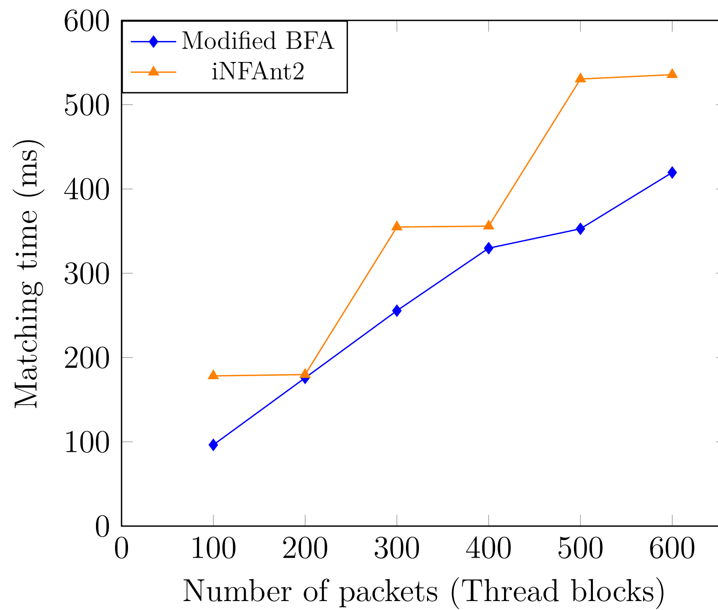


(b) Dotstar ruleset

Figure 5.3: Comparison between iNFAnt2 and Modified BFA using GPGPU device
The change in matching time over size of packet
Number of packets (Thread blocks): 100



(a) Snort34 ruleset



(b) Dotstar ruleset

Figure 5.4: Comparison between iNFAnt2 and Modified BFA using GPGPU device
The change in matching time over number of packets
Size of payload: 20,000 bytes

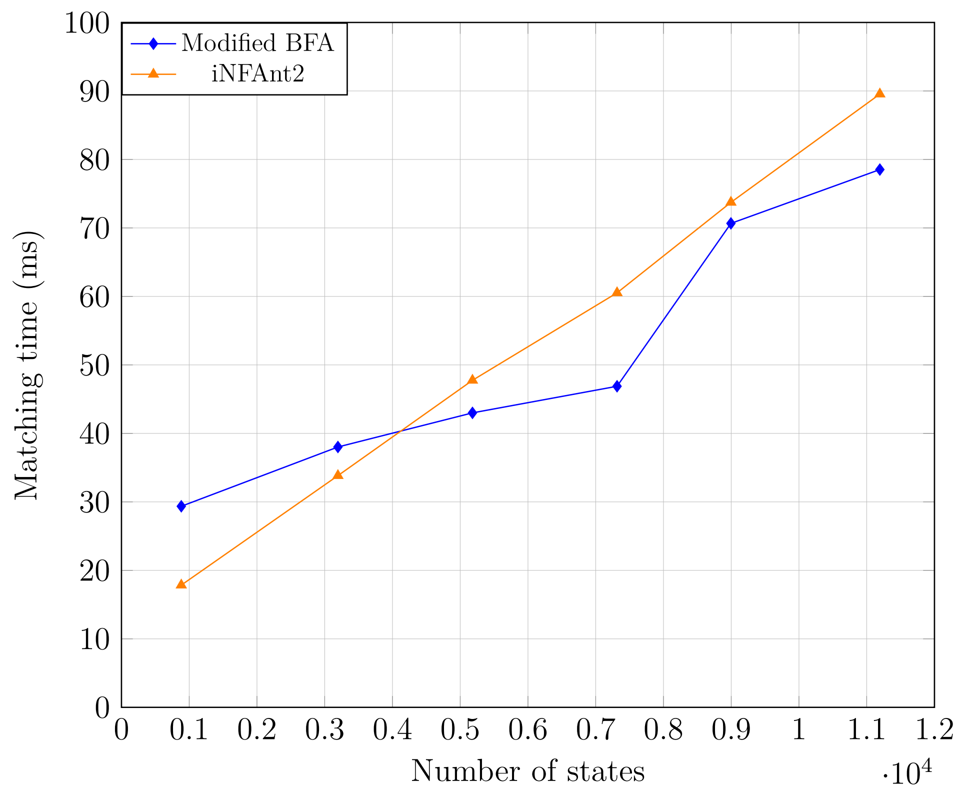


Figure 5.5: The change in matching time over number of states with 200 packets containing 10,000 bytes data

5.5 Summary

In this chapter, we show the experiment results that we obtained from many test cases. The results show that when using the simple ruleset, our method has higher matching time that means lower performance than the original work on the host and the competitor on GPGPUs. However, when using complex ruleset which has large number of states as well as number of transitions, the modified BFA has greater performance compared with the original one and iNFAnt2. The matching time reduces by 50 to 215 times compared with the original BFA on the host. The theoretical speedup ratio is different than the experiment results since some reasons which we have already analysed. Comparing with the iNFAnt2, the matching performance is higher about 29% when using the complex ruleset.

Chapter 6

Conclusion and future work

6.1 Conclusion

In this research, we proposed an enhanced version of the bit-base finite automaton method, which can take advantage of data parallel computation on a GPGPU device. When comparing with the original method on multi-core platforms, our improvement increases the matching performance by 50 to 215 times, depending on the ruleset. For the comparison with iNFAnt2 [13], the matching of our study is lower about 29% when the ruleset contains more than 11,194 states. Hence, our method will reduce the possibility of system bottleneck by moving the packet filtering phase to GPGPU device, which will help CPU to have more available resources for other tasks. Furthermore, the results show potential for scalability since the performance only depends on the device's computational resources. This method will show the best performance in case the network system needs to process protocol streams in continuous flows using complex ruleset to avoid cybercrime.

6.2 Future work

Although we obtains better results than previous works, the throughput of the matching engine is about 24.3 Mbps which is lower compared with some recent studies [5]. We notice that the limit of our work is memory consumption. Since the Boolean matrices are sparse, it wastes a huge amount of device memory to store very little useful information, and the situation becomes worse when the ruleset is more complex. Consequently, computational resources become idle. Our approach is matrix compression, which can reduce the size of sparse matrices. However, the compressed matrix is not easy to perform matrix multiplication in parallel. We will research to

adapt the matrix compression method to our work in the hope of reducing memory consumption but still taking advantage of parallel computation.

Bibliography

[1] NVIDIA. Nvidia tesla p100: The most advanced datacenter accelerator ever built featuring pascal gp100, the world's fastest gpu. Technical report, 2016. Whitepaper.

[2] Marco Nobile, Paolo Cazzaniga, Daniela Besozzi, Dario Pescini, and Giancarlo Mauri. cutauleaping: A gpu-powered tau-leaping stochastic simulator for massive parallel analyses of biological systems. *PloS one*, 9:e91963, 03 2014.

[3] Z. Fu and J. Li. High speed regular expression matching engine with fast pre-processing. *China Communications*, 16(2):177–188, 2019.

[4] Niccolo' Cascarano, Pierluigi Rolando, Fulvio Risso, and Riccardo Sisto. Infant: Nfa pattern matching on gpgpu devices. *SIGCOMM Comput. Commun. Rev.*, 40(5):20–26, October 2010.

[5] C. Xu, S. Chen, J. Su, S. M. Yiu, and L. C. K. Hui. A survey on regular expression matching for deep packet inspection: Applications, algorithms, and hardware platforms. *IEEE Communications Surveys Tutorials*, 18(4):2991–3029, 2016.

[6] R. Mitkov. *The Oxford Handbook of Computational Linguistics*. Oxford Handbooks Series. OUP Oxford, 2004.

[7] Mordechai Ben-Ari and Francesco Mondada. *Finite State Machines*, pages 55–61. Springer International Publishing, Cham, 2018.

[8] Anil Maheshwari and Michiel Smid. Introduction to theory of computation. 2019. `https://cglab.ca/~michiel/TheoryOfComputation/`.

[9] V M Glushkov. THE ABSTRACT THEORY OF AUTOMATA. *Russian Mathematical Surveys*, 16(5):1–53, oct 1961.

[10] Y. E. Yang and V. K. Prasanna. Optimizing regular expression matching with sr-nfa on multi-core systems. In *2011 International Conference on Parallel Architectures and Compilation Techniques*, pages 424–433, 2011.

[11] Yi-Hua E. Yang and Viktor K. Prasanna. Software toolchain for large-scale re-nfa construction on fpga. *Int. J. Reconfig. Comput.*, 2009, January 2009.

[12] Patrick E. O'Neil and Elizabeth J. O'Neil. A fast expected time algorithm for boolean matrix multiplication and transitive closure. *Inf. Control.*, 22:132–138, 1973.

[13] J. Wadden, V. Dang, N. Brunelle, T. T. II, D. Guo, E. Sadredini, K. Wang, C. Bo, G. Robins, M. Stan, and K. Skadron. Anmlzoo: a benchmark suite for exploring bottlenecks in automata processing engines and architectures. In *2016 IEEE International Symposium on Workload Characterization (IISWC)*, pages 1–12, 2016.

[14] Yuan Zu, Ming Yang, Zhonghu Xu, Lin Wang, Xin Tian, Kunyang Peng, and Qunfeng Dong. Gpu-based nfa implementation for memory efficient high speed regular expression matching. *SIGPLAN Not.*, 47(8):129–140, February 2012.

[15] Hongyuan Liu, Sreepathi Pai, and Adwait Jog. Why gpus are slow at executing nfas and how to make them faster. *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020.

[16] Michela Becchi and Patrick Crowley. An improved algorithm to accelerate regular expression evaluation. pages 145–154, 01 2007.

[17] Kien C. Vu, Ryuta Kawano, and Yasushi Inoguchi. Accelerating bit-based finite automaton on a gpgpu device. In *Joint conference of Hokuriku chapters of Electrical and information Societies 2020*, will be presented in Sep 13,2020.