Universidad Autónoma de Madrid

Escuela Politécnica Superior

Máster Universitario en Investigación e Innovación en Tecnologías de la Información y las Comunicaciones

TRABAJO DE FIN DE MÁSTER

LOG FILE REGULAR EXPRESSION PATTERN MATCHING AND CAPTURE WITH GPUS

António Luís Pinto Silva
Advisor: Dr. Jorge E. López de Vergara Méndez

September 2016

Universidad Autónoma de Madrid

Escuela Politécnica Superior

Máster Universitario en Investigación e Innovación en Tecnologías de la Información y las Comunicaciones

TRABAJO DE FIN DE MÁSTER

CAPTURA DE PATRONES EN ARCHIVOS DE LOGS MEDIANTE EL USO DE EXPRESIONES REGULARES EN GPUS

António Luís Pinto Silva
Tutor: Dr. Jorge E. López de Vergara Méndez

Septiembre de 2016

LOG FILE REGULAR EXPRESSION PATTERN MATCHING AND CAPTURE WITH GPUS

Author: António Luís Pinto Silva
Advisor: Dr. Jorge E. López de Vergara Méndez

High Performance Computing and Networking Research Group
Departamento de Tecnología Electrónica y de las Comunicaciones
Escuela Politécnica Superior
Universidad Autónoma de Madrid

September 2016

Abstract

Abstract — The information contained in a system is normally stored into log files. Most of the time, these files store the information in plain text with many not formatted information. It is then necessary to extract parts of this information to be able to understand what is going on such system. Currently, such information can be extracted using programs that make use of extended regular expressions. The use of regular expressions allows the search of patterns but it can be also used to extract data from the searched pattern. Most of the programs that implement regular expressions are based on finite automatas, such as non-deterministic (NFA) or deterministic (DFA). We aim to explore the use of finite automatas to extract data from log files using a Graphic Processor Unit (GPU) device to speedup the process. Moreover, we will also explore data parallelism over the lines present on the log file. Currently, the work done in GPU with regular expressions is limited to matching tasks only, without any capture feature. We present a solution that solves this lack of pattern capture in current implementations. Our development uses as base the implementation of TNFA and converts it to a TDFA before running the GPU task. We explore the new CUDA feature named unified memory, supported since CUDA 6, together with streams to achieve the best possible performance in our GPU implementation. Using real log files and regular expressions made to extract specific data, our evaluation shows that it can be up to 9× faster than the sequential implementation.

Key words — GPU, CUDA, regular expressions, pattern matching, submatching, pattern capture, tagged-NFA, tagged-DFA

Resumen

Resumen — La información contenida en un sistema normalmente se almacena en archivos de registros, conocidos comúnmente como *logs*. La mayor parte de las veces, estos archivos almacenan la información en texto plano, con mucha información sin formatear. Por ello es necesario extraer partes de esta información, de forma que se pueda saber qué está ocurriendo en dicho sistema. Actualmente, esta información se puede extraer usando programas que aprovechan las expresiones regulares extendidas. Su uso permite la búsqueda de patrones, pero también se pueden emplear para extraer datos del patrón buscado. La mayoría de los programas que implementan expresiones regulares se basan en autómatas finitos, tales como los no deterministas (NFA) y los deterministas (DFA). El objetivo de este Trabajo Fin de Máster es explorar el uso de autómatas finitos para extraer datos de archivos de *log* usando una GPU para acelerar el proceso. Es más, también exploramos el paralelismo que se puede aplicar sobre las líneas de un archivo de *log*. En la actualidad, el trabajo realizado con GPUs y expresiones regulares se limita a tareas de búsqueda de patrones, sin ninguna funcionalidad de captura. Presentamos una solución que resuelve esta falta de funcionalidad en las implementaciones actuales. Nuestro desarrollo usa como base una implementación de TNFA y la convierte a TDFA antes de ejecutar la tarea en la GPU. Exploramos la nueva funcionalidad de CUDA denominada memoria unificada, que se soporta desde la versión 6 de CUDA, así como el uso de flujos o *streams* para alcanzar el mejor rendimiento posible en nuestra implementación en GPU. Al usar archivos de *log* reales y expresiones regulares para extraer datos específicos, nuestra evaluación muestra que la implementación paralela es hasta 9 veces más rápida que la implementación secuencial.

Palabras clave — GPU, CUDA, expresiones regulares, búsqueda de patrones, captura de patrones, tagged-NFA, tagged-DFA

Acronyms

CPU Central Processing Unit. 2, 22, 27–30, 32

CSV Comma Separated Values. 17

CUDA Compute Unified Device Architecture. 12

DFA Determinist Finite Automata. 7, 11, 27

GPU Graphic Processor Unit. I, III, 1–3, 5, 11, 12, 15, 21, 22, 27, 28, 30–32

JSON JavaScript Object Notation. 17

NFA Non-determinist Finite Automata. 7, 11, 12, 27

TDFA Tagged Determinist Finite Automata. 10, 22, 27, 30

TNFA Tagged Non-determinist Finite Automata. 9, 22

Contents

List of Tables

Listings

List of Figures

1 Introduction

It is important to analyze the information contained in the system logs to understand what is going on such system. These logs can be generated by local applications or received from external systems. From this analysis it is possible to detect problems, usage patterns and/or planning system optimization's. Reading log files manually line by line is unpractical, a much smarter solution is to tackle the problem programmatically.

Extracting information from the logs is not an easy task due to the variety of information and multiple formats that can be found in these files, and especially because sometimes it is necessary to process a huge volume of data. The fields that can be found in the log files are normally the timestamp, program name, level, message. The most used formats are the common logfile format or NCSA Common log format [4] and syslog [5]. The information stored in log files can be used for multiple analysis [6], such as forensics [7], organizational security [8], user behaviour [9].

Normally, given that logs are stored in text files, regular expressions are the key for powerful, flexible, and efficient text processing. Regular expressions can add, remove, isolate, and generally fold, spindle, and mutilate all kinds of text and data [10]. They are already used in application domains such as bibliographic search, this reason makes the regular expressions suitable and recommended for log manipulation.

This thesis will be focus on the usage of regular expressions to extract information from the log files with help of GPU to accelerate the process. Current tools, such as [11,12], allow the use of regular expressions in log files to search and extract data, but none of them can take advantage of the parallel execution on a GPU co-processor to improve the search and matching process.

1.1 Motivation

In general, current implementations where regular expressions are used in the GPU context only explore the match capability of the regular expression. Central Processing Unit (CPU)-based tools allow matching and extracting data using regular expressions using capture groups. We intended to explore the match capability and focus our work to extract data in GPU, using the capture group feature.

1.2 Objectives

The main objectives to reach in this thesis are the following ones:

- Implement a solution capable of reading log files, match and search content using regular expressions in GPU. To meet this goal, the solution must be able to extract information from the capture groups used in regular expression.
- Support for multiple pattern search and capture the corresponding match.
- Export the data contained in the capture groups into common formats to integrate with other tools, such as CSV, XML and JSON.
- Evaluate our implementation comparing parallel GPU versus sequential CPU implementation.
- Evaluate our implementation comparing it with current tools available for CPUs (sed, awk).

1.3 Used Methodology

During the elaboration of this work we followed the steps below:

- The first phase consisted on finding how the regular expressions were implemented in the GPU context to extract data from log files. During this phase it was important to identify the current State of the Art, and current implementations for regular expressions in the GPU context, listing the most common methods, their improvements and their limitations.
- Secondly, the implementation phase. From the previous phase, the implementation attempted to solve the limitations found or lack of implementations in the literature in order to achieve the main goals of this work.
- As complementary to the previous phase, the micro-benchmarks step was necessary to evaluate the proposed solutions on key points in the implementation algorithm, like the time needed when converting between automatas, or measure the time need to copy data between the GPU and CPU,
- Next, the experimental setup phase. This final phase consisted on the validation of the entire solution by comparing the results against CPU implementations.
- Finally, the documentation of all the previous work, as provided in this document.

1.4 Thesis Organization

This thesis is organized as follows: in Chapter 2, we show the background and related work in the context of regular expression matching on GPU implementation. Then, in Chapter 3, we present our proposed program for regular expression matching and capture engine for GPU based on tagged deterministic finite automata (TDFA). Later, in Chapter 4, we provide an experimental evaluation of the proposed solution and compare the results against current tools. Finally, in Chapter 5, we conclude our discussion.

2

State of the Art

2.1 Introduction

As commented before, it makes the most sense to use regular expression to find and extract data from the log files. So, it is important to understand what regular expressions are and how they are implemented at the present. This chapter provides the background about regular expression and their equivalence with finite automatas, as well as common algorithms used to implement them. It also provides a background on current works in GPUs that make use of regular expressions in the pattern matching task.

2.2 Regular Expression

Regular expressions can be defined as a pattern that describes a set of strings. In a regular expression we can find two types of characters, the literal (also called normal characters) —this type of characters match themselves; and special characters, called meta-characters, that have a special mean (check table 2.1 for a list of some meta-characters [1] that can be used inside a regular expression).

A formal definition of regular expression can be found in [13], where a regular expression over an alphabet Σ can be defined as:

1. α for some symbol in the alphabet Σ,
2. ε is a regular expression for the "empty" string,
3. $\emptyset$, represents a empty language,
4. for any regular expression r1 and r2 over Σ, in languages L_{r1} and L_{r2} respectively, the regular expressions can apply:

[1]check `http://www.regular-expressions.info/characters.html` for a full list of meta-characters

Table 2.1: Regular expression example of meta-characters

Meta-character	Name	Matches
.	dot	any single character
$*$	star	zero or more characters
?	question mark	0 or 1 character
ˆ	caret	the string start with character after the symbol caret
$	dollar	the string ends with character before the symbol dollar
[...]	character class	any character listed
[ˆ...]	negate character class	any character not listed

   (a) r1 ∪ r2, alternation, corresponding to the language $\mathrm{L}_{\mathrm{r1}} \cup \mathrm{L}_{\mathrm{r2}}$,
   (b) r1r2, concatenation, corresponding to the language $\mathrm{L}_{\mathrm{r1}}\mathrm{L}_{\mathrm{r2}}$,
   (c) r1*, kleene star, corresponding to the language L_{r1}*,

5. if expressions follow the rules above, we can say that they are regular expressions over Σ.

In the alphabet used to build regular expressions, meta-characters are not part of the alphabet, and to be able to use this characters in the match process is common to use an escape character, backslash, \, like for example, \?, to match the character ?, and avoid it to be interpreted as meta-character.

The elements that can be present in a regular expression are:

- **Literal** matches a single character, example, $/a/$
- **Character ranges** matches more than a single character, denoted inside brackets. For example, the expression $/[ab]/$ will match any of the letters a or b, the expression $/[a-z]/$ will match any lowercase letter from a to z.
- **Negated character ranges**, as previous element, it allows matching more than a single character but negate the value inside brackets. For example, $/[\hat{}ab]/$, matches everything except any of the words a or b.
- **Alternation** allows joining two different expressions into just one expression using the meta-character, |, meaning *or*. For example, $/ab/$ and $/cd/$ can be put together in a single expression $/ab|cd/$, it will match ab or cd, but will not match $acbd$.
- **Concatenation** allows matching the string in order. For example $/a[bc]/$, this will match a followed by b or c.
- **Star operator** allows the repetition of the preceding element zero or more times. For example, $/a*/$, can match the string a, aa, aaa and so on. It also matches the null or empty string.
- **Plus operator**, it is similar to the star operator, but instead it expects to have one or more repetitions of the preceding element. For example, $/a+/$, can match the string a, aa and so on.
- **Optional operator** it means that the element that precedes the meta-character ? can be present or not in the string for a successful match. For example, $/colou?r/$, can match the string *color* and also the string *colour*

2.2.1 Capture groups

One of the important features that we can find in regular expressions is the capturing group, also known as submatch, it allows extracting data from the matching results after a positive match. For a regular expression like *number=(\d+)* with and input string *calling the number=1234.*, can extract the string *1234* specified by the capturing group (the expression wrapped by the pair of parenthesis). In this work we will focus on this way of extracting data with regular expressions.

2.3 Finite automata

Regular expressions and finite automata are equivalent in their descriptive power [14]. There are two main automatas suitable for regular expression task, such as Non-determinist Finite Automata (NFA) and Determinist Finite Automata (DFA). In figure 2.1 its the equivalence between regular expression and Finite automatas, where from a regular expression we can obtain a NFA, convert it to a DFA, and from the DFA convert it back to a regular expression. We can find NFA and DFA-based approaches in GNU sed [11], GNU grep [15], GNU awk [12] and RE2 [16].

There have been many studies, such as [17–25] that improve the the use of NFA and DFA automatas to execute the task of regular expression matching. Cox in [26–28] exposes the use of finite automatas to implement regular expression using NFA with virtual machines and a DFA approach with cached states, that is the DFA states are expanded as necessary when matching the string. In [29], Becchi distributes a regular expression engine that can be used to build NFA and DFA-based solutions.

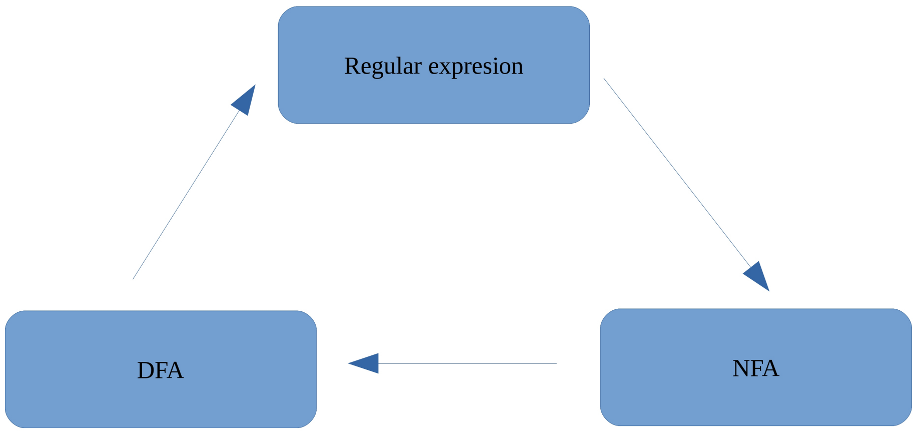


Figure 2.1: Equivalence between regular expressions and Finite automatas

2.3.1 Non deterministic finite-state automata

To convert a regular expression to an NFA, the well known Thompson's construction algorithm [30] can be used. Figure 2.2 represents the NFA for the regular expression $a * def$.

An NFA can be defined as a 5-tuple $(S, \Sigma, \delta, s_0, F)$ where

- S - finite set of states

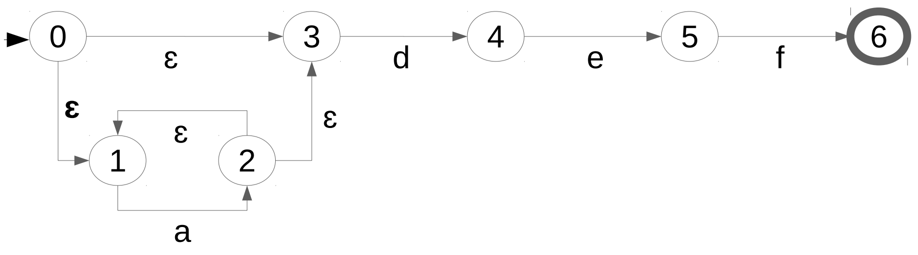


Figure 2.2: Thompson-McNaughton-Yamada NFA for regular expression: $a * def$

- Σ - the input alphabet
- δ - the transition function that takes a state in S and an input symbol in Σ as arguments and returns a subset of S
- $s_0 \in S$ - the start sate
- $F \subseteq S$ - the set of final, or accepting, states

It is called non-deterministic because of the choice to move that may lead from one state to another. When trying to match a string, in the worst case, it requires up to n NFA states traversals per input character processed. The complexity, when converting a regular expression, r, to an NFA, and simulate the NFA on string x is:

- $O(|r|)$ time and $O(|r|)$ space for reduction to NFA.
- $O(|r| \times |x|)$ time for simulation of NFA (it depends on the special characteristics of the NFA obtained by reduction).

2.3.2 Deterministic finite-state automata

From any NFA we can obtain a DFA using the subset construction algorithm [14], which has the time complexity $O(2^m)$. The DFA that results from this conversion will recognize the same language as the source NFA.

In the conversion to a DFA state, each set of NFA states reached in parallel is associated upon processing a given character, and because of this, when converting an NFA to an equivalent DFA it may result in a state explosion [31]; that is, the number of resultant DFA states increase exponentially, a theoretical worst case study shows that a single regular expression of length n can be expressed as a DFA of up to $O(m^n)$ states, where m is the size of the alphabet, 128 for the extended ASCII character set.

A DFA can be defined as a 5-tuple $(S, \Sigma, \delta, s_0, F)$ where:

- S - finite set of states
- Σ - the input alphabet
- δ - the transition function that takes a state in S and an input symbol in Σ as arguments and returns a singe state of S, opposite to the NFA transition function that can return more than a single state.

- $s_0 \in S$ - the start sate
- $F \subseteq S$ - the set of final, or accepting, state

The DFA, opposite to the NFA, has a single state at any time when executing the matching process and this make it possible to search a text at of length n in $O(n)$ time.

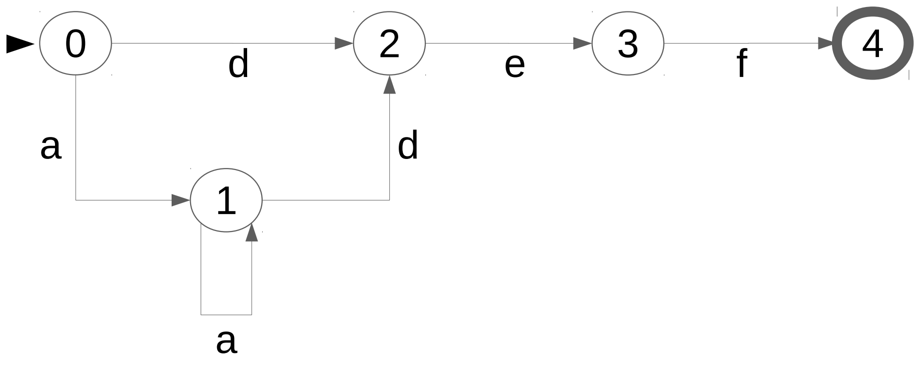


Figure 2.3: Subset Construction DFA corresponding to the NFA in figure 2.2 for the regular expression $a * def$

The complexity when using DFA is that we must convert a regular expression, r, to an NFA; then convert the NFA to a DFA, and simulate the DFA on string x is:

- $O(|r|)$ time and $O(|r|)$ space for reduction to NFA
- $O(2^{|r|})$ time and space for reduction from NFA to DFA
- $O(|x|)$ time for simulation of DFA on x. Due to this a backtracking algorithm cannot used.

2.3.3 Tagged state automata

The previous algorithms are only focused on answering the input string matches the language and do not allow extracting data from the matching results, which is one of the most import features that one can found when working with regular expressions for log processing.

To solve this problem, Lauraki in [32,33] proposed Tagged Non-determinist Finite Automata (TNFA), an NFA-based approach for submatch extraction, where an NFA is augmented with tags to represent capturing groups. The tags are of the form t_x, where x is an integer and each tag has a corresponding variable that can be set and read. When a tagged transition is used, the current position in the input string is assigned to the corresponding variable. On a positive match, is possible access the information in the tags to extract the content of the match. His implementation can be tested using the library TRE[2].

The used of tagged automatas was also explored by Karper, in [34], where the resulting work is a regular expression engine that produces parse trees. The program code of this work can be found in GitHub[3].

[2]http://laurikari.net/tre/
[3]https://github.com/nes1983/tree-regex

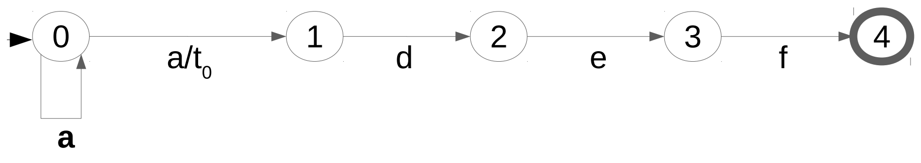


Figure 2.4: TNFA for the regular expression $(a*)def$

In figure 2.4 we can find the the TNFA obtained from the regular expression $(a*)def$, as the figure shows no ϵ states are present. This is because the TNFA generated is ϵ-free. Also notice the parenthesis before the char a and before the char d, this delimits the capture group, in this case the tag number 0. For example, when running the TNFA with input string $aaadef$ we will have a possible match and its possible to obtain the value aaa as the value for tag 0.

The TNFA is defined by a 5-tuple $(K, T, \Sigma, \Delta, s, F)$ [32], where:

- K is a finite set of states,
- T is a finite set of tags, $w \in T$,
- Σ is an alphabet,
- Δ is the prioritized tagged transition relation, a finite subset of $K \times \Sigma^* \times T \times r \times K$, where $r \in \mathbb{N}$ is unique for all items of the relation.
- $s \in K$ is a initial state,
- $F \subseteq K$ is the set of final states.

Lauraki [32, 33] also proposes the algorithm to build a Tagged Determinist Finite Automata (TDFA) version from a TNFA. As for the previous automatas shown, the conversion from a TNFA to a TDFA could take some time but it needs to be done only once. Then, the matching of characters is faster. Figure 2.5 shows the converted TNFA from the regular expression $(a*)def$. A TDFA-based implementation can be found in regex-tdfa [35] which is a pure Haskell[4] regular expression library.

The used algorithm has the same basic principles to convert a TNFA to a TDFA as the subset construction algorithm used to convert traditional NFAs to traditional DFAs with the particularity of the need to keep track where the tag values were stored.

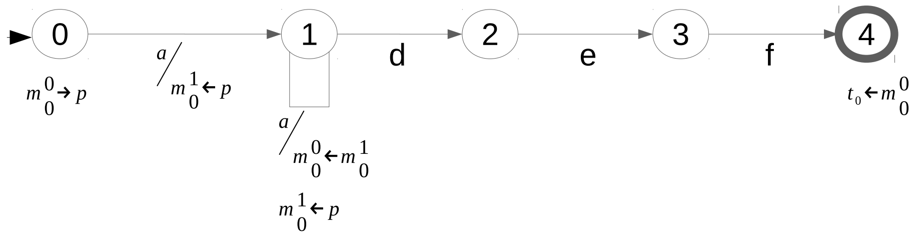


Figure 2.5: TDFA for the regular expression $(a*)def$

The TDFA can be defined as a 7-tuple $(K, \Sigma, \delta, s, F, c, e)$ [32], where

[4]more info in https://www.haskell.org/

- K is a finite set of states,
- Σ is an alphabet,
- δ is the transition function, a function from $K \times \Sigma$ to $K \times C$
- $s \in K$ is a initial state,
- $F \subseteq K$ is the set of final states.
- $c \in C$ is the initializer, a list of commands to be executed before the first symbol is read,
- $e \in F \times C$ is the set of finishers, a list of commands to be executed after the last symbol is read. Each final state has its own finisher list.

2.4 GPU implementations

The GPU implementations have the advantage of increasing the performance of the regular expression task. Most of this implementation has been conducted using the GPU hardware from NVIDIA, which supported the program languages OpenCL [36] and CUDA [37], being this last one the most used.

Regular expressions with finite automatas such as NFAs and/or DFAs have already been implemented in the GPU context essential for matching task, like in the network field to identify the type of traffic [38–51], exploring the parallelism of the data. However, as we will see, none of them is focused on data extraction, which is crucial for log processing.

The use of the NFA-based implementation was initial purposed in [41], the iNFAnt proposal, but later optimized by Gómez Nieto in [46], who detected some limitations in the initial implementation of iNFAnt and proposed the use of Virtual NFA, which creates a group of states from the NFA.

Gómez Nieto in [46] and Leira Osuna in [47] also proposed a DFA-based solution and compared it to the NFA implementation, concluding that it is better to use a DFA-based solution in the GPU context when applied to network traffic inspection. In [44], Wang also claims that the use of a DFA implementation has better performance compared to NFA-based solutions. Even if it takes time to convert the NFA to a DFA, when using GPU to perform DFA matching, massively threads execution concurrently could hide this time efficiently.

Yu and Becchi [48] present two different approaches to use regular expression with GPUs: an NFA-based solution and a DFA-based solution. The NFA solution is also an improvement from iNFAnt [41] where three optimizations are provided: changing the loop when executing the NFA states inside the GPU making it more efficient for larger NFA; group states according to their incoming transitions; and last optimization overlapping groups from the two previous observations. In this study it is also possible to find a DFA solution, in the case two approaches where suggest an uncompressed DFA-based solution and a compressed DFA-based solution that can be used in larger datasets.

A DFA-based solutions is also implemented by Ponnemkunnath and Joshi in [52] and pointed out the problem of state exploding that could occur when converting the NFA to a DFA. To prevent greedy memory consumption caused by some regular expressions, the conversion to NFA cannot exceed 5000 states, which, at the time that paper was written, it covered a 97% of the Snort default rule set.

For text processing, Bellekens *et al.* [53] proposed generic log processing library, which can also be used for deep packet inspection, but instead of using regular expression and by consequent finite automatas, it used the single pattern algorithm Knuth-Morris-Pratt [54]. Although this implementation lacks the possibility to extract sub-matches, it will be used to compare our results in the search phase.

In [55] we can find the code where the authors had implement a NFA-base solution porting the implementation from Ross [26] to GPU.

Based on the research done, at the present the literature does not have any implementation using GPU and regular expressions with finite automatas to extract capture groups on a positive match. One of the solutions expected to achieve in this work is the use of tagged automatas for this task.

2.4.1 CUDA Unified Memory

The Compute Unified Device Architecture (CUDA) programming language used to program the NVIDIA GPU cards has been improved version after version, introducing features that make easier to program and other ones that improve performance.

In this subsection we focus on a feature introduced since version 6.0, called Unified Memory [56, 57]. It can simplify the memory management in GPU-accelerated applications and it also provides performance gains through data locality.

The study conducted by Landaverde *et al.* in [58] investigated CUDA unified memory access, and it concluded that there are cases where using this feature can improve the performance of the program. However, it is still not yet the unique solution, because in some scenarios using another type of memory management is faster than unified memory access. The performance of CUDA unified memory access varies significantly based on the memory access patterns.

2.5 Conclusions

The regular expression can be implemented using finite automatas such as NFA and/or DFA. We can find NFA and DFA-based approaches in GNU sed [11], GNU grep [15], GNU awk [12], and RE2 [16]. The capture group or submatch extraction from a regular expression with finite automatas use tagged automatas like TNFA and/or TDFA.

In the GPU context there have been multiple works for pattern matching but none of them has addressed the possibility to extract data from the matching results. Most proposals have targeted at NVIDIA GPUs and selected the programming language CUDA. Although NVIDIA had introduced since CUDA 6.0 the so-called unified memory access feature, it is necessary to pay special attention to memory access patterns to achieve the best performance.

At the present there are not implementations of TNFA and/or TDFA with GPUs, which is a limitation when exploring the usage of regular expression in the GPU context.

3
Proposed solution

3.1 Introduction

This chapter is devoted to explain the proposed solution, where regular expressions and GPU can be used together to extract information from text files. The main goal of this section is to detail the methods used in the implementation of the proposed solution, the environment setup and later a performance analysis to understand well the results that can be obtained.

3.2 Algorithm

Like the previous implementations found in chapter 2.4, in this work the parallelism is also done at the data level and not in the automata lookup. The proposed algorithm consist in three core functions `tnfa2tdfa`, `data2gpu` and kernel `gpuMatchGet`.

The algorithm 1 is responsible of converting the TNFA to TDFA, the conversion from regular expression to TDFA uses the proposed solution by Laurikari in [32]. Line 3 is the result of that conversion. The decision to use TDFA instead of TNFA is because the TDFA are far more efficient and faster to use with the GPU. But it is necessary to consider the problem of state space explosion when converting from a TFNA to a TDFA, as in when converting from a NFA to a DFA, and also the TDFA automata can increase in memory consumption. To solve this issues the algorithm will only accept converting a regular expression with less that 5000 states (see 2.4).

The resulting TDFA automata (line 3) is then converted (lines 4 - 15) to a transition table where the key is the current TDFA state plus the current symbol, and the value containing the next state, the information about the presence of a tagged state and if is a final state. This format will allow a faster matching and search task inside the GPU avoiding any extra loop needed to search for the input symbol and state on every thread, in this way this is computed

Algorithm 1 Converting TNFA to TDFA

```
1: procedure TNFA2TDFA(tnfa)
2:     $tdfaTransitionTable \leftarrow \emptyset$
3:     $tdfa \leftarrow tnfa\_to\_tdfa(tnfa)$
4:     $state \leftarrow 0$
5:     while $state < |dtfa.numberOfStates|$ do
6:         for each input symbol a do
7:             $nextstate \leftarrow$ TDFA_NEXT_TRANSITION$(state, a)$
8:             $tagid \leftarrow$ TDFA_TAG_ID$(nextstate)$
9:             $final \leftarrow$ TDFA_IS_FINAL$(nextstate)$
10:            $nextstate \leftarrow$ ShiftLeft$(nextstate, 16)$
11:            $tagid \leftarrow$ ShiftLeft$(tagid, 15)$
12:            $tdfaTransitions[state \times |\Sigma| + a] \leftarrow nextstate$ OR $tagid$ OR $final$
13:        end for
14:        set state to state + 1
15:    end while
16:    return tdfaTransitions
17: end procedure
```

only once. It is important to notice that the single state cannot have multiple tag Ids, and when it start a new tag id the previous one is closed, they cannot overlap over different TDFA states.

The size of the transition table will be $m \times n$, where m is the number of states and n the is the alphabet size. The time complexity to convert will take $O(n^m)$ where n is the number of TDFA states and m is the alphabet size.

Another approach to this solution is to execute the TDFA directly in the GPU kernel, so that each thread could search the input symbol from the input text in the TDFA, but that would required extra memory operations to copy the TDFA to the device, as well the mapped tagged arrays, and each thread inside the GPU kernel would had to loop over the TDFA states. Using this format, it is only necessary to loop over the input text.

The value used in the transition table is composed by a 32 bits value where the first 16 bits are to store the TDFA state, giving us the possibility to store 65536 states, then the next 15 bits are to store the tag id and it can store 32768 different tags ids, and the last bit is used to store the information about if the state is final or not. When the automata reaches a final state the input text is accepted and we have a positive match.

When programming for the GPU, it is necessary to consider the architecture of the target device in order to adjust the algorithm to fit in the device hardware specifications. In this case, the GPU belongs to the first generation Maxwell architecture (figure 3.2) with the chip GM107 (figure 3.1). This architecture, as explained in 2.4.1, supports the unified memory feature and it can be explored to speed up the data transfer between host and device. In Algorithm 2 it is explained how this task is accomplished in a efficient manner. In lines 4, 5 and 6 we used the pool memory shared between the host and device, later in order to have concurrency between data transfer and computation we used streams [2], lines 10 to 16 split the data into multiple streams (figure 3.3).

In line 7 the TDFA transition table is transferred only once in the default stream to the GPU global memory and it is available to all streams. When reading the content from the file and

Algorithm 2 Transfer data to/from GPU

```
1: procedure DATA2GPU(file, tdfaTransitions, streamSize)
2:     $number\_of\_lines \leftarrow$ FILE_NUMBER_OF_LINES($file$)
3:     $length \leftarrow$ FILE_LENGTH($file$)
4:     $lines \leftarrow$ pointer to allocated unified memory in device
5:     $positionlines \leftarrow$ pointer to allocated unified memory in device
6:     $result \leftarrow$ pointer to allocated unified memory in device
7:     Copy $tdfaTransitions$ to device on default stream
8:     set $streamstonumber\_of\_lines/streamSize$
9:     set $N$ to 0
10:    for $N < streams$ do
11:        $offset \leftarrow N \times streamSize$
12:        Create stream $N$
13:        $lines, positionlines \leftarrow$ FILE_READ_LINES($file, offset$)
14:        GPUMATCHGET(lines, positionlines, tdfaTransitions, result, offset) in stream $N$
15:        set $N$ to $N + 1$
16:    end for
17:    synchronize device
18:    $N \leftarrow 0$
19:    for $N < streams$ do
20:        destroy stream $N$
21:        set $N$ to $N + 1$
22:    end for
23: end procedure
```

have it split into multiple lines it is important to fix a maximum length on a single line, and if the end line character is not detected force the data to split into a different line.

It's very important to avoid no-coalesced access to global memory which can cause affect directly the speed of the our program. To avoid this type o access its possible to use some techniques like transpose lines and columns (see figure 3.4) so the data is locally available to the threads. This is because a wrap transition on the GPU can read a 128 byte word and this data is cached in the unified L1 / texture cache, so when another thread in the same block request the data it's already cached. Another technique is to store the data to be process by the group threads in the wrap in shared memory. It can be done by a single thread at the start of the wrap execution. The memory access by threads to shared memory are all no-coalescing.

The final function is the task executed on the GPU, the `gpuMatchGet` (algorithm 3) is responsible of executing the transition table on each input symbol per line in the GPU, on a successful match fill the *result* array indicating the presence of successful match on the respective line, and in the presence of tagged states it copies the content of the line of the respective sub-match data.

All the process of matching and search algorithm is parallel over the data, typical of a simple instruction multiple data architecture (SIMD), where the same instructions matching the TDFA automata is executed over multiple data, the lines by many threads.

The GM107 card provides 64 KB of shared memory per SMM but there is a limit of 48 KB

Algorithm 3 kernel gpu match and get

```
1: procedure gpuMatchGet(hostlines, positionlines, hosttdfaTransitions, result, offset)
2:     $tid \leftarrow ThreadID$
3:     __shared__ $localtransitions[]$
4:     if thread id equals 0 then
5:         $transitions[] \leftarrow tdfaTransitions$
6:     end if
7:     syncronize threads
8:     $line \leftarrow hostlines + positionlines[tid]$
9:     set curstate to 0
10:    set taggedcontent to $\emptyset$
11:    while $pos < |line|$ do
12:        fetch the next input symbol c from line
13:        if $c + |\Sigma| \times curstate$ in tdfaTransitions then
14:            $value \leftarrow tdfaTransitions[c + |\Sigma| \times curstate]$
15:            $nextstate \leftarrow \textsc{ShiftRight}(value, 16)$
16:            $tagid \leftarrow \textsc{ShiftRight}(value, 15)$ AND 0xFF
17:            $finalstate \leftarrow value$ AND 0xFF
18:            if nextstate is inferior to currentstate then
19:                set taggedcontent to $\emptyset$
20:            end if
21:            if tagid different from 0 then
22:                taggedcontent[tagid] append c
23:            end if
24:            if finalstate is active then break
25:            end if
26:        end if
27:        set pos to pos + 1
28:    end while
29:    if finalstate is active then
30:        set $result[tid]$ to $taggedcontent$
31:    else
32:        set $result[tid]$ to $-1$
33:    end if
34: end procedure
```

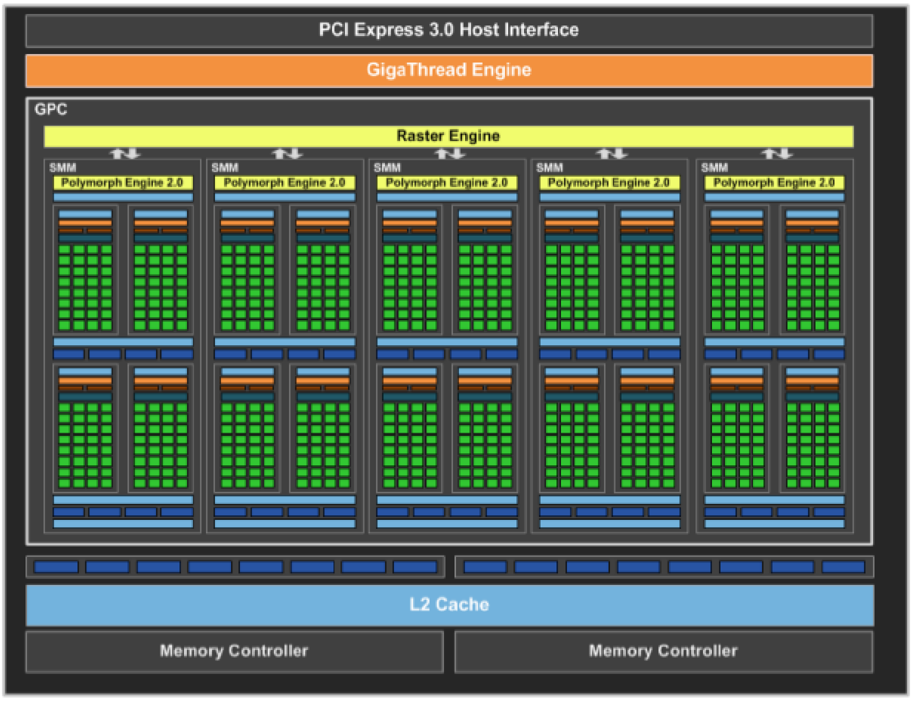


Figure 3.1: GM107 Full-Chip Block Diagramm, adapted from [1]

per-thread-block usage[1]. Figure 3.5 shows the information on the supported card used in this work. Storing the data into the shared memory, which is accessible for all the threads of a single block, will accelerate the access to the data for each thread when executing the matching and search process over the TDFA transition table, instead of reading from the global memory which is slower, also by using the shared memory to store the transitions to be processed we will avoid no-coalesced access to global memory. The different memories that can be used in the GPU devices is shown in figure 3.6.

3.3 Implementation

The library *libtre*[2] that creates epsilon-free automata is used o build the TNFA. Basically, it is like Thompson's construction [14], but the ϵ closures are precomputed. This is to avoid computing the ϵ closures over and over again when executing the TNFA.

From the TNFA, a TDFA is constructed using the algorithm proposed by Laurikari in [32].

Once we get the TDFA, it is converted to a format that makes it faster to run on the GPU kernel. The transition table obtained for the TDFA from figure 2.5 is shown in table 3.1. Using this approach, it is necessary to build the entire TDFA only once, so that later we can take advantage in the GPU matching process where the multiple threads only need to lookup the input string in the transition table.

It is possible to choose the format of the output results. The available formats are Comma Separated Values (CSV) or JavaScript Object Notation (JSON) (see Annex A). These two

[1]`http://docs.nvidia.com/cuda/maxwell-tuning-guide/index.html#shared-memory-capacity`
[2]http://laurikari.net/tre/

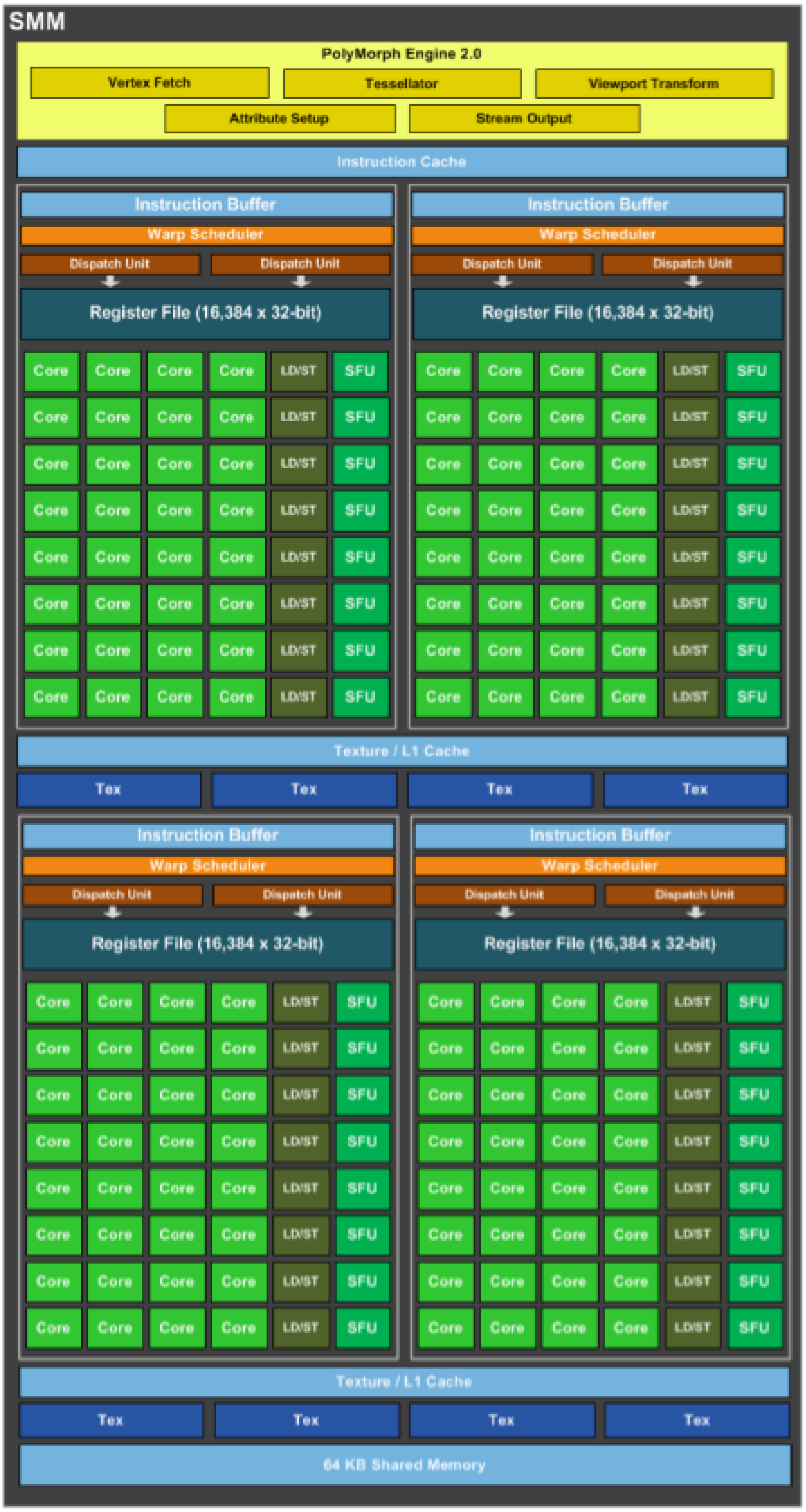


Figure 3.2: Maxwell SM Block Diagram, adapted from [1]

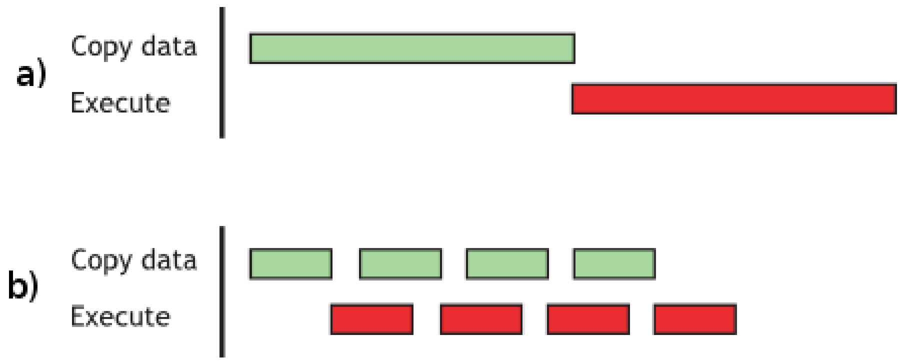


Figure 3.3: CUDA Streams: a) default stream (sequential); b) multiple streams (concurrent), adapted from [2]

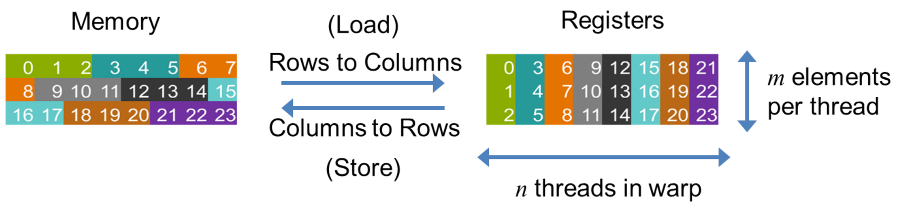
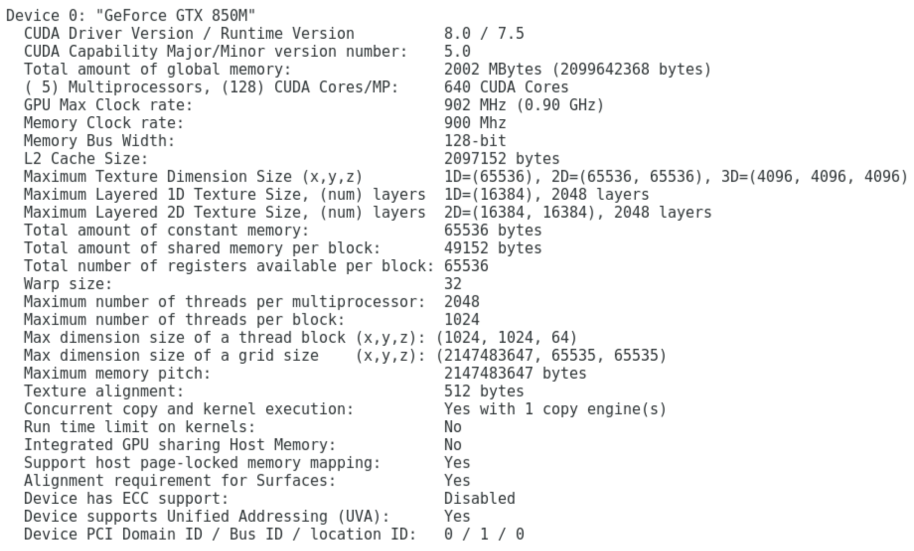


Figure 3.4: CUDA transpose data to avoid no-coalesced access to global memory , adapted from [3]



```
Device 0: "GeForce GTX 850M"
  CUDA Driver Version / Runtime Version          8.0 / 7.5
  CUDA Capability Major/Minor version number:    5.0
  Total amount of global memory:                 2002 MBytes (2099642368 bytes)
  ( 5) Multiprocessors, (128) CUDA Cores/MP:     640 CUDA Cores
  GPU Max Clock rate:                            902 MHz (0.90 GHz)
  Memory Clock rate:                             900 Mhz
  Memory Bus Width:                              128-bit
  L2 Cache Size:                                 2097152 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(65536), 2D=(65536, 65536), 3D=(4096, 4096, 4096)
  Maximum Layered 1D Texture Size, (num) layers  1D=(16384), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(16384, 16384), 2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  2048
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 1 copy engine(s)
  Run time limit on kernels:                     No
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Disabled
  Device supports Unified Addressing (UVA):      Yes
  Device PCI Domain ID / Bus ID / location ID:   0 / 1 / 0
```

Figure 3.5: CUDA device query on GeForce GTX 850M

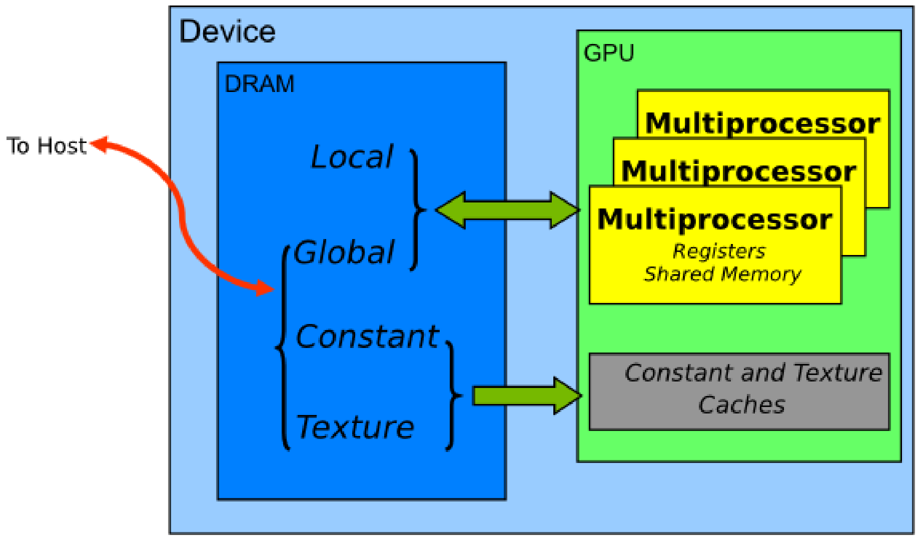


Figure 3.6: Memory spaces on a CUDA device, adapted from [2]

formats were chosen to allow an easier integration with current log analysis tools like elastic search[3], logstash[4] and others.

Key		**Value**		
TDFA state	**input symbol**	**nextstate**	**tagid**	**finalstate**
0	a	1	1	0
1	a	1	1	0
1	d	2	0	0
2	e	3	0	0
3	f	3	0	f

Table 3.1: TDFA transition table for regular expression (a*)def

3.3.1 Test setup

The implementation has evaluated on a system consisting of an Intel Core I7-4710HQ CPU @ 2.50GHz and NVIDA GTX 850M GPU running Linux 4.4.16. The GPU device contains 640 CUDA cores at 876 MHz, and is equipped with 2 GB of global memory.

In the experimental setup we have used real log files obtained from a running server on the cloud, that had running email, web server and VoIP services. The patterns used to search for information were the common ones used in most daily task, like for example, obtain an IP address that accesses the web server service. The generated logs on the server were obtained in different dates and so we have different file sizes to work with.

[3]`https://www.elastic.co/`
[4]`https://www.elastic.co/products/logstash`

3.4 Performance analysis

Before converting a serial task to a parallel task it is necessary to understand if the serial task can be parallelized and if it can scale to a better speedup. The Amdahl's law can gives us the theoretical speedup when using multiple processors [59], in our case the GPU.

In figure 3.7 we can see the sequential match and search when running a TNFA match on some log file. The function we want to parallelize is *nfa_match_string* and from the profiling, this function takes 93% of the total execution time. The expected speedup improvement would not exceed 14× according to Amdahl's law.

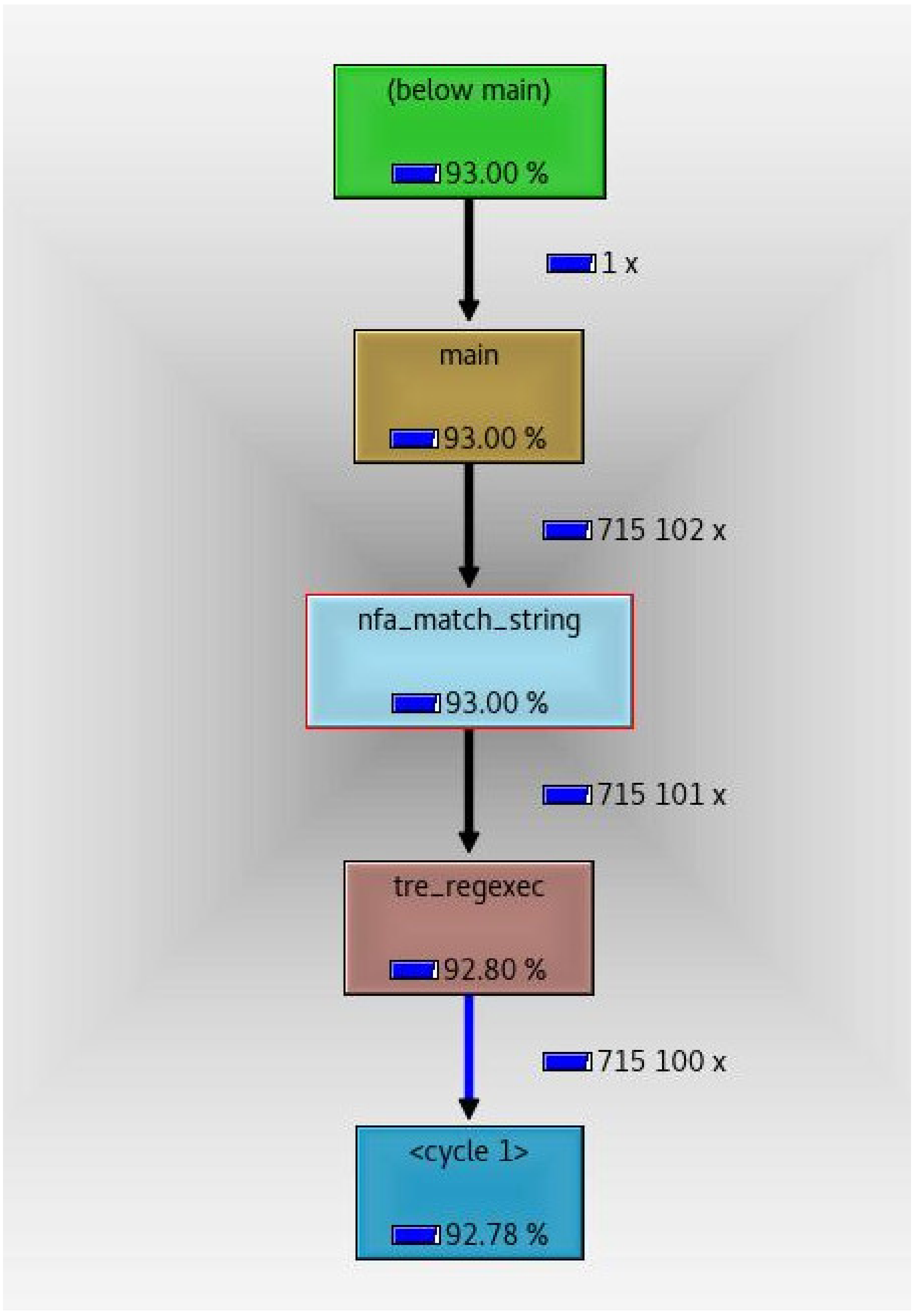


Figure 3.7: Sequential execution of our program when performing a tnfa match

With the help of the tool NVIDIA visual profiler[5] it is possible to analyse how the kernel is executed on the GPU device. In figure 3.8 we can see that the data and the kernel are executed on different streams. It is possible to see that the kernels are executing concurrently with data transfer: the data used in kernel 2 is transferred during the execution of kernel 1, demonstrated by the yellow colour on the figure. The log file used in this example is 1MB long.

Another important metric is the time it takes to convert a regular expression to a TNFA. Taking the regular expression *(a*)def* it takes 28 μs to convert to TNFA (figure 2.4). Then, the conversion to TDFA (figure 2.5) and the respective transition table to use in the GPU is computed in 48 μs. The total time spend to convert the regular expression in question to the format selected to use in the GPU is 72 μs.

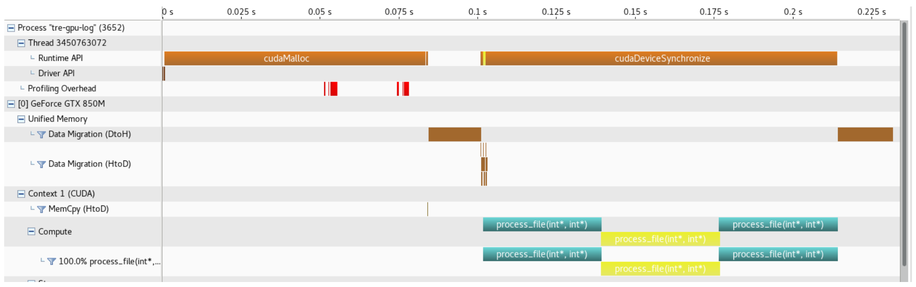


Figure 3.8: NVIDIA visual profiler analysis when executing GPU kernel responsible for match and search

It is very important to take in consideration the hardware environment where the GPU device is installed. On higher temperatures we can get unexpected behaviour from the device. It is possible to obtain this information from the system by running nvidia-smi[6]. This tool can give us more relevant information such as the memory usage. The output of our setup can be found in figure 3.9.

3.5 Conclusions

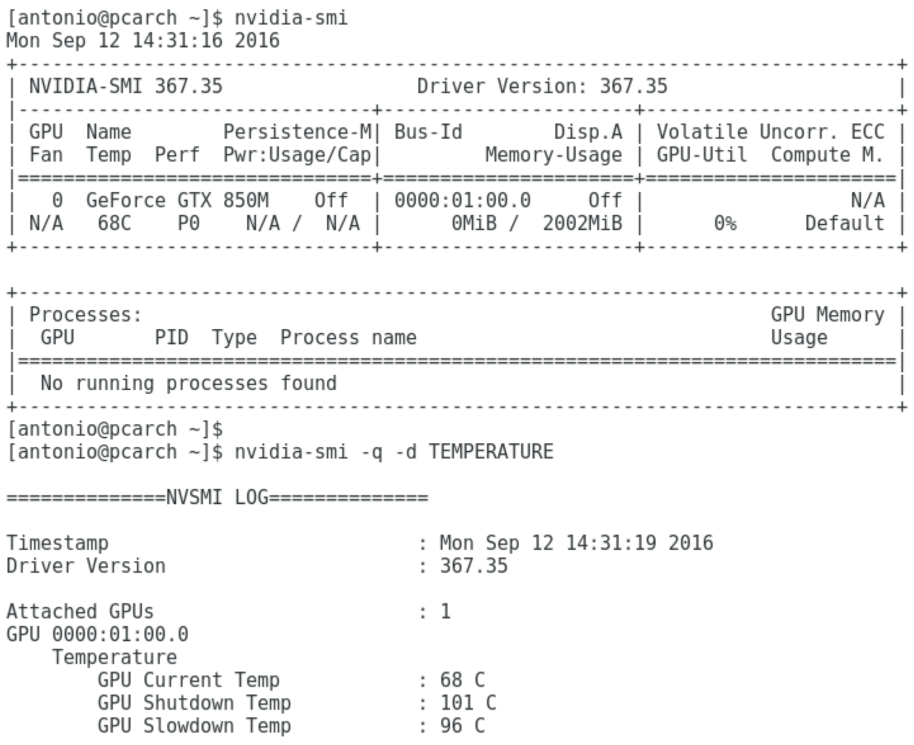
In this chapter we have presented the algorithm to implement regular matching with capture group support with the help of a GPU device. We have described techniques to improve data transfer to the GPU as transposing the data and used shared memory to avoid no-coalesced access.

We have also presented the use of unified memory and streams together to improve data transfer between CPU and GPU. The use of different streams has allowed us overlapping data transfers and kernel executions. It has been identified that the original sequential task can achieve a maximum speedup improvement of 14×, supported by Amdahl's law.

In next chapter 4 we present a comparison with common tools used to search and extract capture groups from log files using regular expressions.

[5]see `https://developer.nvidia.com/nvidia-visual-profiler`

[6]https://developer.nvidia.com/nvidia-system-management-interface



```
[antonio@pcarch ~]$ nvidia-smi
Mon Sep 12 14:31:16 2016
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 367.35                 Driver Version: 367.35                    |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  GeForce GTX 850M    Off  | 0000:01:00.0     Off |                  N/A |
| N/A   68C    P0    N/A /  N/A |      0MiB /  2002MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                       GPU Memory |
|  GPU       PID  Type  Process name                               Usage      |
|=============================================================================|
|  No running processes found                                                 |
+-----------------------------------------------------------------------------+
[antonio@pcarch ~]$
[antonio@pcarch ~]$ nvidia-smi -q -d TEMPERATURE

==============NVSMI LOG==============

Timestamp                           : Mon Sep 12 14:31:19 2016
Driver Version                      : 367.35

Attached GPUs                       : 1
GPU 0000:01:00.0
    Temperature
        GPU Current Temp            : 68 C
        GPU Shutdown Temp           : 101 C
        GPU Slowdown Temp           : 96 C
```

Figure 3.9: nvidia-smi generic and temperature query

4 Results

4.1 Introduction

In this chapter we present the results of running our implementation on real log files, obtained from a server running VoIP and email services on the cloud. We compare the results obtained from the regular expressions with current tools used by most of the system administrators, like GNU sed [11] or GNU awk [12]. We choose these two tools because of the feature of extracting group matches from the regular expressions, for example, we did not compare against GNU grep [15] because of the lack of this feature on that tool. Finally we discuss the obtained results.

4.2 Preliminary results

During the search phase we found a performance comparison for regular expression engines[1] from where it is possible to download the testing environment and reproduce the results with our setup. Basically, the test consist on running a set of regular expressions against the same text file. We decided to run our implementation with the same regular expressions and source file, the results are presented in table 4.1.

[1]Performance comparison of regular expression engines `http://sljit.sourceforge.net/regex_perf.html`

Table 4.1: Results of performance regular expression engines

Regular expression	**PCRE**	**PCRE-DFA**	**TRE**	**Oniguruma**	**RE2**	**PCRE-JIT**	**MyGPU**
`Twain`	2 ms	7 ms	234 ms	13 ms	1 ms	12 ms	640 ms
`(?i)Twain`	43 ms	63 ms	289 ms	78 ms	63 ms	13 ms	840 ms
`[a-z]shing`	322 ms	498 ms	379 ms	12 ms	90 ms	12 ms	846 ms
`Huck[a-zA-Z]+|Saw[a-zA-Z]+`	14 ms	15 ms	375 ms	30 ms	56 ms	2 ms	745 ms
`\b\w+nn\b`	480 ms	689 ms	662 ms	519 ms	46 ms	71 ms	744 ms
`[a-q][^u-z]{13}x`	379 ms	1373 ms	985 ms	29 ms	2015 ms	1 ms	941 ms
`Tom|Sawyer|Huckleberry|Finn`	20 ms	21 ms	645 ms	33 ms	57 ms	20 ms	841 ms
`(?i)Tom|Sawyer|Huckleberry|Finn`	242 ms	268 ms	965 ms	274 ms	77 ms	59 ms	1342 ms
`.{0,2}(Tom|Sawyer|Huckleberry|Finn)`	3354 ms	2498 ms	1710 ms	60 ms	50 ms	216 ms	1348 ms
`.{2,4}(Tom|Sawyer|Huckleberry|Finn)`	3332 ms	2887 ms	2480 ms	56 ms	49 ms	251 ms	1740 ms
`Tom.{10,25}river|river.{10,25}Tom`	45 ms	59 ms	379 ms	56 ms	60 ms	10 ms	750 ms
`[a-zA-Z]+ing`	745 ms	1095 ms	499 ms	544 ms	89 ms	52 ms	741 ms
`\s[a-zA-Z]{0,12}ing\s`	319 ms	469 ms	717 ms	49 ms	65 ms	70 ms	739 ms
`([A-Za-z]awyer|[A-Za-z]inn)\s`	725 ms	735 ms	687 ms	125 ms	76 ms	27 ms	742 ms
`["’][^"’]{0,30}[?!\.]["’]`	40 ms	58 ms	445 ms	56 ms	57 ms	8 ms	641 ms

We can see that our implementation shows in general a slower behaviour than the regular expression engines presented in this test suite. The regular expression engines used in the test can be classified in two types according to [60], performance oriented engines and balanced engines. Our implementation is based on TRE so it is considered to be a balanced engine and by definition it is slower than the oriented regular expression engine, excepted when using non pathological patterns. Other possibility is that the file size is too small for us to notice the improvement presented in this study. The file size used in this test was of 16MB. When using our implementation we have to add extra time to compute the entire TDFA as seen in chapter 2 and the time that takes to copy data between CPU and GPU. We will see later that we can get better results for files with larger size, especially with files with size above 100MB, which are common in logs.

4.3 Experimental evaluation

To make our tests as realistic as possible to the real world we have used logs obtained from a server running VoIP and email services. Two regular expressions were used to search and match patterns from the log files, in listing 4.3 we can find data that correspond to the content data in log files with prefix "freeswitch.log" (related to VoIP records) and in listing 4.4 correspond to the content data of files with prefix "syslog" (related to general system logs, including e-mail logs).

The first regular expression (listing 4.1), on a positive match will extract the information about the username, request client IP address and server bind address from the VoIP software running on the server where the logs were fetched. It should be noticed that this regular expression will only produce results on the log file with prefix named "freeswitch.log". The regular expression contains 92 states when converted to a NFA, and finally, when converted to DFA, it contains 66 states. This expression contains 4 group matches and will extract 4 variables on a positive match.

The second regular expression (listing 4.1), on a positive match with all rejected email messages, will extract email id, the hostname that sent the email, the field from and the reject message. As the previous regular expression, this expression will only produce results when matched against files with prefix "syslog". The regular expression contains 89 states when converted to a NFA, and finally, when converted to DFA, it contains 58 states. This expression contains 5 group matches and will extract 5 variables on a positive match.

Listing 4.1: Regular expression 1

```
^([0-9]+\-[0-9]+\-[0-9]+ [0-9]+:[0-9]+:[0-9]+\.[0-9]+).*sofia_reg.*
    ↪ REGISTER.*profile ''(.*)'' for \[(.*)\] from ip (.*)$
```

Listing 4.2: Regular expression 2

```
^.*exim\[[0-9]+\]:\s*([0-9]+\-[0-9]+\-[0-9]+ [0-9]+:[0-9]+:[0-9]+)\s
    ↪ *(.*) H=(.*) F=<(.*)> rejected after DATA: (.*).*$
```

Listing 4.3: Few lines of freeswitch log

```
2015-04-13 10:08:56.388280 [INFO] mod_commands.c:6371 endpoint.lua:
    ↪ SIP (REGISTER) on sofia profile 'dinamico_udp' for [513
    ↪ @midominio.es] from ip 10.131.100.10
```

```
2015-04-13 10:08:56.868284 [WARNING] sofia_reg.c:1744 SIP auth
    ↪ challenge (REGISTER) on sofia profile 'dinamico_udp' for [112
    ↪ @midominio.es] from ip 10.100.140.14
741b05d0-d22c-4df8-b1e0-26a09d88cfce 2015-04-13 10:08:56.868284 [
    ↪ NOTICE] switch_channel.c:1075 New Channel sofia/dinamico_udp
    ↪ /337@midominio.es[741b05d0-d22c-4df8-b1e0-26a09d88cfce]
```

Listing 4.4: Few lines of syslog log

```
Apr 11 00:17:03 server1 named[1971]: error (network unreachable)
    ↪ resolving 'mail.com.tw/MX/IN': 2001:500:1::803f:235#53
Apr 11 00:17:02 server1 exim[11176]: 2015-04-11 00:17:02 1YghEU-0002
    ↪ uG-BY <= root@dominio.es U=root P=local S=777
Apr 11 00:17:02 server1 exim[11179]: 2015-04-11 00:17:02 1YghEU-0002
    ↪ uG-BY remote host aaaaddress is the local host: server1.dominio
    ↪ .es a
Apr 11 00:17:02 server1 exim[11179]: 2015-04-11 00:17:02 1YghEU-0002
    ↪ uG-BY Completed
Apr 11 00:18:47 server1 exim[18663]: 2015-04-11 00:18:47 1YghGA-0004
    ↪ r1-HT H=(mta52.net) [10.100.175.20] F=<noreply@median.net>
    ↪ rejected after DATA: This message has been classified as SPAM
    ↪ and as such has been rejected. Spam detection software, running
    ↪  on the system "server1", has
```

4.4 Performance comparison with current tools

To compare our implementation we have chosen current tools that also make use of regular expressions and, more important, support group match capture like GNU sed [11] or GNU awk [12]. Other tool that is used very often by system administrator, GNU grep [15], has been discarded from our tests because of the lack of group capture.

In table 4.2 and figure 4.1 we can see the different results obtained when parsing the log files of different sizes. The results presented in table 4.2 show us the time and linear speedup when running version implemented in the CPU and the GPU version. The version implemented in the CPU corresponds to the function execution found in figure **??**, this way we can check that the maximum speedup obtained respects the Amdahl's law. We can see that for smaller log files the CPU version is faster than the GPU version. When the files are larger, the results when running the GPU version tend to be better and we can gain up to a speedup of 9.4× when comparing to the CPU version. The speedup gain was compared with our sequential implementation because of the control over the all program and understand of the sequential code execution. Also in figure 4.1 we can see that our sequential implementation presents results very similar to GNU sed [11] and GNU awk [12].

We have better results when running on files with larger size because of the type of parallelism used in our program, the parallelism is done over the data, that is, the lines are processed in parallel by multiple threads in the GPU. For smaller files the results are better in the CPU, this is mainly because we have to take in account the time needed to copy data to the GPU, like the transition table. For larger files, the time to copy data between host and CPU is hidden by the huge number of threads working together in multiple lines of the file. In our setup the kernel can process up to 1024 lines at a time.

Table 4.2: Results when running CPU only version versus GPU. Time in seconds. The value for re1 see listing 4.1. The value for r2 see listing 4.2

filename	regex	**CPU time**	**GPU time**	**Speedup**
freeswitch.log_5MB	re1	201	453	0.4
freeswitch.log_5MB	re2	171	303	0.6
freeswitch.log_100MB	re1	1747	712	2.5
freeswitch.log_100MB	re2	3119	612	5.1
freeswitch.log_500MB	re1	7628	1989	3.8
freeswitch.log_500MB	re2	15315	1873	8.2
freeswitch.log_1GB	re1	15856	3810	4.2
freeswitch.log_1GB	re2	32371	3632	8.9
syslog_5MB	re1	117	365	0.3
syslog_5MB	re2	192	308	0.6
syslog_100MB	re1	1077	713	1.5
syslog_100MB	re2	3410	611	5.6
syslog_500MB	re1	4952	2219	2.2
syslog_500MB	re2	16274	1874	8.7
syslog_1GB	re1	10044	3711	2.7
syslog_1GB	re2	33397	3534	9.4

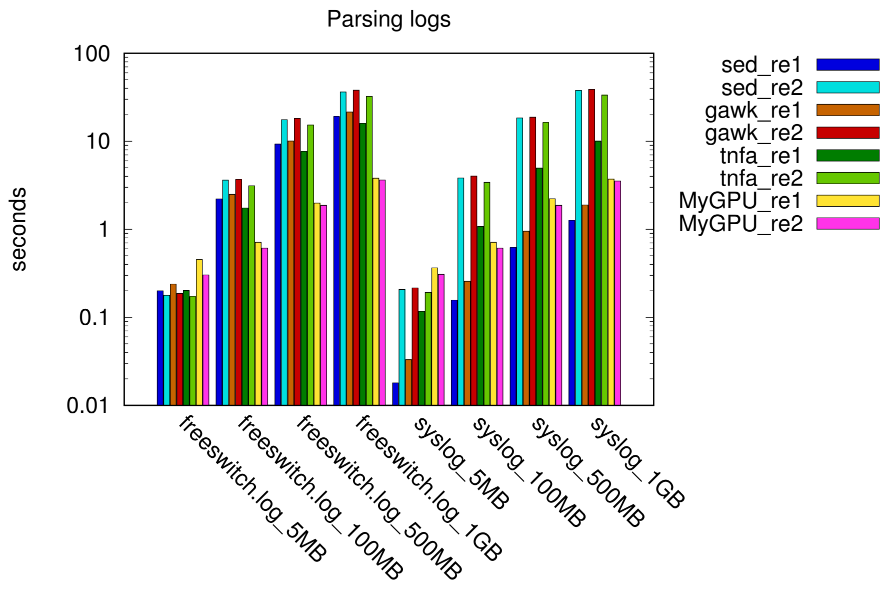


Figure 4.1: Results when parsing real log files using current tools sed, gawk and our implementation

From the figure 4.1 we can see that our implementation is faster than the current tools GNU sed [11] or GNU awk [12], especially when the file size is greater than or equal to 100MB. Another important analysis from the graph is that when the match does not produce any results, the running time is faster in the CPU only implementations.

4.5 Discussion

From the previous results we notice that our implementation is faster than the current tools GNU sed [11] or GNU awk [12] when the file size that is processed is greater than or equal to 100MB. This behaviour is expected because of the time needed to compute the entire TDFA and copy data between CPU and GPU is more evident when the file size is smaller. With larger log files the time that this process takes is hidden in the global time to process the entire file.

We observe that the complexity of the regular expressions can affect the performance of searching and extracting data from the log files: simpler expressions tend to be faster, with large difference in CPU implementations.

4.6 Conclusions

In this chapter we have shown that our implementation has a performance degradation when searching in small files, but it can outperform the CPU tools when the log files are larger. We have obtained a maximum speedup improvement of 9.4× when compared with our sequential implementation.

Finally, we can conclude that the use of GPU and regular expressions with capture group support can be used together and the obtained results show that the GPU version is faster when comparing to current CPU programs.

5 Conclusion

5.1 Summary

We have conducted a study on how to use regular expression with GPU devices. The regular expression engine does not only have support for matching task, but is also capable of extracting information from the capture groups that are present on the regular expression. This solves a limitation in current state of the art, where the use of GPU devices is mainly for matching purpose tasks. For the first time it is now possible to use regular expressions with capture group support with the help of GPU devices.

During the execution of this study multiple tests were run and we have presented an implementation where this task can be performed with success. The results from the execution of the regular expression on the GPU device are better for larger log files. In the best case we can obtain a speedup of $9\times$ faster than a serial implementation. Better results could be obtained using a higher performance GPU card like, for example, the second-generation Maxwell architecture codenamed GM204. Instead, we used the first-generation Maxwell architecture codenamed GM107, which is the one that was available for this research.

The results of this work have been used in the real world in a technology company, where the study of regular expression permitted us to design an efficient log file module processor. The implementation is also freely available on GitHub[1].

5.2 Contributions

In the present work we have provided the following contributions:

- We have conducted a full study on regular expression implemented mechanism and engines.

[1] `https://github.com/apsilva/tre-gpu-log`

- We have demonstrated that regular expressions with capture group support can be use with GPU devices.
- It is important to notice that this work is suitable for processing log files which size is greater than or equal to 100MB. With smaller files the CPU-based current tools outperform our implementation.

5.3 Future work

As future work, it could be interesting to implement a similar solution using another co-processor like the Intel PHI and compare the results against the GPU implementation. There have been works where the regular expression matching task is compared between these two architectures [61].

The implementation was done using the CUDA programming language. Another approach is to implement the solution recurring to OpenCL in order to support a major number of GPU cards and not only NVIDIA cards. Or conduct a study to implement a similar solution using distributed systems.

Another feature that could be implemented to improve the support of regular expression matching and extract data using GPUs is the implementation of regular expression backtracking, which will allow not only to capture the pattern that is been searched but use it to execute more complex searches.

Bibliography

[1] Nvidia, "Whitepaper: NVIDIA GeForce GTX 750 Ti," pp. 1–11, 2014.

[2] "CUDA C Best Practices Guide," http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/. [Online]. Available: http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/

[3] "CUDA library trove," https://github.com/bryancatanzaro/trove. [Online]. Available: https://github.com/bryancatanzaro/trove

[4] "Common Log Format," https://en.wikipedia.org/wiki/Common_Log_Format. [Online]. Available: https://en.wikipedia.org/wiki/Common_Log_Format

[5] "Syslog," https://en.wikipedia.org/wiki/Syslog. [Online]. Available: https://en.wikipedia.org/wiki/Syslog

[6] B. J. Jansen, "Search log analysis: What it is, what's been done, how to do it," *Libr. Inf. Sci. Res.*, vol. 28, no. 3, pp. 407–432, 2006.

[7] R. Marty, "Cloud application logging for forensics," *Proc. 2011 ACM Symp. Appl. Comput. - SAC '11*, p. 178, 2011. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1982185.1982226

[8] X. Shu, J. Smiy, D. Daphne Yao, and H. Lin, "Massive distributed and parallel log analysis for organizational security," *2013 IEEE Globecom Work. (GC Wkshps)*, pp. 194–199, 2013. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6824985

[9] N. Goel, "Analyzing Users Behavior from Web Access Logs using Automated Log Analyzer Tool," *Int. J. Comput. Appl.*, vol. 62, no. 2, pp. 29–33, 2013.

[10] J. E. F. Friedl, *Mastering Regular Expressions*, 3rd ed. O'Reilly Media, 2006. [Online]. Available: http://www.ncbi.nlm.nih.gov/pubmed/5993855

[11] "GNU sed," http://www.gnu.org/software/sed. [Online]. Available: http://www.gnu.org/software/sed/

[12] "GNU awk," http://www.gnu.org/software/gawk. [Online]. Available: http://www.gnu.org/software/gawk

[13] M. Sipser, *Introduction to the Theory of Computation*, internatio ed. Thomson South-Western, 2013.

[14] J. E. Hopcroft, R. Motwani, and J. D. Ullman, *Automata Theory , Languages , and Languages , and Computation.* Pearson, 2006.

[15] "GNU grep," https://www.gnu.org/software/grep/. [Online]. Available: https://www.gnu.org/software/grep/

[16] "Google RE2," https://github.com/google/re2. [Online]. Available: https://github.com/google/re2

[17] G. Xing, "A simple way to construct NFA with fewer states and transitions," in *ACM-SE 42 Proc. 42nd Annu. Southeast Reg. Conf.*, 2004, pp. 214–218.

[18] ——, "Minimized Thompson NFA," *Int. J. Comput. Math.*, vol. 81, no. 9, pp. 1097–1106, 2004. [Online]. Available: http://www.tandfonline.com/doi/abs/10.1080/00207160412331272153

[19] M. Becchi and P. Crowley, "Extending finite automata to efficiently match Perl-compatible regular expressions," *Proc. 2008 ACM Conex. Conf. - Conex. '08*, pp. 1–12, 2008. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1544012.1544037

[20] D. Ficara, S. Giordano, and G. Procissi, "An improved DFA for fast regular expression matching," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 5, p. 29, 2008.

[21] H. Hyyrö, "Improving the bit-parallel NFA of Baeza-Yates and Navarro for approximate string matching," *Inf. Process. Lett.*, vol. 108, no. 5, pp. 313–319, nov 2008. [Online]. Available: http://dx.doi.org/10.1016/j.ipl.2008.05.026http://linkinghub.elsevier.com/retrieve/pii/S0020019008001853

[22] L. Yang, P. Manadhata, W. Horne, P. Rao, and V. Ganapathy, "Fast submatch extraction using OBDDs," in *Proc. eighth ACM/IEEE Symp. Archit. Netw. Commun. Syst. - ANCS '12*, no. October. New York, New York, USA: ACM Press, 2012, p. 163.

[23] S. Memeti and S. Pllana, "PaREM: A Novel Approach for Parallel Regular Expression Matching," in *2014 IEEE 17th Int. Conf. Comput. Sci. Eng.* IEEE, dec 2014, pp. 690–697. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=7023656

[24] R. Sin'ya, K. Matsuzaki, and M. Sassa, "Simultaneous Finite Automata: An Efficient Data-Parallel Model for Regular Expression Matching," *2013 42nd Int. Conf. Parallel Process.*, pp. 220–229, may 2014. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6687355http://arxiv.org/abs/1405.0562

[25] R. Antonello, S. Fernandes, D. Sadok, J. Kelner, and G. Szabó, "Design and optimizations for efficient regular expression matching in DPI systems," *Comput. Commun.*, vol. 61, pp. 103–120, 2015. [Online]. Available: http://linkinghub.elsevier.com/retrieve/pii/S014036641500002X

[26] R. Cox, "Regular Expression Matching Can Be Simple And Fast," https://swtch.com/%7Ersc/regexp/regexp1.html. [Online]. Available: https://swtch.com/$\sim$rsc/regexp/regexp1.html

[27] ——, "Regular Expression Matching: the Virtual Machine Approach," https://swtch.com/%7Ersc/regexp/regexp2.html. [Online]. Available: https://swtch.com/$\sim$rsc/regexp/regexp2.html

[28] ——, "Regular Expression Matching in the Wild," https://swtch.com/%7Ersc/regexp/regexp3.html. [Online]. Available: https://swtch.com/$\sim$rsc/regexp/regexp3.html

[29] M. Becchi, “Regular Expression Processor,” http://regex.wustl.edu/index.php/Main_Page. [Online]. Available: http://regex.wustl.edu/index.php/Main_Page

[30] K. Thompson, “Programming Techniques: Regular expression search algorithm,” *Commun. ACM*, vol. 11, no. 6, pp. 419–422, jun 1968. [Online]. Available: http://portal.acm.org/citation.cfm?doid=363347.363387

[31] F. Moore, “On the Bounds for State-Set Size in the Proofs of Equivalence Between Deterministic, Nondeterministic, and Two-Way Finite Automata,” *IEEE Trans. Comput.*, vol. C-20, no. 10, pp. 1211–1214, oct 1971. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1671701

[32] V. Laurikari, “NFAs with tagged transitions, their conversion to deterministic automata and application to regular expressions,” in *Proc. Seventh Int. Symp. String Process. Inf. Retrieval. SPIRE 2000.* IEEE Comput. Soc, 2000, pp. 181–187. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.29.4395http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=878194

[33] ——, “Efficient submatch addressing for regular expressions,” Master, Helsinki University of Technology, 2001.

[34] A. Karper, “Efficient regular expressions that produce parse trees,” Master of Science, University of Bern, 2014.

[35] “Haskell: regex-tdfa,” http://hackage.haskell.org/package/regex-tdfa. [Online]. Available: http://hackage.haskell.org/package/regex-tdfa

[36] “OpenCL,” https://developer.nvidia.com/opencl. [Online]. Available: https://developer.nvidia.com/opencl

[37] “CUDA,” http://www.nvidia.com/object/cuda_home_new.html. [Online]. Available: http://www.nvidia.com/object/cuda_home_new.html

[38] G. Vasiliadis, S. Antonatos, M. Polychronakis, E. Markatos, and S. Ioannidis, “Gnort: High performance network intrusion detection using graphics processors,” *Recent Adv. Intrusion Detect.*, pp. 116–134, 2008. [Online]. Available: http://www.springerlink.com/index/g2w54q11130r7126.pdf

[39] G. Vasiliadis, M. Polychronakis, S. Antonatos, E. P. Markatos, and S. Ioannidis, “Regular expression matching on graphics hardware for intrusion detection,” *Proc. 12th Int. Symp. Recent Adv. Intrusion Detect.*, pp. 265–283, 2009.

[40] M. Onsjö and Y. Aono, “Online Approximate String Matching with CUDA,” *Technology*, pp. 1–4, 2009. [Online]. Available: http://pds13.egloos.com/pds/200907/26/57/pattmatch-report.pdf

[41] N. Cascarano, P. Rolando, F. Risso, and R. Sisto, “iNFAnt: NFA Pattern Matching on GPGPU Devices,” *SIGCOMM Comput. Commun. Rev.*, vol. 40, no. 5, pp. 20–26, 2010. [Online]. Available: http://doi.acm.org/10.1145/1880153.1880157

[42] S. Han, K. Jang, K. Park, and S. Moon, “PacketShader: a GPU-Accelerated Software Router,” in *Proc. ACM SIGCOMM 2010 Conf. SIGCOMM - SIGCOMM ’10*, vol. 40, no. 4. New York, New York, USA: ACM Press, 2010, p. 195. [Online]. Available: http://portal.acm.org/citation.cfm?id=1851207http://portal.acm.org/citation.cfm?doid=1851182.1851207

[43] G. Vasiliadis, M. Polychronakis, and S. Ioannidis, “Parallelization and characterization of pattern matching using GPUs,” *2011 IEEE Int. Symp. Workload Charact.*, pp. 216–225, 2011.

[44] L. Wang, S. Chen, Y. Tang, and J. Su, “Gregex: GPU based high speed regular expression matching engine,” *Proc. - 2011 5th Int. Conf. Innov. Mob. Internet Serv. Ubiquitous Comput. IMIS 2011*, pp. 366–370, 2011.

[45] Y. Zu, M. Yang, Z. Xu, L. Wang, X. Tian, K. Peng, and Q. Dong, “GPU-based NFA implementation for memory efficient high speed regular expression matching,” *Proc. ACM SIGPLAN Symp. Princ. Pract. Parallel Program. PPOPP*, pp. 129–139, 2012. [Online]. Available: http://www.scopus.com/inward/record.url?eid=2-s2.0-84863362945&partnerID=40&md5=781c686004f691a5a1cc55a66f129263

[46] P. Gómez Nieto, “Clasificación de tráfico TCP / IP mediante dispositivos GPU,” Master, Universidad Autónoma de Madrid, 2012.

[47] R. Leira Osuna, “Clasificación de flujos en 10 Gbps Ethernet mediante Intel DPDK y GPUS,” Trabajo fin de Grado, Universidad Autónoma de Madrid, 2013.

[48] X. Yu and M. Becchi, “GPU acceleration of regular expression matching for large datasets: : Exploring the Implementation Space,” in *Proc. ACM Int. Conf. Comput. Front. - CF ’13*. New York, New York, USA: ACM Press, 2013, p. 1. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2482767.2482791http://dl.acm.org/citation.cfm?id=2482767.2482791

[49] C.-h. Lin, C.-h. Liu, L.-S. Chien, and S.-C. Chang, “Accelerating Pattern Matching Using a Novel Parallel Algorithm on GPUs,” *IEEE Trans. Comput.*, vol. 62, no. 10, pp. 1906–1916, oct 2013. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6338923

[50] H. Sasakawa, Hirohito and Arimura, “Faster Multiple Pattern Matching System on GPU based on Bit-Parallelism,” in *18th Work. Synth. Syst. Integr. Mix. Inf. Technol.*, 2013.

[51] Mr.Gaurav K. Bhamare and P. S. Banait, “Parallelization of Multipattern Matching on GPU,” *Int. J. Electron. Commun. Soft Comput. Sci. Eng.*, vol. 3, no. 3, 2014.

[52] S. Ponnemkunnath and R. C. Joshi, “Efficient Regular Expression Pattern Matching on Graphics Processing Units,” 2011, vol. 168, pp. 92–101. [Online]. Available: http://www.springerlink.com/content/m811165163585182/abstract/http://link.springer.com/10.1007/978-3-642-22606-9_13

[53] X. J. A. Bellekens, C. Tachtatzis, R. C. Atkinson, C. Renfrew, and T. Kirkham, “GLoP: Enabling Massively Parallel Incident Response Through GPU Log Processing,” in *Proc. 7th Int. Conf. Secur. Inf. Networks - SIN ’14*. New York, New York, USA: ACM Press, 2014, pp. 295–301. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2659651.2659700

[54] X. Bellekens, R. C. Atkinson, C. Renfrew, and T. Kirkham, “Investigation of GPU-based Pattern Matching,” *14th Annu. Post Grad. Symp. Converg. Telecommun. Netw. Broadcast.*, p. 5, 2013. [Online]. Available: http://www.cms.livjm.ac.uk/pgnet2013/Proceedings/papers/1569777259.pdf

[55] "CUDA grep," http://www.cs.cmu.edu/afs/cs/academic/class/15418-s12/www/competition/bkase.github.com/CUDA-grep/finalreport.html. [Online]. Available: http://www.cs.cmu.edu/afs/cs/academic/class/15418-s12/www/competition/bkase.github.com/CUDA-grep/finalreport.html

[56] "Unified Memory in CUDA 6," https://devblogs.nvidia.com/parallelforall/unified-memory-in-cuda-6/. [Online]. Available: https://devblogs.nvidia.com/parallelforall/unified-memory-in-cuda-6/

[57] "Unified Memory Programming in CUDA," http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#um-unified-memory-programming-hd. [Online]. Available: http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#um-unified-memory-programming-hd

[58] R. Landaverde, T. Zhang, A. K. Coskun, M. Herbordt, Tiansheng Zhang, A. K. Coskun, and M. Herbordt, "An investigation of Unified Memory Access performance in CUDA," in *2014 IEEE High Perform. Extrem. Comput. Conf.* IEEE, sep 2014, pp. 1–6. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=7040988

[59] "amdhl's law," https://en.wikipedia.org/wiki/Amdahl%27s_law. [Online]. Available: https://en.wikipedia.org/wiki/Amdahl%27s_law

[60] "regex compare," http://sljit.sourceforge.net/regex_compare.html. [Online]. Available: http://sljit.sourceforge.net/regex_compare.html

[61] T. T. Tran, Y. Liu, and B. Schmidt, "Bit-parallel approximate pattern matching: Kepler GPU versus Xeon Phi," *Parallel Comput.*, vol. 54, pp. 128–138, may 2016. [Online]. Available: http://linkinghub.elsevier.com/retrieve/pii/S0167819115001477

Appendices

A Appendix

A.1 Program help

```
Usage: ./trelog -f <filename> -r <regular expresion> [-rndgo]
        -f          path to the file
        -r          regular expression
        -n          enable tnfa (default: disabled)
        -d          enable tdfa cpu (default: disabled)
        -g          disable tdfa gpu (default: enabled)
        -o          output. json - for format json, csv (default) - for
    ↪ csv format, none - match only
```