A Massively Parallel Regular Expression and String Matching Engine for Commodity Hardware

Dimitris Deyannis

Thesis submitted in partial fulfillment of the requirements for the

Master of Science degree in Computer Science

University of Crete
School of Sciences and Engineering
Computer Science Department
University Campus, Voutes, Heraklion, GR-70013, Greece

Thesis Advisors: Prof. *Evangelos Markatos*, Dr. *Sotiris Ioannidis*

This work was partially supported by **Institute of Computer Science, Foundation of Research and Technology Hellas**

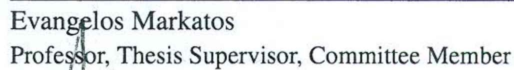
UNIVERSITY OF CRETE
COMPUTER SCIENCE DEPARTMENT

A Massively Parallel Regular Expression and String Matching Engine for Commodity Hardware

Thesis submitted by
Dimitris Deyannis
in partial fulfillment of the requirements for the
Master of Science degree in Computer Science

THESIS APPROVAL

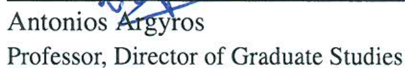
Author: ______________________
Dimitris Deyannis



Committee approvals: ______________________
Evangelos Markatos
Professor, Thesis Supervisor, Committee Member

____
Sotiris Ioannidis
Principal Researcher, Thesis Advisor, Committee Member

____
Ioannis Tsamardinos
Associate Professor, Committee Member



Departmental approval: ______________________
Antonios Argyros
Professor, Director of Graduate Studies

Heraklion, February 2017

Abstract

String pattern matching is one of the most studied fields in the research community, mainly due to the fact that it can be used and applied in various and diverse fields, such as computer science, computational biology, chemistry and others. Since 1970, researchers aim to develop algorithms for efficient string searching and until today, the problem of pattern matching remains a popular area for studying.

Recently, in order to cope with the ever advancing technology, parallel computing platforms –such as CUDA and OpenCL– offer general purpose programming using traditional CPUs, hardware accelerators and GPUs.

In this work, we propose a framework for string pattern matching on parallel hardware architectures. Using CUDA and OpenCL, our framework offers uniform execution on any processor available in a system. The framework provides an abstraction layer to the user –without penalizing the performance– and it is provided as either a C- or Java-like API. Except for simple string matching, our engine supports the use of multiple regular expressions that comply with the POSIX ERE standard. Specifically, we achieve the simultaneous matching of multiple simple strings and binary patterns against multiple data streams as input. Finally, our framework manages to simultaneously match large sets of regular expressions against multiple data streams.

The performance evaluation shows that our massively parallel engine can achieve up to 21 times performance increase when processing simple strings and up to 15 times when processing regular expressions, compared to the CPU versions of both matching algorithms. Specifically, the engine can sustain string matching throughput up to 65Gbits/s and regular expression matching throughput up to 50Gbits/s.

Περίληψη

Το ταίριασμα αλφαριθμητικών προτύπων (string pattern matching) είναι ένα πεδίο έρευνας στο οποίο έχει αφιερώσει σημαντικό ποσοστό έρευνας η επιστημονική κοινότητα ανά τα χρόνια. Ήδη, από το 1970, επιστήμονες από διάφορους φορείς και ερευνητικά πεδία, προσπαθούν συνεχώς να αναπτύξουν αλγορίθμους, τόσο έξυπνους όσο και αποδοτικούς. Ακόμα όμως, το πρόβλημα της αντιστοίχισης αλφαριθμητικών προτύπων αποτελεί ανοιχτό πεδίο σκέψης και μελέτης. Ο λόγος της ραγδαίας αυτής δημοτικότητας της αντιστοίχισης αλφαριθμητικών προτύπων στην επιστημονική κοινότητα, είναι η ευρεία χρήση και εφαρμογή της σε πολλές και ποικίλες περιοχές, όπως για παράδειγμα στην πληροφορική, στη βιοπληροφορική, στην υπολογιστική βιοϊατρική και άλλες.

Πρόσφατα, καθώς η τεχνολογία συνεχώς εξελίσσεται, η χρήση των παράλληλων επεξεργαστών έχει αποτελέσει σημαντικό παράγοντα για την ανάπτυξη όλο και πιο γρήγορων και αποδοτικών συστημάτων. Ο προγραμματισμός αυτών των παράλληλων επεξεργαστών –είτε αυτοί είναι πολυπύρηνοι επεξεργαστές (CPUs) είτε είναι επεξεργαστές γραφικών γενικού σκοπού (GPGPUs)– βασίζεται σε πλατφόρμες που επιτρέπουν στο χρήστη την εποπτεία και τον προγραμματισμό τους. Σε αυτή τη δουλειά, οι πλατφόρμες που χρησιμοποιούνται ονομάζονται CUDA και OpenCL. Συγκεκριμένα, η CUDA απευθύνεται σε επεξεργαστές γραφικών γενικού σκοπού της εταιρίας NVIDIA, σε αντίθεση με την OpenCL, η οποία επιτρέπει τον προγραμματισμό οποιουδήποτε είδους επεξεργαστή.

Σε αυτή τη δουλειά, παρουσιάζουμε μία βιβλιοθήκη για αντιστοίχιση αλφαριθμητικών προτύπων που μέσω μιας αφηρημένης προγραμματιστικής διεπαφής, επιτρέπει την χρήση της σε κάθε είδους πολυπύρηνο επεξεργαστή. Πέρα από την αντιστοίχιση απλών αλφαριθμητικών προτύπων, η βιβλιοθήκη αυτή επιτρέπει τον εντοπισμό και το ταίριασμα προτύπων που προκύπτουν από κανονικές γραμματικές. Για αυτόν το σκοπό, αναπτύξαμε μία μηχανή παράλληλης αναζήτησης αλφαριθμητικών και κανονικών εκφράσεων με την χρήση πολυπύρηνων επεξεργαστών και καρτών γραφικών. Επιπλέον, η μηχανή μπορεί να πετύχει ταυτόχρονη αναζήτηση πολλών αλφαριθμητικών και κανονικών εκφράσεων σε είσοδο πολλαπλών δεδομένων με μία μόνο προσπέλαση αυτών.

Τέλος, η αξιολόγηση της απόδοσης του συστήματος αυτού, μέσω της βιβλιοθήκης που παρέχουμε, έδειξε ότι μπορεί να επιτύχει μέχρι και 21 φορές μεγαλύτερη απόδοση στην αναζήτηση απλών αλφαριθμητικών, καθώς και μέχρι 15 φορές μεγαλύτερη απόδοση στην αναζήτηση κανονικών εκφράσεων, σε σχέση με τις αντίστοιχες εκδόσεις των αλγορίθμων για κεντρικούς επεξεργαστές. Συγκεκριμένα, το σύστημά μας μπορεί να επιτύχει απόδοση έως και 65Gbits/s στην αναζήτηση αλφαριθμητικών και έως 50Gbits/s στην αναζήτηση κανονικών εκφράσεων.

Acknowledgments

First of all, I would like to thank my supervisor, Professor Evangelos Markatos, for his valuable guidance and all the constructive conversations we had. I also want to express my deepest gratitude to my advisor, Dr. Sotiris Ioannidis, for giving me the opportunity to work on so many different, challenging and interesting projects, over the past three years. His support and advice greatly contributed to my academic and technical growth. Moreover, I feel thankful to Dr. Giorgos Vasiliadis, for his guidance during my first steps of this academic journey and for setting the foundations of this work. Also, to Lazaros Koromilas for everything I learned from him throughout our collaboration.

My warmest regards to Evangelos Ladakis, Elias P. Papadopoulos, Giorgos Christou, Giorgos Tsirantonakis, Michalis Diamantaris, Konstantinos Kleftogiorgos, Eirini Degkleri, Kostas Solomos, Nick Christoulakis and all the other present and past members of the Distributed Computing System Laboratory, for their friendship, advice and commitment.

I am also indebted to Christos Papachistos for his technical support and Eva Papadogianaki, for bearing with me during the development of this work...

Finally, I want to thank my family and friends, back in my home-town, as well as Katerina Karagiannaki, for all their invaluable support and caring.

Contents

1 Introduction **1**
1.1 Motivation 2
1.2 Contributions 2
1.3 Publications 3
1.4 Outline 3

2 String Matching Algorithms **5**
2.1 Classification Based on Pattern Number 5
2.1.1 Single Pattern 5
2.1.2 Finite Number of Patterns 6
2.1.3 Infinite Number of Patterns 6
2.2 Other Classifications 6
2.2.1 Naive String Search 7
2.2.2 Finite State Automaton Based Search 7
2.2.3 Index Based Search 7
2.2.4 Exact String Search 8
2.2.5 Approximate String Search 8
2.3 The Aho-Corasick Algorithm 8

3 Regular Expressions **11**
3.1 Formal Definition 11
3.2 Syntax 12
3.2.1 Delimiters 12
3.2.2 Metacharacters 13
3.2.3 Character Classes 14
3.2.4 Standards 14
3.2.4.1 POSIX Basic (BRE) 15
3.2.4.2 POSIX Extended (ERE) 16
3.2.5 Lazy Quantifiers 16
3.2.6 Backreferences 16
3.3 Regular Expression Engine Internals 17
3.3.1 Regex-directed Engines 17
3.3.2 Text-directed Engines 17

4 Architectures and Parallel Computing **19**
4.1 Levels of Parallelism 19
4.1.1 Bit-level 19
4.1.2 Instruction-level 19
4.1.3 Task-level 20
4.1.4 Data Parallelism 20
4.2 Flynn's Taxonomy 21
4.2.1 SISD 22
4.2.2 SIMD 22
4.2.3 MISD 22
4.2.4 MIMD 22
4.2.5 Other Classifications 22
4.2.5.1 SIMT 22
4.2.5.2 SPMD 23
4.2.5.3 MPMD 23
4.3 Commodity Hardware 23
4.3.1 CPUs 23
4.3.2 Graphics Processing Units 24
4.3.3 Other Accelerators 25
4.4 Parallel Computing Platforms 25
4.4.1 CUDA 25
4.4.2 OpenCL 26
4.5 Programming Considerations 26

5 Implementation **27**
5.1 Design Overview 27
5.2 Data Gathering and Transferring 28
5.3 String Matching 31
5.3.1 Readable Binary Signature Support 31
5.3.2 DFA Representation 32
5.3.3 Input Processing 33
5.4 Regular Expression Matching 34
5.4.1 DFA Construction 34
5.4.2 Data Processing and Backtracking 34
5.5 String Assisted Regular Expressions 34
5.5.1 Searching Rules 35
5.5.1.1 Reporting and Events 36
5.6 API 36

6 Evaluation **39**
6.1 Experimental Testbed 39
6.2 Workloads 39
6.2.1 Synthetic ASCII Workload 40
6.2.2 Huge Regular Expressions Workload 40

6.3 DFA Properties . . . 41
6.3.1 Simple String DFAs . . . 41
6.3.2 Regular Expression DFAs . . . 42
6.4 Baseline . . . 44
6.5 Graphics Accelerators . . . 46
6.5.1 CUDA . . . 46
6.5.2 OpenCL . . . 49
6.6 OpenCL with Intel Processors . . . 51

7 Related Work **55**
7.1 Pattern matching applications . . . 55
7.2 String searching tools . . . 57

8 Conclusions and Future Work **59**
8.1 Summary of Contributions . . . 59
8.2 Future Work . . . 59
8.3 Conclusion . . . 60

List of Figures

2.1 Goto function 9

4.1 The four basic computer architectures, proposed by Flynn [41]. The first two figures (Figures 4.1(a) and 4.1(b)) present the architectures with a single data stream, while the remaining two (Figures 4.1(c) and 4.1(d)) demonstrate the architectures with multiple data streams. 21

4.2 The NVIDIA CUDA architecture comprised of multiprocessors, each one containing multiple stream processors and a set of various memory spaces. 24

5.1 The overall architecture of our system. The interaction between the four components (*modules*) is performed as follows: data chunks arrive from various input data streams to the *data transfer* module (1) that batches them and hands those batches to the two processing modules, named *string matching* and *regex matching* modules, respectively (2); when the processing is finished (3) the *data transfer* module returns the results to the *events module*, responsible for the match reporting (4). In addition, we present the runtimes and the underlying hardware. 28

5.2 Ring buffer overview. Data chunks arriving from various input streams are batched into ring buffer buckets. Once a bucket is complete, it is transferred to the discrete GPU via the PCIe bus. If the buckets are processed by the CPU, or an integrated GPU, the bucket transferring process is omitted since the integrated GPU shares the same physical address space with the CPU; thus, there is no need for transferring data to a dedicated DRAM. 29

5.3 Overview of a ring buffer bucket. The bucket contains the batch of data, concatenated into a single-dimensional character array. Padding is used to keep the offsets of each data chunk in memory aligned positions. The *offsets* array holds the offsets of the individual data chunks while the *sizes* array stores their sizes (excluding the padding). 30

5.4 The state table produced by the serialization of the Aho-Corasick DFA as a two-dimensional integer array. Negative values indicate final states. . . 32

5.5 Overview of the tree matching phases. When the task of the first matching phase encounters a negative state, a match has been successfully found. In the second phase, we use the absolute value of this state in order to index the table that pairs the simple strings with their regular expressions. We use those IDs so as to index the array housing the regex automata. If a match using these automata is found, we use again this ID in order to index the array housing the rule messages. 38

6.1 Characteristics of the four DFAs produced by the four pattern sets found in the **Synthetic ASCII** workload. The x-axis of both plots indicates the number of regular expressions combined into a singe DFA. Figure 6.1(a) displays the size of the produced DFAs, while Figure 6.1(b) displays the time needed for their creation. We notice that the automaton size and its creation time are proportional to the number of regular expressions combined. 42

6.2 Characteristics of the one hundred DFAs produced by the regular expressions of the **Huge Regular Expressions** workload. The x-axis in both plots indicates the size of the various regular expressions. Figure 6.2(a) displays the size of the DFA produced by each regular expression, while Figure 6.2(b) displays the time needed for its creation. We notice that the size of the DFA as well as its creation time are not always proportional to the size of the regular expression. Complex regular expressions produce larger DFAs or require more time to be compiled. 43

6.3 Characteristics of the DFAs produced by combining 25 and 50 random regular expressions found in the **Huge Regular Expressions** workload. The x-axis in both plots indicates the number of regular expressions combined in a singe DFA. Figure 6.3(a) represents the size of the produced DFAs, while Figure 6.3(b) displays the time needed for their creation. We notice that the automaton size and its creation time are not linear to the number of regular expressions combined. 44

6.4 Sustained throughput achieved by our single-threaded CPU implementation of the *string matching* module using the automata produced by the four pattern sets found in the **Synthetic ASCII** workload. As the number of signatures increases, the size of the automaton grows. Bigger automata result to an increased number of cache references, imposing a performance penalization. Note that the x-axis is in log-scale. 45

6.5 Sustained throughput achieved by our single-threaded CPU implementation of the *regex matching* module. Figure 6.5(a) displays the processing throughput using one hundred DFAs produced by the regular expressions of the **Huge Regular Expressions** workload, while Figure 6.5(b) displays the performance achieved using DFAs combining 25 and 50 random regular expressions of the same workload respectively. 46

6.6 Sustained throughput achieved by the CUDA flavor of the *string matching* module using the automata produced by the four pattern sets found in the **Synthetic ASCII** workload. The x-axis in both plots represents the size of the DFAs produced by combining 10, 100, 1000 and 10000 patterns respectively. Figure 6.6(a) displays the throughput achieved by the GPU, while Figure 6.6(b) displays the end-to-end throughput, including the data transfers to and from the device. We notice that when the size of the automaton grows larger than the size of the cache, the performance is substantially decreased and remains consistent regardless of the size of the DFA. 47

6.7 Sustained throughput achieved by the CUDA flavor of the *regex matching* module using one hundred DFAs produced by the regular expressions of the **Huge Regular Expressions** workload. The x-axis in both plots represents the size of the various regular expression DFAs. Figure 6.7(a) displays the throughput achieved by the GPU, while Figure 6.7(b) displays the end-to-end throughput, including the data transfers to and from the device. We notice that in most cases the size DFA does not directly affect the performance. The sustained throughput is function of the complexity of the regular expression and the amount of backtracking performed on the input. 48

6.8 DFA sizes and sustained throughput achieved by the CUDA flavor of the *regex matching* module using combinations of 25 and 50 random regular expressions. Figure 6.8(b) displays the size of the DFAs, while Figure 6.8(a) displays the sustained throughput. We notice that the size difference of the DFAs is irrelevant to the number of regular expressions combined and does not affect the performance of the system 48

6.9 Sustained throughput achieved by the OpenCL flavor of the *string matching* module, using the pattern sets described in section § 6.2.1. The x-axis in both plots represents the size of the DFAs produced by combining 10, 100, 1000 and 10000 patterns respectively. Figure 6.9(a) displays the "in-core" throughput, while Figure 6.9(b) displays the end-to-end throughput, including the data transfers to and from the device. 49

6.10 Comparison of the sustained throughput achieved by the OpenCL flavor of the *regex matching* module versus the CUDA implementation, using one hundred DFAs produced by the regular expressions of the **Huge Regular Expressions** workload. Figure 6.10(a) displays the "in-core" throughput achieved by the OpenCL flavor, while Figure 6.10(b) presents the results of the same experiment conducted using the CUDA version of the module. We notice that the difference between the throughput of the two implementations is insignificant, since the performance gained by the hardware optimized CUDA version is hidden due to backtracking. 50

6.11 Comparison of the sustained throughput achieved by the OpenCL flavor of the *regex matching* module, versus the CUDA flavor using combinations of 25 and 50 random regular expressions found in the **Huge Regular Expressions** workload. Figure 6.11(a) displays the performance of the OpenCL flavor, while Figure 6.11(b) displays the throughput achieved by the CUDA flavor. We notice that the CUDA version slightly outperforms the OpenCL version since it is optimized for the NVIDIA hardware. . . . 51
6.12 Sustained throughput achieved by the OpenCL flavor of the *string matching* module, executed on the Intel Xeon E5-2697 CPU, using the automata produced by the four pattern sets found in the **Synthetic ASCII** workload. The x-axis represents the size of the DFAs produced by combining 10, 100, 1000 and 10000 patterns respectively. Each pattern set is processed using 6, 12, 18 and 24 CPU cores. Bigger automata result to increased number of cache references, thus imposing a performance penalization. The processing throughput is increased as more cores are available to the engine. However, the performance gained by each extra CPU core is not fixed, since more cores compete for the same memory space. 52
6.13 Sustained throughput achieved by the OpenCL flavor of the *regex matching* module, executed on the Intel Xeon E5-2697 CPU, using one hundred DFAs produced by the regular expressions of the **Huge Regular Expressions** workload. Each regular expression is processed using 6, 12, 18 and 24 CPU cores. The x-axis in all plots represents the size of the various regular expression DFAs. The processing throughput is increased as more cores are available to the engine. We notice that the performance gained by each extra CPU core is not fixed, since more cores compete for the same memory space. 53
6.14 Sustained throughput achieved by the OpenCL flavor of the *regex matching* module, executed on the Intel Xeon E5-2697 CPU, using combinations of 25 and 50 random regular expressions found in the **Huge Regular Expressions** workload. Each pattern set is processed using 6, 12, 18 and 24 CPU cores. The x-axis indicates the number of regular expressions combined in a singe DFA. The processing throughput is increased as more cores are available to the engine. We notice that the performance gained by each extra CPU core is not fixed, since more cores compete for the same memory space. 54

List of Tables

2.1 Output function 9
2.2 Failure function 9

3.3 Regular expression metacharacters and their functionality. 13
3.4 Regular expression character classes and their functionality. 15

Chapter 1

Introduction

String pattern matching is one of the most common operations in various applications. Searching for a certain string pattern in a text file may seem trivial for the everyday user. However, pattern matching is the core operation of many applications, ranging from text editors, intrusion detection systems and databases, to the analysis of biological sequences and classification of music signals [1]. Thus, there is a demand for efficient pattern matching algorithms, that each time suit to the nature or the specified application.

To begin with, network monitoring applications, such as network intrusion detection systems and spam filters, are dedicated to inspecting the contents of a huge amount of network traffic against an increasing number of suspicious signatures. These signatures are usually being preprocessed in a finite automaton that will be used later in order to match any suspicious incoming packets from the network. In addition, the efficiency of spell checking (i.e. text retrieval, databases, etc.) strictly depends on the efficiency of the pattern matching algorithm that is being utilized. Likewise, the performance of an approximate pattern matching algorithm can highly affect the duration of various experimental results in the area of biology and other relevant fields.

Consequently, it is obvious that the improvement of a pattern matching algorithm can drastically affect the performance of any application. This is the reason that a wide research community has dedicated many years on developing new algorithms and improving the already existing ones. Some approaches focus on matching multiple patterns –either simple strings or binaries– in a given input text simultaneously [2, 3, 4, 5, 6, 7]. Furthermore, other approaches propose algorithms for single-pattern searching [8, 9, 10, 11, 12]. Either way, taking advantage of the computational power and capabilities of modern processors has led to a new burst of algorithms, tailored for highly parallel environments. Previous works rely on the parallel hardware architecture of accelerators and GPGPUs [13, 14]. In this work we extend the already known functionality for string pattern matching. Utilizing any underlying commodity system setup, we propose a pattern searching framework, supporting simultaneous matching of multiple simple strings and regular expressions with a single pass. Our technique exhibits performance gain and high efficiency, regarding the throughput and power consumption. In the following section we explicitly present the main contributions of this work.

1.1 Motivation

Current string matching tools and libraries, offered as state-of-the-art, are mostly targeting a single group of applications (e.g. network packet processing applications) and are tailored for the needs of this particular group. For example, the `hyperscan` library – Intel's multiple regular expression matching library– is constructed to be used in a typical deep packet inspection (DPI) library stack [15, 16]. This means that any optimization that has been applied in order to boost the performance of this library, is basically best-fit for network processing applications. Likewise, a protein similarity search tool (e.g. FASTA [17]) is dedicated to a single application –the successful utilization of this tool for a different purpose is not guaranteed, since using a different and larger alphabet set A might not scale well. Therefore, the literature lacks of efficient general purpose string matching libraries or tools. To fill this gap, we propose a string pattern matching framework that can be used for diverse applications, exploiting the computational capabilities of accelerators and any commodity processor in a system. In addition, our framework provides multiple regular expression matching, something that – to the best of our knowledge– is rarely provided as a functionality from popular string searching tools[1].

1.2 Contributions

The contributions of this work are:

- Our implementation performs simultaneous matching of multiple fixed strings and binary patterns against multiple input data streams, with a single pass over the input bytes (Chapter 5 § 5.3).
- The engine is able to achieve simultaneous matching of multiple POSIX Extended Regular Expressions against multiple input data streams (Chapter 5 § 5.4).
- Our massively data parallel engine is implemented as a portable framework, exposing C and Java APIs. Thus it can be embedded and utilized by various applications (such as Search Engines, NIDS, Firewalls, Antivirus etc.) (Chapter 5 § 5.6).
- The engine exposes input data buffers easily handled by the API, able to receive input from various data streams (storage, NICs, etc) (Chapter 5 § 5.2 and § 5.6).
- Since the engine is implemented using both CUDA and OpenCL, it is easily executed on the vast majority of data parallel platforms such as CPUs and high-end discrete or integrated GPUs (Chapter 5).
- The API hides the specifics of the data parallel implementation while our initialization task is responsible to identify the appropriate devices, present to the system, and properly set the various parameters (Chapter 5 § 5.6).

[1] We exclude the `hyperscan` string matching library from this statement, since we refer only to works built upon accelerators.

- We extend the syntax of common string matching in order to support readable binary signatures and their combinations with clear-text (Chapter 5 § 5.3).
- We offer a syntax for matching-rules that allows events to be triggered when a match is achieved (Chapter 5 § 5.5).

1.3 Publications

Parts of the work for this thesis have been used and published in three European projects; the NECOMA project under grant agreement No.608533 [18], the RAPID project under grant agreement No.644312 [19] and the SHARCS project under grand agreement No.644571 [20]. Specifically, parts of this work are included in the following:

- NECOMA Project. Deliverable D3.5: Countermeasure Application - Results, 2015 [21].
- RAPID Project. Deliverable D2.4: Antivirus Ported on RAPID, 2016 [22].
- SHARCS Project. Deliverable D4.2: Design specification of the SHARCS runtime system, software tools and reporting, 2016

1.4 Outline

The rest of this dissertation is organized as follows. Chapter 2 presents a brief background on various string matching algorithms and their classification. In this chapter, we also describe in detail the properties and the functionality of the Aho-Corasick string matching algorithm. In Chapter 3, we provide the formal definition of regular expressions and a thorough description of their syntax and properties. Moreover, we present how a regular expression engine works according to POSIX standards. In Chapter 4, we enumerate and present the various parallel architectures available, as well as their classification according to the popular Flynn's Taxonomy. We also provide information about the most popular parallel hardware platforms, the way they achieve parallelism and a break-down of the popular NVIDIA GPU architecture.

Chapter 5 describes in detail the development and functionality of our massively parallel regular expression and string matching engine. This chapter expands on the two major components used in this work and the way parallelism is achieved using the techniques and architectures shown in Chapter 4. The first component performs the string matching and it is based on a variation of the Aho-Corasick algorithm, described in Chapter 2. The second component is a parallel regular expression engine, operating according to the specifications provided in Chapter 3. A thorough evaluation of our engine, developed using three different programming languages on two different parallel architectures, is provided in Chapter 6.

Chapter 7 surveys prior work and finally Chapter 8 summarizes this dissertation and points out future research directions.

Chapter 2

String Matching Algorithms

String matching algorithms, also called *string searching* algorithms, are an important subset of string algorithms that try to locate the occurrence of one or more strings (also called patterns or signatures) within a larger string (e.g. plain-text). String pattern matching, using finite alphabets is a very common technique in order to locate any occurrence of a string pattern into a text. For example, when searching for a string pattern $P = p_1p_2...p_n$ inside a text $T = t_1t_2...t_m$ (with lengths n and m accordingly), both characters sequences form a finite alphabet set A.

2.1 Classification Based on Pattern Number

The approach that each pattern matching algorithm follows, contributes to a manner of classification. In regard to the number of patterns each algorithm uses, we define the following three categories: *a*) single-pattern, *b*) multi-pattern (using a finite set of patterns), and *c*) regular expression (using an infinite set of patterns) algorithms.

2.1.1 Single Pattern

Single-pattern matching algorithms are used in order to individually search one pattern P against a given text T. First of all, there is the naive string search algorithm, which does not require any preprocessing – however, it is extremely inefficient. Karp-Rabin algorithm [4] uses hashing to find any single pattern in a set of strings in a text. The Knuth-Pratt-Morris [8] and Boyer-Moore algorithms [9] have been proposed in order to locate string patterns into a given text with errors or mismatches. More specifically, the Knuth-Pratt-Morris algorithm exploits the properties of the target pattern in order to decrease the comparisons needed, by using a partial-match table for each pattern. The table is built by preprocessing each pattern separately and indicates how many positions the pattern should be shifted to the right, based on the position in the pattern where a mismatch occurs. The Boyer-Moore algorithm skips as many characters as possible without missing a possible instance of the string it is searching for. The algorithm begins matching from the last character of the pattern and in case of a mismatch, skips a part of the input. The

execution time of the algorithm can be sub-linear when the suffix of the pattern does not appear frequently in the input text. In addition, a modified version of the Boyer-Moore algorithm, called Boyer-Moore-Horspool [23] requires less memory without penalizing the performance in terms of latency. Furthermore, there is the Baeza-Yates-Gonnet algorithm [24] – also known as the bitap algorithm – that efficiently utilizes bit-wise operations, in order to allow approximate string matching. This algorithm reports whether a given input text contains a sub-string that is *approximately equal* to a pattern. This "equality" is defined through the Levenshtein distance –if the sub-string and pattern are within a given distance k of each other, then they are considered equal [25]. The algorithm starts by pre-processing a set of bit-masks that contain one bit for each element of the pattern. Then, most of the remaining work is accomplished with bit-wise operations, something that can significantly boost the performance.

2.1.2 Finite Number of Patterns

The algorithms described in the previous section have been developed in order to look up a single pattern with minimal latency. However, it is obvious that in order to search a set of k $(k > 1)$ string patterns into a given text, we must execute the algorithms k times –something that is far from efficient. Consequently, when there is a necessity for locating *multiple* patterns into a text, these algorithms lack effectiveness. The algorithms that correspond to this category are mainly called multi-pattern matching algorithms and scale much better than searching each pattern individually. Karp and Rabin proposed such a multi-pattern algorithm [4]. For text T of length m and p patterns of length n, the worst-case time achieved by the Karp-Rabin algorithm is *O(nm)* in space *O(p)*. In contrast, the Aho-Corasick string matching algorithm has asymptotic worst-time complexity *O(n+m)* in space *O(m)* [2]. The algorithm matches all strings simultaneously. We thoroughly discuss the Aho-Corasick algorithm in the following section (§ 2.3). Furthermore, there is the Commentz-Walter algorithm [3], which combines the basic from the Aho–Corasick algorithm with the fast matching of the Boyer-Moore string search algorithm. Finally, Wu and Manber proposed an algorithm for multi-pattern searching [5]. This algorithm is being used and implemented as part of the *agrep* tool [26].

2.1.3 Infinite Number of Patterns

A regular expression is a sequence of symbols that defines a pattern. Concerning string matching, regular expressions are used in order to define the pattern to search in a text. Regular expressions describe the form of a pattern. For example, `(a|b)*` denotes the infinite set {ϵ, "a", "b", "AA", "ab", "BA", "BB", "AAA", ...}. In the following chapter (Chapter 3) we dive into more details regarding regular expressions and regular grammars.

2.2 Other Classifications

Except for the aforementioned classification, string pattern matching algorithms can be further categorized, regarding other aspects of design and implementation. For instance,

one of the most common approaches for the classification of string matching algorithms is referring to the *preprocessing* phase as criterion. The preprocessing is conducted in order to achieve faster searching, preparing either the pattern or the input text (sometimes even both). There are four categories regarding preprocessing, where: *a*) neither the pattern nor the input text is preprocessed –these algorithms are called *naive* or *elementary*–, *b*) only the pattern is preprocessed –these string pattern matching algorithms are based on finite state automata–, *c*) only the input text is preprocessed –these algorithms follow indexing approaches–, *d*) both the pattern and the input text is preprocessed –this algorithm category contains approaches based on finite state automata and indexing.

Furthermore, taking into consideration the matching strategy, another classification approach appears. The algorithms are also clustered as: *a*) prefix-matching (e.g. Aho–Corasick), *b*) suffix-matching (e.g. Boyer-Moore), *c*) best-factor-matching, and *d*) others (e.g. Karp-Rabin, naive pattern matching).

In the following subsections, the aforementioned pattern matching approaches (i.e. naive string, finite state automaton based and index based search) are briefly explained. Then, the definitions of exact and approximate (fuzzy) string matching are discussed.

2.2.1 Naive String Search

In the *naive* string search, the procedure to locate an instance of a string *–needle–* inside another one *–haystack–* is extremely simple and straightforward. We simply look for the needle, starting from the first character of the whole haystack; if the needle is not there, we proceed to the second character of the haystack; if the needle is not matched there neither, we proceed to the third character of the haystack, and so on. This procedure is continued until the needle is found somewhere inside the haystack. As expected, this approach is the most inefficient and resource consuming. The complexity of the naive string search is $O(\nu\mu)$, where μ is the length of the haystack and ν the length of the needle. In average, it requires $O(\nu + \mu)$ steps.

2.2.2 Finite State Automaton Based Search

To avoid backtracking, a *deterministic finite automaton (DFA)* is constructed. Constructing a DFA is ,in some cases, an expensive procedure, since it utilizes the powerset construction –a standard method for converting a non-deterministic finite automaton (NFA) into a deterministic finite automaton (DFA). Therefore, this approach is commonly used in cases, where the DFA needs to be rarely generated.

2.2.3 Index Based Search

This kind of search requires that the input text is being preprocessed. In this approach, a sub-string index (structures like a suffix tree, a suffix array, a compressed suffix array, etc.) is built in order to allow the fast discovery of the occurrences of a string pattern. For instance, building a suffix tree requires $\Theta(\mu)$ time, while locating all x occurrences of a

string pattern takes $O(\nu)$ time –assuming a constant size for the alphabet A and that all inner nodes in the tree know their child leafs.

2.2.4 Exact String Search

Exact string matching is the approach of locating the occurrence of a simple string pattern into another string. The exact string matching problem is to find all sub-strings in the input text that are *exactly* the same as the pattern. For example, the pattern `exact` matches the string `exactly` inside an input text.

2.2.5 Approximate String Search

Approximate string matching –also called fuzzy string matching– is the process of *approximately* locating a pattern into a string. The approximate string matching problem is to find all sub-strings in the input text, that are as close to the string pattern as possible, under some metric of closeness (e.g. using the Hamming distance [27]). In approximate string search, the pattern is matched under some error tolerance. For instance, the pattern `fuzzy` matches the string `furry` inside an input text –given an error tolerance of 40%.

2.3 The Aho-Corasick Algorithm

This section is dedicated to the Aho-Corasick algorithm – the most efficient algorithm that suits best to the SIMD (§ 4.2.2) and SIMT (§ 4.2.5.1) model of this work. In general, it is one of the most widely used algorithms for simple string pattern matching [2]. The Aho-Corasick algorithm is considered as the best option for multiple pattern searching, since it matches all signatures simultaneously. This simultaneous matching can be achieved when the set of patterns is being preprocessed. In the preprocessing phase, an automaton is built, which will be eventually used in the matching phase. Also, each character of the text will be processed only one time during the matching phase. The Aho-Corasick algorithm has the property that, theoretically, the processing time does not explicitly depend on the number of patterns. Let $P = p_1p_2...p_n$ be the patterns to be searched inside a text $T = t_1t_2...t_m$ (with lengths n and m accordingly), both sequences of characters form a finite character set Σ. The complexity of the algorithm is linear in the pattern length ν, plus the length of the given text μ, plus the number of output matches.

Given a set of patterns, the algorithm constructs a pattern matching machine, that matches all patterns in the text at once, one byte at a time. Each processing action of the automaton, accepts an input event. The very first action starts with the initial state, represented with zero. Each action that accepts an input event moves the current state to the next state, based exclusively on that input. There are three distinct functions: *a*) a `goto` function, *b*) a `failure` function, and *c*) an `output` function. According to the input event, one function is being triggered. Figure 2.1 and Tables 2.2 and 2.1 [1], present an example of these functions for the set of patterns {he, she, his, hers}.

[1] The figure and tables are adapted from [2]

The `goto` function (Figure 2.1) determines if a state transition can be performed, based on the current state and the ASCII value of the input character. If the input character matches one of the transitions starting from the current state, then the state pointed to by this transition becomes the next state. Otherwise, the next state is resolved by the `failure` function $f(i = \mathit{current\ state})$. For example (based on Figure 2.1), the edge labeled h from 0 to 1 indicates that $goto(0, h) = 1$, while the absence of an arrow for a indicates failure. The `failure` function either drives a transition to one or more intermediate states, or to the initial state (the one that is represented with 0 in the `goto` graph). After each state transition, the algorithm checks the `output` function $output(i = \mathit{current\ state})$ in order to determine if the pattern matches a sub-string of the text T. This procedure continues and terminates with the end of the input text T.

Since failed transitions may not consume any input –the so-called ϵ-transitions– the produced automaton is non-deterministic (NFA). Also, the `failure` function can result to numerous state transitions for a single input character. In this way, the matching operation might require the exploration of multiple paths before the actual match of a pattern.

A revised version of the traditional Aho-Corasick exists and replaces all failure transitions in order to avoid the performance loss, when the patterns' sizes are large. The new automaton that is produced is called "deterministic finite automaton" (DFA) and provides one transition per state and input character. Despite this approach requiring more memory than the previous one, it is more efficient in terms of processing throughput. The achieved complexity of this approach is $O(n)$.

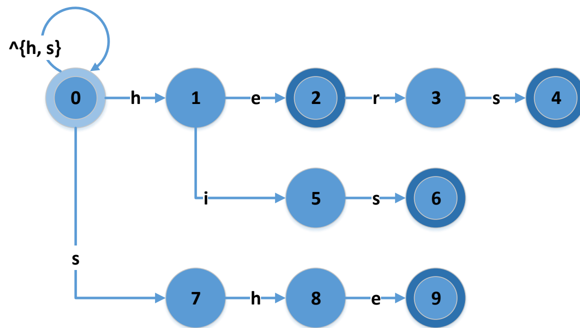


Figure 2.1: Goto function

Table 2.1: Output function

i	$output(i)$
2	{he}
5	{she, he}
7	{his}
9	{hers}

Table 2.2: Failure function

i	1	2	3	4	5	6	7	8	9
$f(i)$	0	0	0	1	2	0	3	0	3

Chapter 3

Regular Expressions

A *regular expression* [28], also called *regex*, *regexp* or *rational expression*, is a pattern describing a certain amount of text. The concept of regular expressions originated by the mathematician Stephen Cole Kleene [29] in 1956. Kleene described regular languages using his mathematical notation, called regular sets. Regular expressions are used as *search patterns* by searching algorithms in order to locate sub-strings into a large input string. Regexes can be found in search engines, word processors (search and replace functionality), lexical analyzers, text processing utilities (`sed` [30], `AWK` [31], etc.) as well as Network Intrusion Detection systems (NIDS), such as `snort` [32], or Antivirus, like `ClamAV` [33].

3.1 Formal Definition

Regular expressions have the same expressive power as regular grammars and describe regular languages. They consist of constants and operators, defined as symbols. In this way, they define string sets and the operations on these sets.

Given a finite alphabet Σ, the following constants are defined as regular expressions:

$\varnothing$ (empty set)	Represents the empty set $\varnothing$.
ϵ (empty string)	Represents the set containing the "empty" string, containing no characters at all.
Literal character	Let α be a character and $\alpha \in \Sigma$, then α represents the set containing only the character α.

Given the regular expressions R and E, the following operations over them are defined to produce regular expressions:

RE (concatenation)	Defines the set of strings obtained after concatenating the strings in R and the strings in E. For example, let R be {"ra","b"} and E be {"s","t"}. Then RE = {"ra","b"}{"s","t"} = {"ras","rat","bs","bt"}.
R|E (alternation)	Defines the union of the sets described by R and E. For example, let R be {"ra","w"} and E be {"s","ra","q"}. Then R|E = {"ra","w"}|{"s","ra","q"} = {"ra","w","s","q"}.
R* (Kleene star)	Defines the smallest super-set of the set described by R that contains the empty string ϵ and is closed under string concatenation. This set contains all strings that can be made by concatenating *zero* or *any* finite number of strings found in R. For example if R is the set {"ra","s"}*, this it is equivalent to {"ϵ","ra","s","rara","ras","ss","rasras","rasra", etc.}.

The Kleene star has the highest priority. Then, follows concatenation and then alternation. This assumption is made in order to avoid the use of parenthesis, if there is no ambiguity. For example `(ra)s` is equivalent to `ras` and `r|(a(s*))` is equivalent to `r|as*`. The formal definition does not define the quantifiers `+` and `?` as they can be expressed using the Kleene star. For example `r+` = `rr*` and `r?` = `r|`ϵ.

3.2 Syntax

The regular expressions are *patterns* used to match a target *string*. In contrast to simple string matching, regular expressions use character literals as well as meta-characters, which define special operations. Formally, each regular expression is composed as a sequence of *tokens*. A *token* is a single point within the regular expression, which tries to match with the target string. The concept is to compose a small set of characters, the tokens of the regular expression, in order to match a (very) large number of possible target strings, rather than compose a list of all the literal possibilities (target string permutations). For this reason, a match is achieved when all the regex tokens match the target string and *not* when all the string characters are matched. The meta-characters provide extra functionality rather than matching as literal characters in the target string. Most tools and regular expression engine implementations use delimiters, so as to receive the regular expressions as input.

3.2.1 Delimiters

Programming languages and string matching tools use delimiters in order to receive the regular expression as input. A common case is to enter the regex as a quoted string literal. For example in Python, C and Java the input regular expression would be `"[0-9A-Z]"`. An other set of delimiters used is the starting and ending slashes. This originated by the `ed` [34] text editor. The UNIX [35] `grep` [36] utility got its name by the famous `g/re/p` (**g**lobally search a **r**egular **e**xpression and **p**rint) command used in `ed`, following

this delimiter convention. An other famous utility using this convention is `sed` (**s**treaming **ed**itor) where replace functionality is provided by the command `s/re/replacement`. However, this notation became popular due to its use in Perl. Sometimes, alternative delimiters, such as comma, are used by some tools in order to avoid escaping occurrences of the delimiter character or content collision. For example the command `s,/,A` in `sed` will replace the "/" with "a".

3.2.2 Metacharacters

In the need of searching more than literal pieces of text, a set of characters is reserved for special use. This set of characters, known as *metacharacters* or *special characters*, provide functionality such as *grouping*, *negation*, *logical OR*, *logical NOT*, *quantification* and *backreferencing*. Most metacharacters produce an error when used alone or without proper syntax but can be escaped if needed to be matched as literals. A list of metacharacters found in most regular expression engines, along with their syntax and description, can be found in Table 3.3.

Table 3.3: Regular expression metacharacters and their functionality.

Metacharacter	Description
[ ]	The square brackets define a *character class*, also called *character set*. The *character class* will match a *single* character contained within the brackets. For example, `[ab]` matches "a" or "b" but does not match "eabc". Ranges can be specified using the − character. For example `[a-z]` matches any *single* lowercase letter from "a" to "z". The two forms can be mixed. For example `[ab0-3]` matches "a", "b", "0", "1", "2" or "3".
[ˆ]	The ˆ character, after the opening square bracket, negates the character class. The *negated character class* matches any character that is *not* in the character class.
( )	Parentheses define a marked sub-expression as well as create a numbered capturing group. This group stores the part of the input matched by the part of the regular expression inside the parentheses.
{min,max}	A range between curly brackets, specified with a comma, defines how many times a token can be repeated. The *min* value can be an integer greater or equal to zero and indicates the minimum number of matches while *max* can be an integer equal to or greater than *min* indicating the maximum number of matches. For example `r{2,4}` matches only "rr", "rrr" or "rrrr". The infinite maximum number of matches can be specified if the comma is present but *max* is omitted. For example `a{5,}`. Omitting both the comma and *max* repeats the token exactly *min* times. For example `s{3}` only matches "sss".

.	The dot matches (almost) any *single* character. For example `r.s` matches both "ras", "rbs", etc.. The dot character included in square brackets matches the literal dot. For example `[r.s]` matches only "r", "." or "s".
?	The question mark indicates *zero* or *one* occurrence of the preceding token. For example rege?x matches "regx" or "regex".
*****	The asterisk indicates *zero* or *more* occurrences of the preceding token. For example `rege*x` matches "regx", "regex", "regeex", "regeeex", etc..
+	The plus sign indicates *one* or *more* occurrences of the token. For example `rege+x` matches "regex", "regeex", "regeeex", etc. but not "regx".
|	The pipe character, called *alternation* or *set union*, matches either the expression before or the expression after it. For example `regular|expression` matches either "regular" or "expression".
^	The caret character, when not included in square bracket, is called *start of string anchor*, and does not match any character. It is used to "anchor" the match at the starting position of the input. For example the regex `^a` when used on the input "abcd" matches "a" but the regex `^b` when used on the same input will *not* match "b" because it is not the first character of the input.
$	The stand-alone `$` character, called *end of string anchor* behaves similar to the stand-alone ^ but "anchors" the match at the end of the input.

3.2.3 Character Classes

The character classes allow a small sequence of characters to define, and thus match, a larger set of characters. This is the most basic concept after the literal match. Character classes can be defined using square brackets. For example the character class `[0-A]` specifies all the numbers and symbols found in the ASCII range from the character "0" up to character "A". The POSIX [37] standard defines various character classes which will be known by the regex processor. The "default" character classes used by the POSIX and Perl implementations can be found in Table 3.4.

3.2.4 Standards

POSIX is a collection of standards that define some of the functionality that a UNIX operating system should support and stands for "Portable Operating System Interface for uniX". One of the standards in this collection defines three regular expression flavors, named POSIX BRE (Basic Regular Expressions) [38], POSIX ERE (Extended Regular

Table 3.4: Regular expression character classes and their functionality.

POSIX	Perl	ASCII	Description
[:lower:]		[a-z]	Lowercase letters
[:upper:]		[A-Z]	Uppercase letters
[:digit:]	\d	[0-9]	Digits
	\D	[ˆ0-9]	Non-digits
[:xdigit:]		[A-Fa-f0-9]	Hexadecimal digits
[:alpha:]		[A-Za-z]	Alphabetic characters
[:alnum:]		[A-Za-z0-9]	Alphanumeric characters
	\W	[ˆA-Za-z0-9_]	Non-word characters
[:punct:]		[]!"#$%&'()*+,./:;<=>?@\ˆ_‘| -	Punctuation characters
[:blank:]		[\t]	Space and tab
[:space:]	\s	[\t\r\n\v\f]	Whitespace characters
	\S	[ˆ \t\r\n\v\f]	Non-whitespace characters
[:graph:]		[\x21-\x7E]	Visible characters
[:print:]		[\x20-\x7E]	Visible characters and the space
		[\x00-\x7F]	ASCII characters
	\b	(?<=\W)(?=\w)|(?<=\w)(?=\W)	Word boundaries
[:cntrl:]		[\x00-\x1F\x7F]	Control characters

Expressions) [39] and POSIX SRE (Simple Regular Expressions) [40]. The POSIX SRE flavor is deprecated in favor of POSIX BRE since it provides backward compatibility.

3.2.4.1 POSIX Basic (BRE)

The Basic Regular Expressions (BRE) set of compliance is the oldest regular expression flavor still in use and it is supported by the UNIX `grep` utility. The distinctive characteristic of BRE is that some metacharacters require a backslash to give a metacharacter the special meaning. Specifically, BRE requires that the metacharacters `( )` and `{ }` are designated as `\( \)` and `\{ \}`. The reason behind this convention is that the oldest versions of UNIX `grep`, did not support these metacharacters and the developers wanted to keep `grep` compatible with existing regular expression syntax, which may use these characters as literal characters. For example the BRE `r{2,4}` matches "r{2,4}" literally while the BRE `r\{2,4\}` matches "rr, "rrr" or "rrrr". This means that using a backslash to escape a character that is never a metacharacter is an error, while other regex flavors, such as POSIX ERE, use the backslash to suppress the meaning of metacharacters.

The BRE also supports POSIX bracket expressions, a feature similar to character classes, with the exception that shorthands are not supported. The *dot* metacharacter is used to match any character, excluding the line brake, and the *star* is used to match a token zero or more times. Anchors are also supported using the *caret* and *dollar* signs. All these characters can be matched literally by escaping them with the \ symbol. No other features are supported by POSIX BRE, such as alternation.

3.2.4.2 POSIX Extended (ERE)

The POSIX Extended Regular Expressions flavor supports all the features of the BRE flavor but introduces many new ones. The "Extended" keyword is relative to the original UNIX `grep`, which only supported brackets, the dot, the star and the anchor metacharacters. The ERE flavor is used by the UNIX `egrep` utility. However, the developers of `egrep` did not try to maintain compatibility with `grep` but opted for a separate tool instead. For this reason POSIX ERE, and the `egrep` tool respectively, support additional metacharacters and do not require backslashes. Backslashes can be used in order to suppress the meaning of metacharacters. Escaping a character that is not a metacharacter always produces an error.

One of the most important features introduced in POSIX ERE is alternation. Using the vertical bar `|` as a metacharacter, the regular expression can match either the expression before or the expression after it. ERE also introduces the quantifiers `?`, `+`, `{n}`, `{min,max}` and `{min,}` which repeat the preceding token zero or once, once or more, n times, between min and max times, and min or more times, respectively. The POSIX standard, however, does not define backreferences. Based on the standard's definition, ERE is an extension of the old UNIX `grep` as `egrep` and not an extension of POSIX BRE.

3.2.5 Lazy Quantifiers

Certain implementations allow for *lazy* or *possessive* matching. For example, in Python and Java regular expression engines, the quantifiers `?`, `+` and `*` are *greedy* by default, trying to match as many characters as possible. However, those quantifiers can be made *lazy* –also called *ungreedy* or *reluctant*– by appending the question mark `?`. For instance, given the input string "
Regular Expression Engine
", the regex `<.*>` will match "
Regular Expression Engine
" while the regex `<.*?>` will only match "
" and "
".

3.2.6 Backreferences

Backreferences provide the functionality of matching again the same input text, as previously matched by a capturing group enclosed in parentheses. The backreference \n refers to the n-th capturing group and most implementations allow up to 99 capturing groups. For example, the regular expression `<[a-zA-Z0-9]>.*?</[a-zA-Z0-9]>` can be used in order to match HTML tags (the parts of the expression starting and ending with angle brackets) and the enclosed text `(.*?)`. The same regular expression can be declared as `<([a-zA-Z0-9])>.*?</\1>` using backreferences. In this case, `\1` refers to the first and only capturing group `([a-zA-Z0-9])`.

3.3 Regular Expression Engine Internals

The various regular expression engine implementations, found in many tools and applications, differ in terms of syntax and behavior. However, there are only two kinds of designs: (i) *regex-directed* engines and (ii) *text-directed* engines. Nearly all modern regex flavors are based on regex-directed engines. The reason behind this, is that they provide certain functionality, such as *lazy quantifiers* and *backreferences*, that can not be implemented in text-directed engines.

3.3.1 Regex-directed Engines

Regex-directed engines walk through the regular expression trying to match each token against the characters of the input string. Each time a match is achieved the engine advances through the regex and the target string. Upon a token mismatch, the engine *backtracks* to a previous regex token and string character. From this position, a different path is followed through the regular expression. Backtracking can be controlled in most regex flavors via *possessive quantifiers* and *atomic grouping*.

3.3.2 Text-directed Engines

In contrast to regex-directed engines, text-directed engines walk though the target input string attempting to match all regular expression permutations before continuing to the next string character. For this reason, a text-directed engine never backtracks. Although text-directed engines find the same matches as regex-directed engines (in most cases) and seem to be more efficient due to their lack of backtracking, they are not used by the modern regex engines. Such engines are not able to implement various features found in regex-directed engines and finding all regex permutations when attempting to match each string character is not efficient. Matches different than those found by regex-directed engines occur when the regular expressions use *alternation* with two possible alternatives that can match at the same target string position.

Chapter 4

Architectures and Parallel Computing

In many cases, large computational problems can be divided into smaller ones, which can be solved at the same time. This approach, where many executions or calculations are carried out simultaneously, is called *parallel computation*. This design concept is employed for many years in the field of *high-performance computing*. Interest in parallel computing has also grown lately due to the physical constraints preventing frequency scaling. Parallelism is divided into three major categories. The existing hardware architectures can be classified according to *Flynn's Taxonomy* [41], based upon the number of concurrent instruction and data streams available by the architecture.

4.1 Levels of Parallelism

Parallel computation can be divided in several different levels such as *bit*, *instruction* and *task* level parallelism. An other classification in existence is data parallelism.

4.1.1 Bit-level

The basic concept behind bit-level parallelism is increasing the processor word size. In this way, the number of instructions needed to be executed by the processor in order to perform an operation on variables whose sizes are greater than the length of the word, reduces. For example, if a 16-bit processor has to add two 32-bit integers, it must first add the 16 *low* bits from each integer and then proceed with the rest 16 *high* bits. This process requires two instructions to complete, while a 32-bit processor is able to perform it with a singe instruction.

4.1.2 Instruction-level

It is common for processors to be able to execute many instructions simultaneously. Instruction-level parallelism (ILP) is a measure of how many instructions in a process

can be executed simultaneously. The achievable level of ILP is application and workload specific. For example, if a process contains the following operations:

0. a = b + c;

1. d = e - f;

2. g = a / d;

then, the operations 0. and 1. can be executed in parallel, since they do not have any dependencies. However, the operation 2. can not be executed before 0. and 1. are completed.

This simultaneous instruction execution can be achieved either via hardware support or via software support. In the first case, also called *dynamic parallelism*, the *processor* decides at run time which instructions to execute in parallel. In the second case, also called *static parallelism*, the *compiler*, decides a-priori the set of instructions that are going to be simultaneously executed. Some common techniques used for exploiting instruction level parallelism are *instruction pipelining*, *superscalar execution*, *out-of-order execution* and *branch prediction*.

4.1.3 Task-level

Multiprocessor systems are able to execute different processes (threads) by assigning them on the various processors. Each process can execute on different or the same data-sets as the others. This level of parallelism is called task-level parallelism (TLP). For example, if an application composed by two threads (T_0 and T_1) is executed on a multiprocessor with two cores (C_0 and C_1), then it can dictate the thread execution and assign T_0 to C_0 and T_1 to C_1. In this way, the two threads will be executed in parallel instead of being serialized and context-switched on a single core.

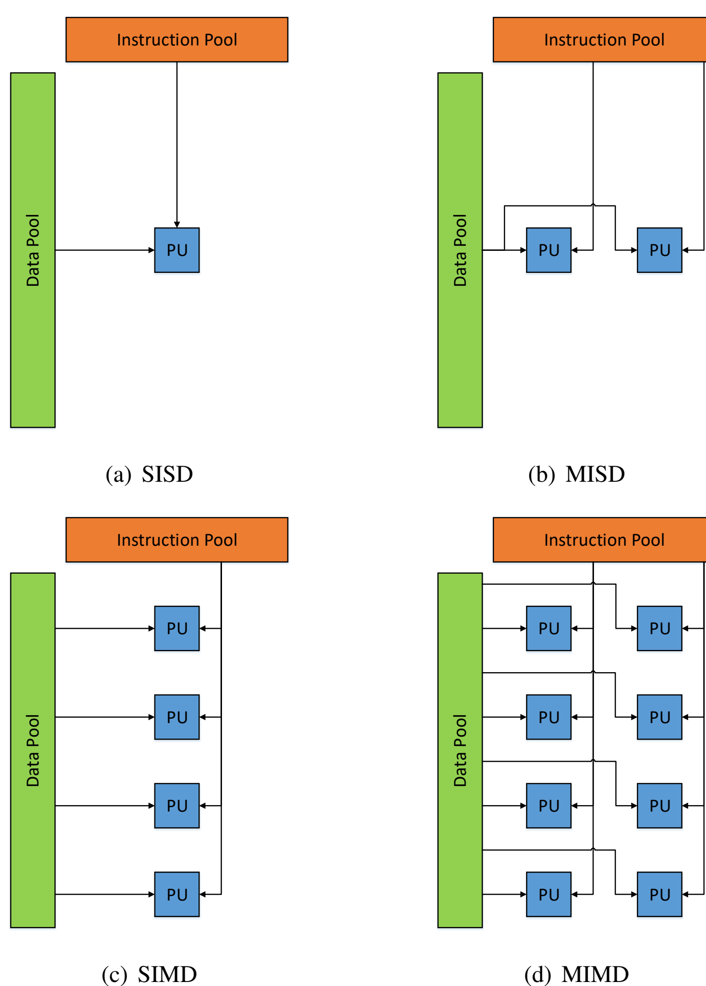
The concept of thread level parallelism became particularly popular into the commercial market with the advent of multi-core microprocessors. The main reason commercial desktop processors began to follow the multi-core trend, is that increasing the clock speed or instructions per clock is no longer practical or efficient. By increasing the number of processors, the applications able to utilize the hardware, are capable to overcome the clock-speed problem and benefit from the increase in computing power.

4.1.4 Data Parallelism

The concept of data parallelism is fundamentally different from the levels of parallelism described above. Data parallelism is achieved when the *same function* is simultaneously executed on multiple cores across *different data*. On the other hand, task-level parallelism is the simultaneous execution of *different functions* on multiple cores across the *same or different data*. In most cases, when data parallelism is applied, the different threads control the operations on all data elements, while in other cases a single execution thread controls the operations. However, in both designs all threads execute the same code.

4.2 Flynn's Taxonomy

Michael J. Flynn proposed in 1972 four broad classifications of computer architectures, based on (i) the interaction of their instruction set with the data streams, (ii) the number of concurrent instructions and (iii) data streams present in the architecture. Those four categories are named *SISD*, *MISD*, *SIMD* and *MIMD*. The first two categories include architectures with a single data stream, while the other two include architectures with multiple data streams. This classification system is adopted by the community and named *Flynn's Taxonomy*. A graphical representation of the four categories, displaying how the instruction streams interact with the data streams, is displayed in Figure 4.1



(a) SISD (b) MISD (c) SIMD (d) MIMD

Figure 4.1: The four basic computer architectures, proposed by Flynn [41]. The first two figures (Figures 4.1(a) and 4.1(b)) present the architectures with a single data stream, while the remaining two (Figures 4.1(c) and 4.1(d)) demonstrate the architectures with multiple data streams.

4.2.1 SISD

The SISD (**S**ingle **I**nstruction stream, **S**ingle **D**ata stream) classification, describes the type of architecture in which a single (uni-core) processor executes a single instruction stream, to operate on a single data stream.

4.2.2 SIMD

The SIMD (**S**ingle **I**nstruction stream, **M**ultiple **D**ata streams) category describes an architecture with multiple processing elements that perform the same operation on multiple data streams. Flynn divides the SIMD processors in three categories:

1. *Array* processors, where a single control unit connected with n independent processing elements operating on multiple data streams, upon command of the control unit.

2. *Pipelined* processors, a time-multiplexed version of the array processors.

3. *Associative* processors, a variation of the array processors, where the processing elements are not directly addressed.

4.2.3 MISD

The third class, according to Flynn, is called MISD (**M**ultiple **I**nstruction streams, **S**ingle **D**ata stream) and describes the computer architecture where multiple processing elements operate on a single data stream. This architecture is rather uncommon and generally used for fault tolerance. For example, in some heterogeneous systems, operating on the same data streams, where the various processing units must agree on the result.

4.2.4 MIMD

MIMD (**M**ultiple **I**nstruction streams, **M**ultiple **D**ata streams) is the last classification in Flynn's Taxonomy and describes the architecture where multiple autonomous processing elements execute different instructions on multiple data streams.

4.2.5 Other Classifications

With the rise of the multiprocessing CPUs and GPGPU (General-Purpose computing on Graphics Processing Units), the concept of Flynn's Taxonomy has been evolved and new classifications have been defined in order to cover the available architectures. The three most prominent ones are described in the following sections.

4.2.5.1 SIMT

The SIMT (**S**ingle **I**nstruction stream, **M**ultiple **T**hreads) classification was introduced by NVIDIA's Fermi microarchitecture [42]. This classification, describes the architecture that combines SIMD with multithreading. Processors designed with this architecture are

equipped with multiple processing units but are able to execute more tasks than the available units. This is achieved by each processing unit having multiple threads, also called work-items, which execute in lock-step. Each thread is a sequence of SIMD Lane operations and all threads are able to execute concurrently using a single instruction. Their properties vary from those of POSIX threads.

4.2.5.2 SPMD

One of the most popular classifications is SPMD (**S**ingle **P**rogram, **M**ultiple **D**ata streams), which describes the architecture where multiple autonomous processing elements simultaneously execute the same program as independent points on multiple data streams. It differs from SIMD in that no lock-step is imposed.

4.2.5.3 MPMD

The final architecture classification is called MPMP (**M**ultiple **P**rograms, **M**ultiple **D**ata streams) and describes systems where multiple autonomous processing units execute simultaneously at least two independent processes on multiple data. Typically, one processing unit is assigned to execute a manager which runs one program that farms out data to all the other processing units, all running the second process.

4.3 Commodity Hardware

In the following sections we will introduce definitions and we will provide some insight regarding the most common off-the-shelf hardware components that constitute modern commodity systems.

4.3.1 CPUs

The CPU (Central Processing Unit) is considered as the vital part of a system, responsible for executing the various processes. With the rise of multi-core CPUs, their capabilities have been extended in order to support parallel computing models. Processors, provided by the majority of vendors, offer extended instruction sets with support for hyper-threading and vector operations. Multi-core CPUs can be utilized in order to concurrently execute various independent, or co-operating, processes with high execution bandwidth. In the recent years, many CPUs also provide SIMD functionality which can be exploited either via dedicated intrinsics or by a parallel computing platform, such as OpenCL.

Due to the fact that modern CPUs allow parallelism in both instruction and task levels, as well as some level data parallelism, they are ideal for a vast variety of workloads. In comparison to GPUs, CPUs are regarded as more efficient for branch intensive workloads since they do not follow lock-step model.

4.3.2 Graphics Processing Units

Traditionally, GPUs are dedicated to graphics rendering. Nowadays, GPUs are also capable of handling massively parallel computations. GPGPU architectures consist of a set of *multiprocessors*, each containing a set of *streaming processors*, operating according to the SIMT model, as shown in Figure 4.2, thus being able to execute thousands of threads simultaneously. As described in section § 4.2.5.1, SIMT is similar to SIMD but provides multithreading. The SIMT instructions specify the execution and branching of a single thread while the SIMD vector organization exposes the SIMD width to the software. This architecture enables GPUs to be ideal for compute-intensive parallel applications that require high memory access bandwidth.

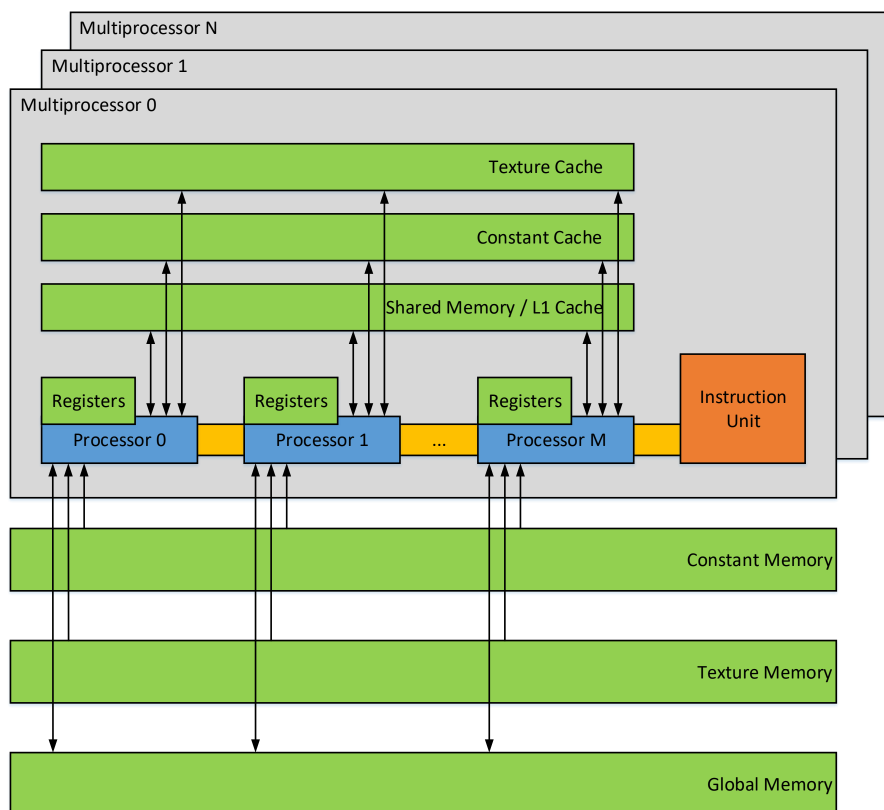


Figure 4.2: The NVIDIA CUDA architecture comprised of multiprocessors, each one containing multiple stream processors and a set of various memory spaces.

Tasks issued by the host computer to its connected GPU are called *kernels*. Each kernel is launched on the GPU device as a set of hundreds or thousands of *threads*, organized in *thread blocks*. Those thread blocks are executed by the multiprocessors, in a SIMT fashion, organized into same-sized groups called *wraps*. A typical kernel execution follows four steps:

1. A DMA controller transfers the required data from the host memory to the GPU memory.
2. The host program issues the kernel launch on the GPU.
3. The GPU executes the threads in parallel.
4. The DMA controller transfers the resulting data from the GPU memory back to the host memory.

A typical GPU memory organization consists of various memory spaces. Each thread block has a *shared* memory space visible to all threads of the block and has the same lifetime as the block, while each thread in a block has its own *local* memory. Moreover, there are three additional memory spaces named *global*, *constant* and *texture* respectively. All threads have access to those memory spaces and they are persistent across kernel launches by the same process.

Besides the discrete high-end GPUs, there are also integrated GPUs –packed on the same die with the CPU. An integrated GPU does not have a dedicated memory; it shares the same physical address space with the CPU. In comparison to the discrete GPU, the integrated GPU performs better with workloads bound to the I/O interface. However, the computational capacity of an integrated GPU is capped by the internal power control unit, in order not to exceed the thermal constraints (TDP) of the processor die [43, 44].

4.3.3 Other Accelerators

Except for GPUs, other hardware accelerators exist. Examples of such accelerators are (i) FPGAs, (ii) cryptographic accelerators, (iii) AI accelerators, (iv) physics processing units and (v) coprocessors –like the Intel Xeon Phi coprocessor [45].

4.4 Parallel Computing Platforms

Over the years, general-purpose computing is moving towards parallel designs and architectures. Recently, developers have taken advantage of the powerful multi-core processors in order to build highly parallel and efficient applications and systems. Vendors like Intel, NVIDIA and AMD offer convenient programming libraries that allow the parallelism for general purpose computing. The following two sections introduce the two main frameworks for parallel programming, used in this work; *CUDA* (§ 4.4.1) and *OpenCL* (§ 4.4.2).

4.4.1 CUDA

CUDA was proposed by NVIDIA and is a parallel computing platform and application programming interface [46]. Over the last years, CUDA has enabled developers to program NVIDIA GPUs for general purpose processing –not only for the traditional graphics rendering. The CUDA platform is a software layer that gives direct access to the GPU's virtual instruction set and various parallel computational elements. This access allows the execution of the *compute kernels* –units of work issued by the host computer to the GPU.

4.4.2 OpenCL

OpenCL is a framework that allows the uniform programming of heterogeneous platforms. Heterogeneous platforms mainly consist of CPUs, GPGPUs, DSPs, FPGAs and other type of processors and hardware accelerators (e.g. Intel Xeon Phi-coprocessor) [47]. OpenCL provides a standard interface for parallel computing allowing task-based and data-based parallelism through the execution of the compute kernels.

4.5 Programming Considerations

While the benefits of employing a parallel computing model seem to be appealing, one should first consider the following aspects. Transitioning from a serialized execution model to a task and/or data parallel model is not an automated process. In most cases it requires a lot of engineering effort. Deciding which parts to parallelize on a multicore CPU or offload to a highly parallel GPU relies on experience and understanding of the algorithms in use. For example, GPUs are suitable for offloading tasks with high data to instruction ratio (i.e. tasks performing the same operations on different input data), while multicore CPUs are more suitable for tasks with high I/O demands or processes where different tasks can be executed simultaneously. Moreover, the chosen architecture or programming platform may impose various limitations. For example, a SIMT model can suffer significant performance overheads due to the control flow divergence as threads executing with lock-step can follow different code paths. Thus, implementing efficient parallel programs also requires a deep understanding of the underlying hardware.

Chapter 5

Implementation

This chapter describes the implementation of our massively parallel regular expression and string matching engine. The engine is able to simultaneously match a very large number of different input data against large sets of fixed strings and binary patterns, as well as regular expressions. We present the design of our engine, using the CUDA platform in order to utilize the high processing power of NVIDIA's GPUs. To expand our implementation, we also use the OpenCL framework, exploiting the SIMT capabilities of any modern processor and accelerator. These two implementation flavors are designed as libraries, sharing a common simple API developed in C and Java –via the use of Java Native Interface (JNI). The API offers a layer of abstraction to the developer, who does not need to be aware of the underlying hardware platforms. Our system is able to detect the present hardware and utilize it to the maximum extent. The libraries can be used in order to embed our engine in various scenarios, such as Network Intrusion Detection Systems (NIDS), Antivirus and search engines, in order to benefit from the high processing throughput. Moreover, the functions exposed by the API can be used for the development of command-line utilities, operating with the same principles as `sed` or `grep`.

5.1 Design Overview

Our matching engine is designed as a composition of four different modules, each one developed using both CUDA and OpenCL. The modules are then compiled as a single library, exposing a unified API. Upon the bootstrap of our engine, an initialization task scans the system in order to identify the available hardware and the supported platforms. After this procedure is complete, the four modules are properly set in order to execute on the underlying hardware, using the suitable flavor. The overall architecture is shown in Figure 5.1. One component is responsible for transferring the data to and from the selected processing units and an other one for handling events when a match is found in the input. The other two components are responsible for performing the fixed string and regular expression matching respectively.

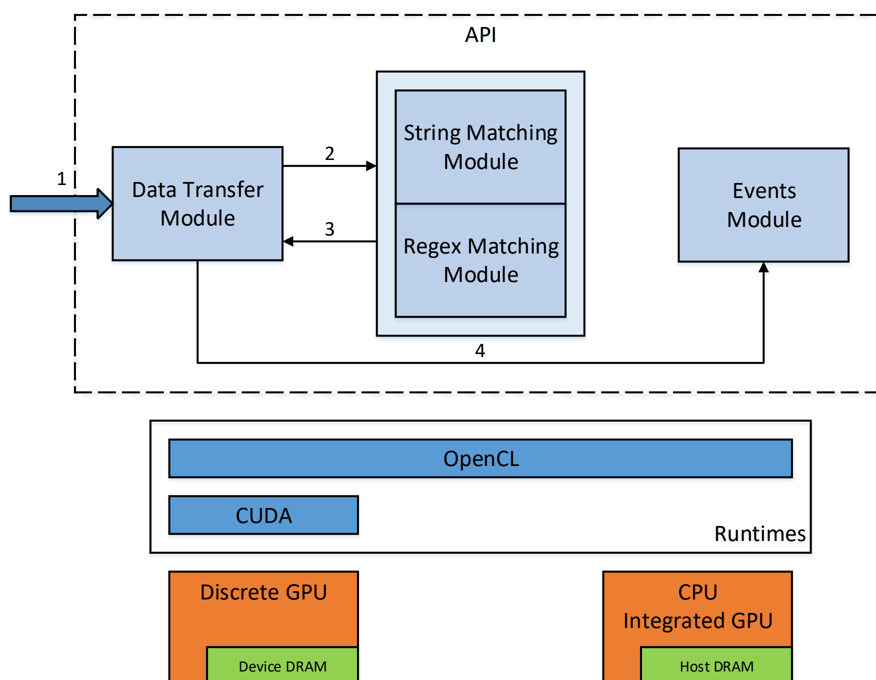


Figure 5.1: The overall architecture of our system. The interaction between the four components (*modules*) is performed as follows: data chunks arrive from various input data streams to the *data transfer* module (1) that batches them and hands those batches to the two processing modules, named *string matching* and *regex matching* modules, respectively (2); when the processing is finished (3) the *data transfer* module returns the results to the *events module*, responsible for the match reporting (4). In addition, we present the runtimes and the underlying hardware.

5.2 Data Gathering and Transferring

The input data that must be scanned for matches are handled by the *data transfer* module, which is responsible for gathering them in batches and transferring those batches and the results to and from the selected processing unit.

The *data transfer* module exposes an input buffer via the API. This buffer can be used in order to receive input from various sources, such as input files or network sockets. The simplest approach would be to transfer each chunk of input data, upon its arrival on the buffer, to the processing unit. However, this technique is rather inefficient for two reasons. Firstly, if a discrete GPU is chosen as processing unit by our engine, the data are transferred via the PCIe bus. Batching many small transfers into larger ones performs much better, since the initialization of the bus is not performed every time each individual data chunk is transferred. Secondly, the main concept behind SIMT and SIMD respectively, is that the same task –in our case, pattern matching– will be simultaneously

performed on multiple input data. When the data is batched and transferred as a set, a processing task is able to start on the batch. The main process, the one that launched the processing task, can construct the next batch in the meantime, achieving a pipeline between the I/O and the processing.

We design the data buffer as a ring buffer holding multiple buckets. The implementation of the ring buffer is particularly useful in our engine, since it allows three major operations to be performed on different buckets, achieving a pipeline between data gathering, data transferring and data processing. In its simplest form, the ring buffer contains three buckets. If we observe a snapshot of the ring buffer at a random time t_{rand} we will notice the following: (i) one of the three buckets is being constructed using data from the input stream, (ii) the second bucket, which is already constructed, is being transferred to the appropriate processing unit, and (iii) the transferring of the third bucket is complete and its contents are being processed. Since these three operations are performed in parallel, on different buckets, a pipeline is achieved. In practice, our ring buffer can hold more than three buckets and dynamically increase their number, up to a certain extend, in order to accommodate the needs of host processes with high I/O bandwidth. Such applications can be network facing processes that use the data buffer in order to scan incoming packet traffic. An overview of the ring buffer is shown in Figure 5.2.

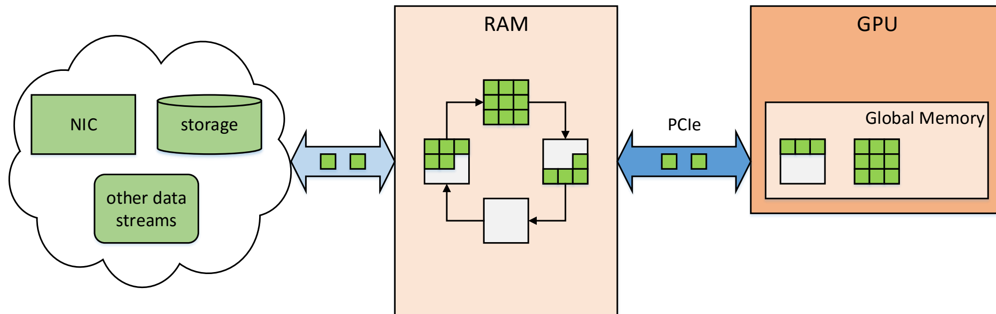


Figure 5.2: Ring buffer overview. Data chunks arriving from various input streams are batched into ring buffer buckets. Once a bucket is complete, it is transferred to the discrete GPU via the PCIe bus. If the buckets are processed by the CPU, or an integrated GPU, the bucket transferring process is omitted since the integrated GPU shares the same physical address space with the CPU; thus, there is no need for transferring data to a dedicated DRAM.

Each ring buffer bucket is a batch of various input data chunks and it is constructed as a set of arrays, using one to contain the actual data and the remaining to hold the metadata needed for the processing. The structure of a data bucket is presented in Figure 5.3. In order to utilize the SIMD capabilities of the available SIMT architectures, we choose to utilize the vector data types offered by the two parallel computing platforms selected for the implementation of our engine. The use of vector data types can increase the overall performance via vectorized memory accesses. The architectures provide hardware sup-

port for these vector data types via special 128- and 256-bit registers, as well as dedicated SIMD-capable ISA extensions. For example the `int4` data type can hold four 32-bit signed integers using a 128-bit register. Since we handle the input data as single-byte characters, we can exploit vector types, like `int4`, in order to simultaneously fetch 16 input characters and store them on the available 128-bit or 256-bit registers. However, the appropriate use of these features requires special handling. The allocation of the arrays inside the data bucket, as well as all accesses to them, should be memory aligned.

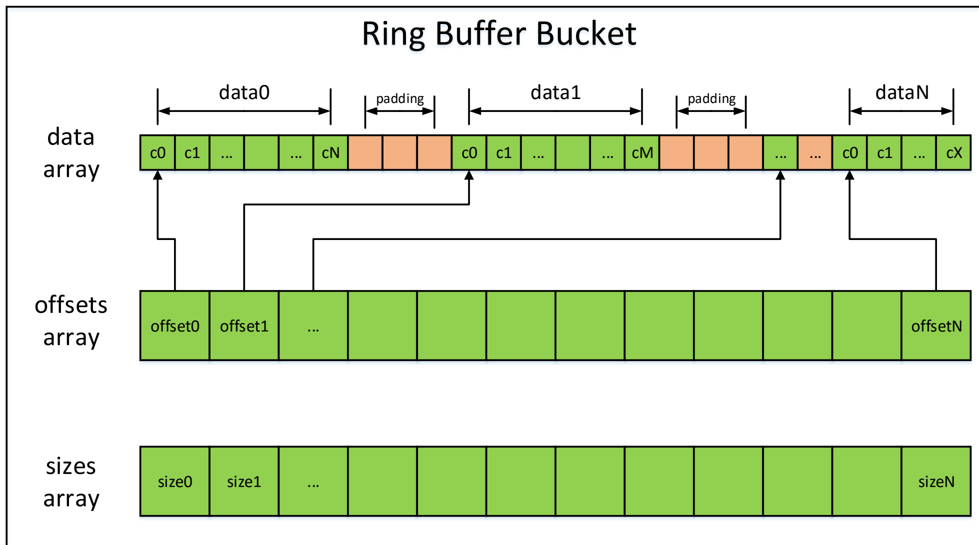


Figure 5.3: Overview of a ring buffer bucket. The bucket contains the batch of data, concatenated into a single-dimensional character array. Padding is used to keep the offsets of each data chunk in memory aligned positions. The *offsets* array holds the offsets of the individual data chunks while the *sizes* array stores their sizes (excluding the padding).

As we will describe in the following section (§ 5.3), we choose not to handle the input data as single-byte characters but fetch and store them in `int4` vectors in order to exploit the vectorization of the memory accesses. The batching of the input data into the buffer bucket is done as following. Once a chunk of data is issued for processing, the *data transfer* module saves the data bytes into the *data* array of the bucket. Since we use the `int4` vector type for data handling, the size of each data chunk should be aligned to 16 bytes (the number of characters each `int4` type can hold). If the size of the input data is not 16-bytes aligned, an appropriate padding is used. After the data is successfully stored, the module saves its actual size, without the padding, to the *sizes* array, as well as the offset of its first byte inside the *offsets* array. When other data chunks arrive to the buffer, the same process is followed and the data is stored in the *data* array, starting from the byte following the last padding byte. In this way, all data chunks inside the *data* array start from a 16-byte aligned position, making their memory access vectorization possible, while they are fetched and processed.

When a batch is complete, it is transferred to the selected processing unit in order to be scanned for matches. A batch is complete when as many data chunks as the available SIMT threads are successfully stored. If a discrete GPU is chosen as processing unit, the *data transfer* module sets a DMA with the graphics card. An appropriate memory space is preallocated in the GPU in order to house the copy of the buffer bucket. The full bucket is transferred via the PCIe bus to the graphics card's DRAM. If the execution is performed on a CPU, using OpenCL, the *data transfer* module does not create a DMA channel but rather utilizes the structures provided by the OpenCL platform for fast data processing. In this case, the ring buffer solely resides in the host system's DRAM and the bytes are fetched by the CPU using the appropriate SIMD instructions (integrated GPUs share the same physical address space with the CPU and they are not equipped with a dedicated DRAM).

When the processing of a buffer bucket is complete, a buffer containing all the appropriate results is returned back to the host process. The data chunks are never transferred back to the host process since no transformation is performed on them and their initial copy already exists in the memory space of the caller process.

5.3 String Matching

One of the key functionalities of our engine is the fixed string and binary signature matching. The hardware architectures and computing platforms selected for the development of our engine allow the simultaneous processing of multiple input streams. Each data stream is matched against large sets of strings with a single pass over the input bytes. This is achieved by utilizing the Aho-Corasick string matching algorithm described in Chapter 2. The main principle of the Aho-Corasick algorithm is that all patterns, fixed strings and binary signatures, are compiled into a single DFA. Each byte of the input data stream moves the current DFA state to the next correct state. A match is achieved when a state, marked as final during the construction of the DFA, is encountered as the current state. The process of advancing through the state machine using one byte of the input stream at a time continues until the whole input is consumed. The Aho-Corasick algorithm seems to be a perfect fit for the data parallel execution of our engine, since no backtracking on the input data is required and the process of acquiring the next DFA state lacks control flow instructions that could lead to thread divergence.

5.3.1 Readable Binary Signature Support

The definition of the Aho-Corasick algorithm treats each pattern character as a literal upon the construction of the DFA. In order to provide support for human readable binary signatures and patterns where clear-text is mixed with unprintable characters, such as command characters, we define the following syntax. The user is able to represent binary patterns by declaring each byte with its hexadecimal value, following the commonly used `0x` notation, surrounded by pipes. For example, the pattern "hers\n", where \n represents the new line feed, can be declared as "hers|0x09|", with 0x09 representing hexadecimal

value of the ASCII new line character. This syntax can be particularly useful when our engine is used in order to perform deep packet inspection (DPI), where the various packet rules contain a mixture of text and unprintable bytes. Very large binary patterns, such as virus signatures that do not need to be in a human readable format, can be given as an input to the engine "as-is".

5.3.2 DFA Representation

In order to achieve data parallelism, we develop a variation of the standard Aho-Corasick implementation, found in many searching tools such as `grep`. In most implementations, the state machine (DFA) is constructed as a tree with each node containing information about the state it represents, as well as various metadata. Since complex data structures using pointers are not an appropriate fit for data parallel platforms, we choose to represent the DFA as a serialization of the state machine tree to a single-dimensional integer array. In order to make the process of constructing this array easy to follow, we will describe the procedure using a two-dimensional array as the DFA representation. We will use the tree of Figure 2.1, produced by the patterns {he,she, his, hers}, as example.

During the bootstrap phase of our engine, the various signatures are processed by a simple parser. The purpose of the parser is to identify the signatures and process the binary notations described in § 5.3.1, if any. When all the available patterns are gathered and processed, they are compiled into a single Aho-Corasick DFA, constructed as a tree. The next step is to serialize the produced tree as a two-dimensional integer array. This array will have 256 columns, which represent the size of the ASCII set, and N rows, where N is the number of the states in the DFA. Each row represents a DFA state and each cell contains the number of the next valid transition, corresponding to the ASCII character that the cell represents. An example of this array can be found in Figure 5.4.

ASCII set (columns) / **States** (rows)

States	0	...	e 101	...	h 104	i 105	...	r 114	s 115	...	255
0	0	0	0	0	1	0	0	0	7	0	0
1	0	0	-2	0	1	5	0	0	7	0	0
2	0	0	0	0	1	0	0	3	7	0	0
3	0	0	0	0	1	0	0	0	-4	0	0
4	0	0	0	0	8	0	0	0	7	0	0
5	0	0	0	0	1	0	0	0	-6	0	0
6	0	0	0	0	8	0	0	0	7	0	0
7	0	0	0	0	8	0	0	0	7	0	0
8	0	0	-9	0	1	5	0	0	7	0	0
9	0	0	0	0	1	0	0	3	7	0	0

Figure 5.4: The state table produced by the serialization of the Aho-Corasick DFA as a two-dimensional integer array. Negative values indicate final states.

5.3.3 Input Processing

In order to traverse the serialized DFA tree, the string matching task starts from state 0 (row 0) and selects the appropriate column, according to the ASCII value of the first character of the input. In this cell, it finds the next valid state, which is located in another row of the array. Then, it fetches the next character from the input and moves to the cell pointed to by the row given in the previous step and the column given by the ASCII value of the current character. The final states in the array are annotated with a negative sign. When the task hits a negative state, we know that a match has been successfully found. Then, the search is continued using its absolute value for the next step. The fail states either point the matcher to a previous valid state or to the initial state 0.

In practice, this array is single-dimensional and all the rows that we mentioned earlier are concatenated. Since the size of every row is 256 integers, the `goto` function traverses the array as follows: `state = dfa[state * 256 + char_ASCII_value]`

Once the DFA is constructed and serialized as a single dimensional integer array, it is transferred to the external GPU or pined in the appropriate memory spaces, if a CPU or an integrated GPU is used as the processing unit of our engine. The process of constructing, serializing and pinning the DFA is executed during the initialization phase of the engine. The data processing is performed on a thread-per-data-chunk fashion. Each thread is assigned a data chunk from the data buffer, described in § 5.2.

The process of assigning the various data chunks in the batch (buffer bucket) to each thread is done as follows. Every thread has its own unique ID, starting from zero up to the number of available threads. During the batching process, the *data transfer* module stores exactly as many data chunks as the available SIMT threads, so every thread can be assigned with a different data chunk. Each thread, using its unique ID as index, fetches the size of the data it is going to process from the bucket's *sizes* array as well as its starting offset from the *offsets* array. Using the offset as index for the *data* array, it starts fetching and processing the input. Each thread fetches the data using the `int4` vector data type. This vector type is designed to hold four 32-bit signed integers; however, via appropriate casting we can exploit it to store 16 single-byte string characters. An external loop prefetches the appropriate data using the `int4` vector, thus storing 16 bytes per loop to a 128-bit register, used by the architecture to support the vector type. This process is repeated for `size[thread_id] / sizeof(int4)` times. An internal loop unpacks the vector and processes each one of the 16 characters individually. Data prefetching using vectorized memory accesses can significantly improve the performance of our *string matching* module, by exploiting the SIMD characteristics of the underling SIMT architecture. The nested loops are used for data prefetching and vector unpacking, thus, the complexity of the string matching algorithm is not increased to O(n^2).

Every time a thread discovers a match, it saves the required information (data chunk ID, matched pattern, matching offsets, etc.) to another buffer. When the processing of a batch is complete, the *data transfer* module returns the (batched) results to the host process. By utilizing the properties of the Aho-Corasick DFA, along with the data parallel execution that SIMT architectures offer, our *string matching* module is able to simultaneously match a large set of patterns against a large set of input data streams.

5.4 Regular Expression Matching

The second basic functionality provided by our engine is the one of regular expression matching. The *regex matching* module, responsible for this process, operates similarly to the *string matching* module described in § 5.3. The various threads process the data buffer in by advancing through the states of a serialized DFA, consuming a single byte of the input in each transition. The results are also reported to the host process in the same manner. However, the operation of matching regular expressions differs during the construction of the DFA and the consumption of the input bytes during processing.

5.4.1 DFA Construction

Parsing the regular expressions requires a sophisticated parser, able to understand and interpret their syntax. Our parser supports POSIX regular expressions, as well as various escape characters and character classes, borrowed by other implementations such as Perl. After the parsing is complete, our engine compiles the various regular expressions into a singe NFA. The NFA is then transformed to a fully expanded DFA and serialized in the same fashion as the Aho-Corasick DFA, described in § 5.3.2.

5.4.2 Data Processing and Backtracking

The processing of the input data is performed by advancing through the DFA in the same manner as during simple string matching. The major difference between the two modules is that the *regex matching* module *backtracks* on the input data, if required. The algorithm keeps track of the position of the character that caused a successful transition through the DFA. When a thread encounters a fail state, after a sequence of successful transitions, it backtracks on the input data and re-attempts the match starting from the character after the one that caused the first successful transition. We choose to implement a backtracking *regex-directed* engine (described in Chapter 3 § 3.3.1) in order to support various important features, such as *backreferences* and *lazy quantifiers*. Even though backtracking diverges from the optimal execution path, that being processing the whole input data with one pass over each byte, it can be controlled via *possessive quantifiers* and *atomic grouping*. *Possessive quantifiers* prevent the engine from trying all the permutations in order to match the input data streams. However, the task of minimizing the backtracking events lays upon the author of the regular expressions and it is not an automated process carried out by the engine.

5.5 String Assisted Regular Expressions

Compiling together multiple complex and large regular expressions into a singe DFA may produce a very dense automaton that drives the *regular expression* module into more frequent backtracking, causing significant performance degradation. In order to mitigate this problem, we introduce the concept of *string assisted regular expressions* via custom searching rules. The basic idea behind string assisted regular expressions is that we

pair each regular expression with a fixed string, deriving from the regular expression, and a unique identifier. Then, we compile all the fixed strings into a single Aho-Corasick DFA and each regular expression into its own DFA. We process the input using the *string matching* module and the strings found in the various rules. The regular expression matching is performed only when the simple string, paired with the regular expression in the same rule, matches the input. In this way, we can filter out most of the data chunks and perform the regular expression matching only to those that triggered their paired simple strings or binary signatures.

5.5.1 Searching Rules

In order to support the concept of string assisted regular expressions, we define the syntax of custom *searching rules*. Each rule is assigned with a unique ID and consists of a set of fixed string and binary patterns, a regular expression and a message. A valid rule should contain at least one fixed string or regular expression, while the message field can be omitted. The message field can be occupied by a text message, reported by the engine upon the successful match, or be replaced with an event handler. The matching process is performed in two phases and when a rule is successfully triggered the corresponding text message or event are triggered as well. Our engine is capable of processing standalone patterns (strings or regular expressions) along with a set consisted of different rules.

The various fields of a rule are distinctive via their tags. An example of a searching rule is the following: `str:"her", re:"[0-9]her[0-9]4", msg:"Rule 4 triggered"`. The rule indicates that the fixed string "her" should be searched using the *string matching* module and if it is found in a data chunk, this data chunk should be matched against the regular expression `[0-9]her[0-9]4` by the *regex matching* module. Upon a successful match of both the fixed string and the regular expression, the engine must forward the message "Rule 4 triggered" to the user. In order to generate the automata needed for the execution of the two scanning phases, a rule-set preprocessor parses the rule files and separates the patterns according to their tags.

In the first phase of the matching process, we compile every single string and binary signature found in our rule-set into a single Aho-Corasick DFA. The processing of the input data is performed as described in § 5.3. In the second matching phase, we use the regular expressions paired with the patterns of the previous phase, if present, in order to perform a more thorough matching. The *regex matching* module compiles each regular expression found in the rule set, individually, in separate DFAs.

The paring of the simple string patterns with the regular expressions is done as follows. A NxM regex ID array is kept, where N is the number of states of the Aho-Corasick DFA and M is the maximum number of patterns found ending in a single state of this DFA. We also keep an array that contains all the regular expression automata, ordered by rows, according to the ID of the rule they belong to. Using the number of a final state, extracted upon a match in the first matching phase, we are able to index the corresponding row of the regex ID array. In each column we can find the IDs of the regular expressions that correspond to the patterns matched in the first phase. If the simple string pattern that triggered the second matching phase does not have a corresponding regular expression,

the IDs in the table are marked with the value -1 and the second phase is terminated. Then, we return to the first phase again. Using those IDs we can fetch the appropriate regular expression state table using them as direct indices for the regular expression DFA array. The second phase of the matching is performed using these state tables on the entire data chunk. Once the process is completed, we return to the first matching phase in order to complete the matching to the rest of the data. This procedure stops when all the bytes of the data chunk are processed.

5.5.1.1 Reporting and Events

Reporting is performed using the messages provided in the rule-set. These messages are stored in an array, sorted according to the ID of the rule they belong to. Upon a successful match, either in the first or the second matching phase, the processing of the corresponding rule is considered complete and the appropriate message is selected from the messages array by indexing it with the rule's ID. Those messages, handled by the *event* module, can either be strings or simple error codes and are forwarded back to the user along with the ID of the data chunk that triggered them. Optionally, the messages can be interchanged with events. In this case, a function dispatcher table is used –responsible to dispatch tasks upon a successful match of a rule. The indexing of this table can be achieved using the error codes. An overview of the three phases described can be found in Figure 5.5.

5.6 API

The functionality of our engine is exposed via a C API. The API hides the specifics of the platforms deployed for the development of the engine, making its use and integration in other applications very simple. In order to extend the use cases of our engine, we also provide a Java API –developed using JNI wrappers for the original C version– targeting the vast majority of desktop, server and mobile developers. The process hosting the engine needs to initialize it, provide the various rules and patterns as input and then start feeding the data buffer.

The initialization can be performed using the functions provided by the API in two ways. The first option is to call the default initializer. This will spawn a task that searches for the available hardware and programming platforms, pick a processing unit and set default properties to the various modules. The task opts to find the hardware with the maximum performance capabilities. For this reason, the hardware will be initialized using the maximum number of available threads and the number of data chunks per bucket will be set to this number. The second option is to initialize the engine using a configuration file. This option is tailored for users experienced with parallel programming and aware of the properties of the hardware in use. The configuration file can be used to change many parameters affecting the behavior of the engine. For example, a specific processing unit can be selected, as well as the number of threads. The properties of the ring buffer can also be altered in this phase. For example, the size of the data chunks can be customized properly in order to achieve maximum matching throughput or minimum latency.

The API also offers various ways, to the user, in order to provide the patterns to the engine. The patterns can be stored and retrieved "as-is" from one or multiple files. There is also the option to provide the rule-set described in § 5.5.1 in the same manner. Finally, the patterns or the rule-set can be retrieved from memory (e.g. character arrays), targeting applications that construct custom patterns at runtime or receive them via data channels (e.g. network sockets).

The data chunks are handled as character arrays and they are stored to the ring buffer using the available API functions. The batching of the data and the handling of their metadata is performed by the engine, thus being completely transparent to the developer. The API also mitigates the need to explicitly launch the processing tasks once a bucket is full, since this process is automated by the ring buffer. However, the option of triggering a matching process on a bucket –whether it is complete or not– is also provided, targeting latency intolerant applications that need to deploy watchdog functionalities upon the handling of their input data.

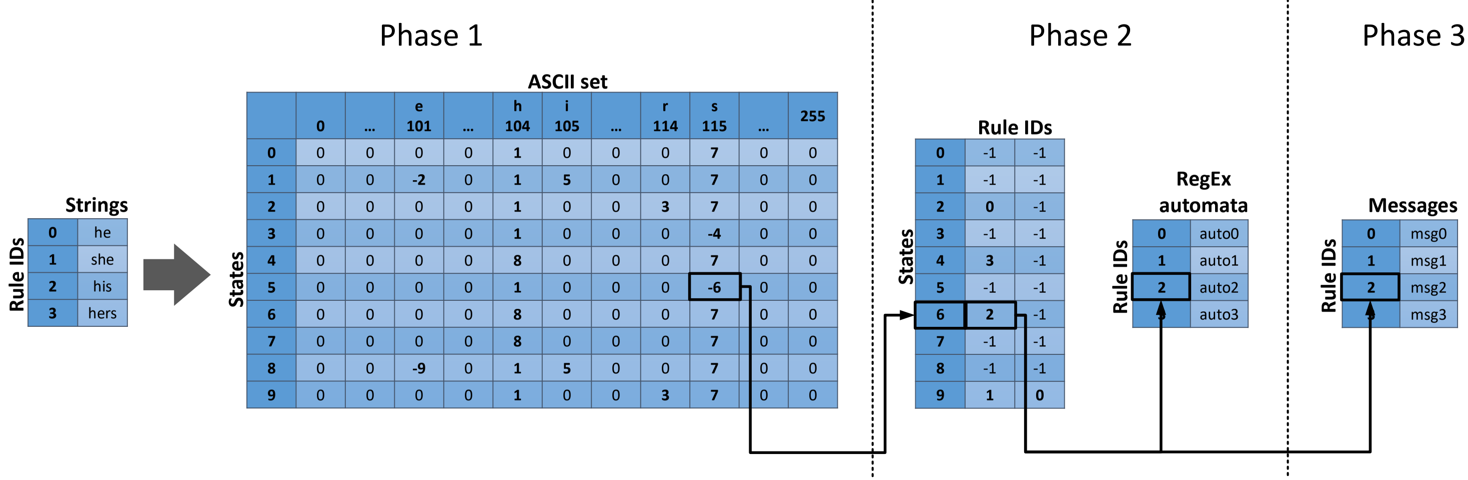


Figure 5.5: Overview of the tree matching phases. When the task of the first matching phase encounters a negative state, a match has been successfully found. In the second phase, we use the absolute value of this state in order to index the table that pairs the simple strings with their regular expressions. We use those IDs so as to index the array housing the regex automata. If a match using these automata is found, we use again this ID in order to index the array housing the rule messages.

Chapter 6

Evaluation

In this chapter, we present the evaluation of the string pattern matching framework that we develop for this work, using various workloads as described in the following sections (§ 6.2). Firstly, we present the properties of the produced DFA (§ 6.3). Then, we conduct a number of benchmarks using a single-threaded implementation of our engine in order to draw a base line for the evaluation of our parallel engine (§ 6.4). In section § 6.5, we present the evaluation of our system when executed on the NVIDIA GTX 980 GPU. Finally, in section § 6.6, we describe the performance characteristics of our engine when executed on the Intel Xeon E5-2697 processor.

6.1 Experimental Testbed

In this section, we describe the underlying hardware setup. Our base system is equipped with one Intel Xeon E5-2697 v2 processor and one NVIDIA GeForce GTX 980 graphics card. The processor contains twelve CPU cores operating at 2.7GHz, with hyper-threading support, that provides twenty-four hardware threads. The processor's Thermal Design Power (TDP) is 130 Watt. The system is equipped with 32GB of quad-channel DDR3 DRAM at 1866MHz. The GTX 980 has 2048 CUDA cores and 4GB of GDDR5 memory. The GPU is rated at 5040 GFlops and its TDP is 200 Watt. The system is running Arch Linux with stock kernel (version 4.9.7) and no modifications are performed to the operating system's kernel or the clock ratings of the various components. Our engine is developed using gcc version 6.3.1, CUDA toolkit version 8.0 (using the proprietary NVIDIA driver version 375.26) and OpenCL version 1.2.

6.2 Workloads

The key components of the two processing modules, the *string matching* and *regex matching* modules, are the DFAs they use in order to scan the input data streams. The number of signatures compiled in the same DFA, the range of the ASCII set they cover, as well as the complexity of the regular expressions, play a significant role in the characteristics of the DFAs, which directly impact the performance of our system. In order to examine

how the various signatures affect the characteristics of the automata and the performance of our system, we design two different workloads, as described below.

6.2.1 Synthetic ASCII Workload

This workload, contains four sets of 10, 100, 1000, and 10000 patterns respectively. Those patterns randomly vary in size from 5 to 20 bytes each. Every one of these signatures contains random characters covering the entire ASCII set. The purpose of this workload is to examine how the increasing number of signatures affects the size of the produced DFA and how this impacts the overall performance of our system. In order to put the sets of this workload to the test, we craft four synthetic data traces containing data chunks of random ASCII characters, again covering the entire ASCII set, one for each pattern set.

We choose to generate this workload using random ASCII characters for two reasons. Firstly, pattern sets containing signatures with random characters, covering the entire ASCII set, minimize the probability of including various signatures that share common suffixes or prefixes. This might happen with pattern sets containing natural language signatures (i.e. English words). In a DFA perspective, this means that no common subtrees will be constructed, thus the DFA will have bigger density, that is every state will have as many valid transitions as possible. Secondly, data chunks containing random ASCII characters ensure that during the evaluation, the processing task threads will perform the worst case random accesses to the DFA memory since no common sub-trees (paths in the DFA) exist in order to be cached. Moreover, we randomly inject the various signatures to various data chunks in order to be sure that all states of the DFA will be visited during the processing of the input. This eliminates the probability of an input trace containing data chunks that fail after very few transitions, allowing the first n states of the DFA to be cached.

Using this workload, we stress the engine to the maximum extent since we cover the worst possible scenario, something that it is unlikely to happen in a real use case.

6.2.2 Huge Regular Expressions Workload

In order to evaluate the performance of our *regex matching* module, we develop a workload containing 100 huge regular expressions, varying in size from 488 bytes up to 1 Mega bytes (considering the number of characters they consist of). Since the process of generating random regular expressions is not straightforward, we manually construct the patterns using regular expression generators. These regular expressions are grouped in four different categories, paired with an appropriate input trace file.

The **first set** consists of regular expressions containing many character sets and ranges, while the trace file paired with this set contains very few data chunks that match the expressions. The purpose of this set is to emulate a use case where simple regular expressions are used to extract alphanumerics from the data stream while in most cases no matches exist and minimal backtracking takes place.

The **second set** contains regular expressions that aim to extract various fields from public records such as names, addresses, bank account numbers and social security num-

bers. This regular expression set is paired with a data input of various fake records containing multiple matches. The purpose of this set is to impose greater stress on the engine, opposed to the first set, and emulate cases where multiple matches occur in every single data chunk.

The **third set** contains regular expressions similar to those of the **second set**, enhanced with fuzzy matching functionality. Fuzziness can be achieved by extending the regular expressions in order to contain multiple permutations of the various patterns they try to extract using the *dot*, *star*, *question mark* and *plus sign* metacharacters. The data input of this set contains records similar to those of the previous set, where multiple fields are misspelled and matched by one of the permutations found in the regular expressions. This workload aims to emulate the use case of the second workload when more severe backtracking needs to take place.

The **fourth set** contains a few very complex regular expressions, containing multiple metacharacters and very big ranges. The input data used for the evaluation of this set contain matches found after severe backtracking. The combination of regular expressions of this set along with their paired data input dive the *regex matching* module to severe and constant backtracking. The purpose of this workload is to evaluate our engine under the worst possible scenario.

We construct the regular expressions described in this section by combining multiple sub expressions in each one, thus making them grow dramatically in size. The DFA compiler handles multiple regular expressions compiled together in the same way it handles multiple sub expressions found in a single regular expression. For this reason, the DFA produced by each expression has the same properties as an automaton constructed by multiple smaller regular expressions with the same characteristics.

In order to stress the performance of the *regex matching* module even further, we construct two DFAs by randomly combining 25 and 50 regular expressions found in the sets described above. We also evaluate the performance using these two DFAs with data streams containing data chunks found in the data traces of all four sets. In this way we can measure the performance of our engine when multiple regular expressions with different characteristics are combined in a single automaton, used to process data streams with different characteristics.

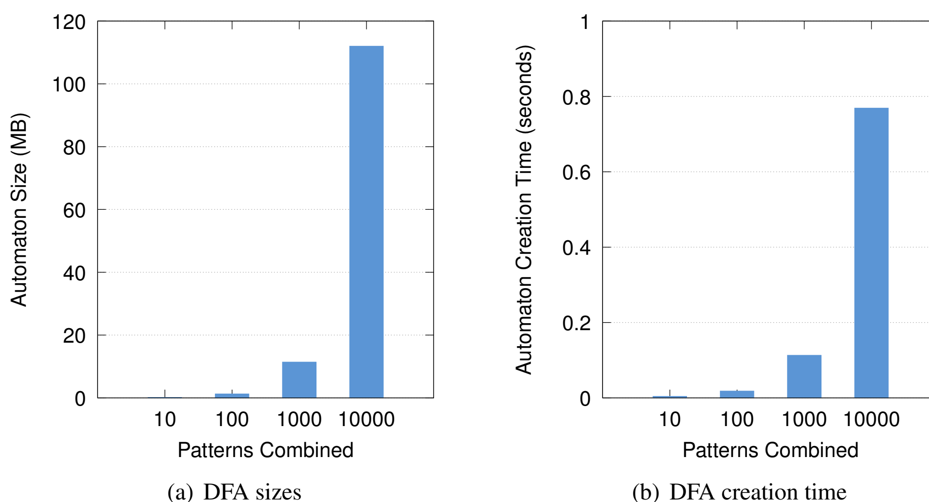
6.3 DFA Properties

In this section, we describe the properties of the DFAs produced by compiling the various pattern sets found in the two workloads described sections § 6.2.1 and § 6.2.2.

6.3.1 Simple String DFAs

Firstly, we compile the four pattern sets found in the **Synthetic ASCII** workload described in section § 6.2.1. Each set contains 10, 100, 1000 and 10000 signatures, constructed by random characters covering the entire ASCII set. Figure 6.1 displays the characteristics of the four DFAs. In Figure 6.1(a) we can see that the size of each DFA is proportional to the

number of patterns it contains. Figure 6.1(b) shows the time needed for the construction of the four automata. The compilation of the biggest DFA, containing ten thousand patterns, is under one second, indicating that the recompilation of the automata is feasible during the execution of the engine. This property can be particularly useful in use cases where patterns need to be added to or removed by the system at runtime.



(a) DFA sizes

(b) DFA creation time

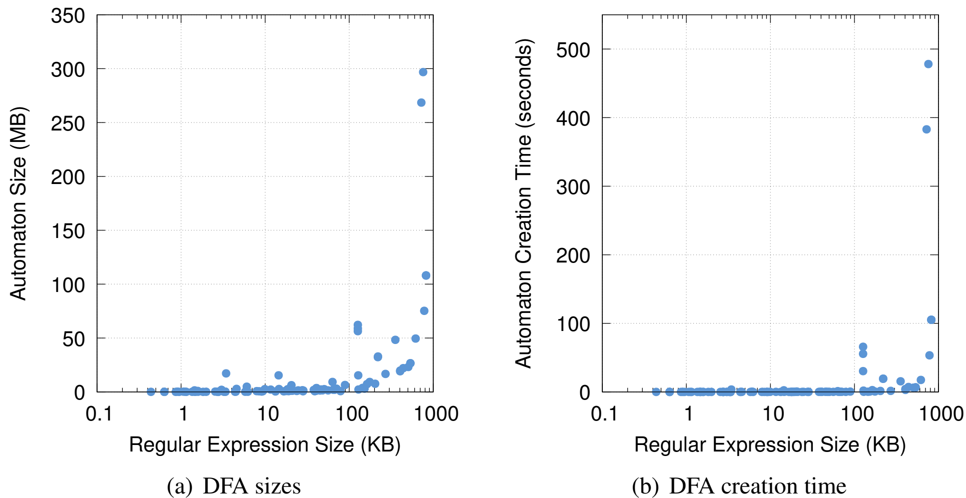
Figure 6.1: Characteristics of the four DFAs produced by the four pattern sets found in the **Synthetic ASCII** workload. The x-axis of both plots indicates the number of regular expressions combined into a singe DFA. Figure 6.1(a) displays the size of the produced DFAs, while Figure 6.1(b) displays the time needed for their creation. We notice that the automaton size and its creation time are proportional to the number of regular expressions combined.

6.3.2 Regular Expression DFAs

In order to examine the properties of the DFAs produced by compiling regular expressions, we construct one DFA for each one of the 100 regular expressions found in the **Huge Regular Expressions** workload, described in section § 6.2.2. Figure 6.2 displays the sizes of the DFAs and the time needed for their construction with respect to the size of the regular expression. As we can see in Figure 6.2(a), the size of the produced automaton is not always proportional to the size of the regular expression. This happens due to the fact that the expressive power of metacharacters and character classes leads to the construction of multiple states. We can also observe that some regular expressions produce automata in the order of 250 and 300 MBytes. These regular expressions have high complexity and contain multiple permutations of their sub-expressions as well as huge ranges, resulting to a significant growth of the DFA size when all possible states are constructed. In most cases, the size of each automaton is less than 1Byte's.

The time required for the construction of the various DFAs is displayed in Figure 6.2(b).

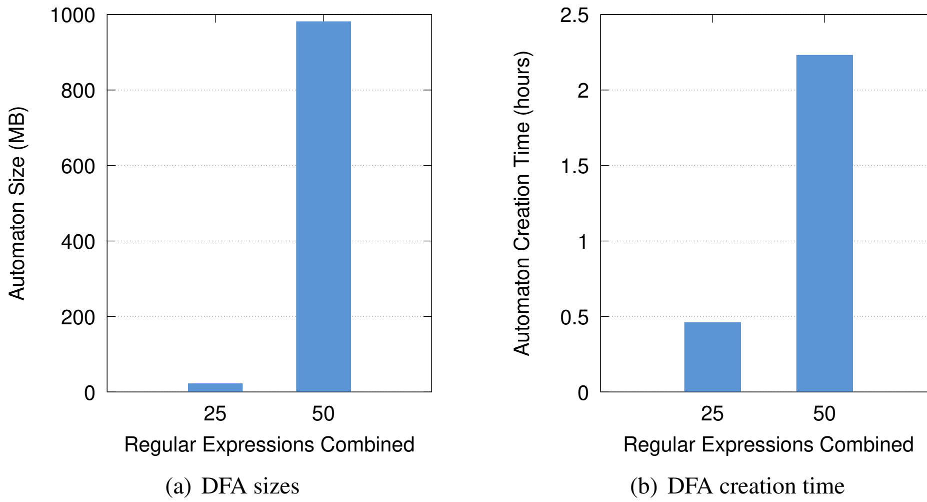
We notice that, in most cases, the distribution of the required time follows the distribution of the automaton size. The compilation time remains well under one second for 74 out of 100 regular expressions, with a maximum value of 480 seconds for the biggest and most complex regular expression. After taking into consideration that the regular expressions in this workload are considerably bigger than those found in most applications (such as NIDS, Antivirus,etc.), we conclude that it is feasible to recompile the regular expressions during runtime, if required by the application using our engine.



(a) DFA sizes (b) DFA creation time

Figure 6.2: Characteristics of the one hundred DFAs produced by the regular expressions of the **Huge Regular Expressions** workload. The x-axis in both plots indicates the size of the various regular expressions. Figure 6.2(a) displays the size of the DFA produced by each regular expression, while Figure 6.2(b) displays the time needed for its creation. We notice that the size of the DFA as well as its creation time are not always proportional to the size of the regular expression. Complex regular expressions produce larger DFAs or require more time to be compiled.

We conclude the analysis of the various DFAs by compiling together 25 and 50 random regular expressions, picked up by all four regular expression categories described in section § 6.2.2. Figure 6.3 displays their properties. As we can see in Figure 6.3(a) the sizes of the produced DFAs are not linear to the number of regular expressions combined together. The same property also applies to the required creation time, shown in Figure 6.3(b). The compilation time as well as the size of the automaton can not be approximated a-priori, since they are a function of the complexity of the individual regular expressions and the way their states are distributed inside the DFA. In our case, the sizes vary from 20 Byte's up to almost 1 Byte and the creation time reaches up to over two hours. Construction of such complex DFAs is most likely unreal for the majority of applications. However, in our case, they are particularly useful in order to stress the performance of the *regex matching* module to the maximum extent.



(a) DFA sizes (b) DFA creation time

Figure 6.3: Characteristics of the DFAs produced by combining 25 and 50 random regular expressions found in the **Huge Regular Expressions** workload. The x-axis in both plots indicates the number of regular expressions combined in a singe DFA. Figure 6.3(a) represents the size of the produced DFAs, while Figure 6.3(b) displays the time needed for their creation. We notice that the automaton size and its creation time are not linear to the number of regular expressions combined.

6.4 Baseline

In order to draw a baseline for the evaluation of our massively parallel matching engine, we also develop a single-threaded CPU version of the framework, implemented using C. We first evaluate the performance of the single-threaded CPU version of the *string matching* module. It is also essential to examine how the size of the DFA used for the data processing affects the performance of our system.

We evaluate the throughput of our system using the four compiled automata produced by the pattern sets, found in the **Synthetic ASCII** workload, against their matching synthetic data traces. We observe that the linear increase in the size of the automaton results in a logarithmic decrease of the system's sustained throughput, as shown in the first plot of Figure 6.4. However, this throughput degradation contradicts with the theoretical characteristics of the Aho-Corasick algorithm. In order to understand this behavior, we plot the number of cache references required for the processing of the four automata. We observe that the linear increase of the automaton size results in a logarithmic increase of the cache references required to process the automaton. This increase is the exact reverse of the throughput decrease.

We also conduct the evaluation of the single-threaded version of the *regex matching* module using the various regular expression sets and trace files found in the **Huge Regular Expressions** workload. Figure 6.5 shows the results of the evaluation. Figure 6.5(a) depicts the sustained throughput achieved by the engine when processing each individ-

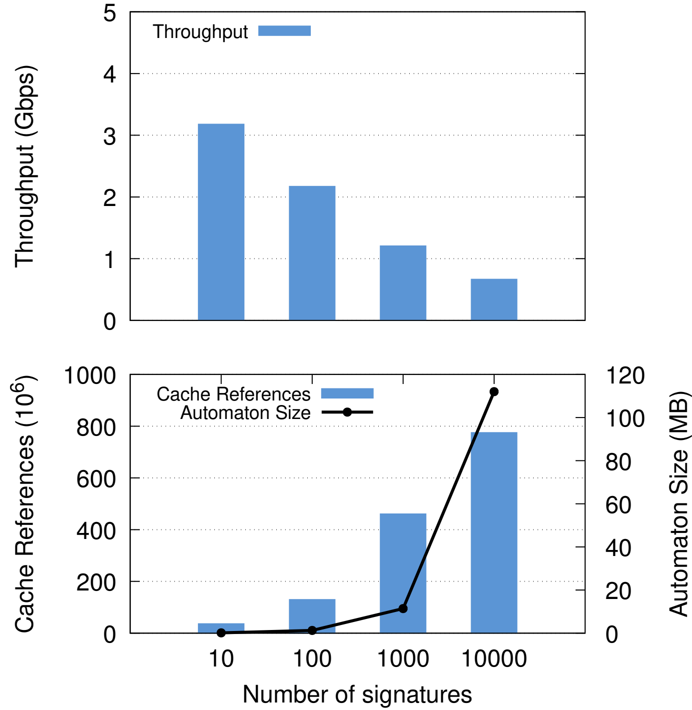
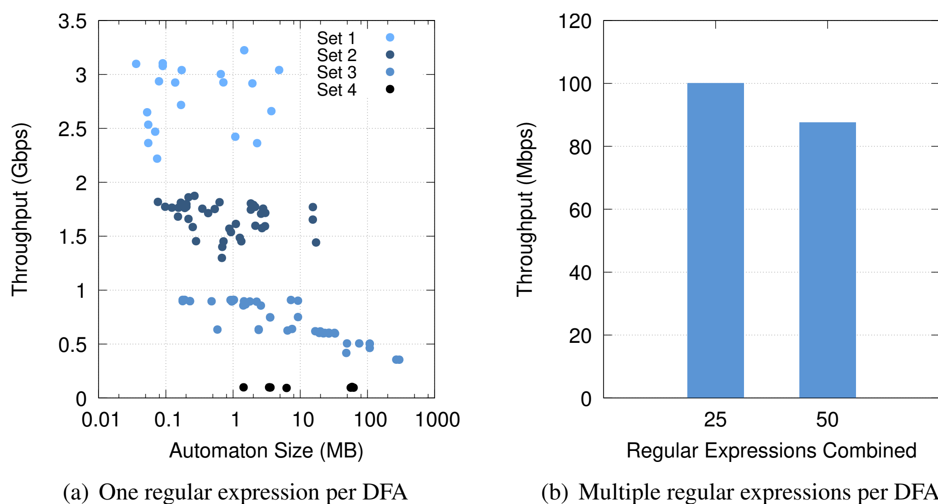


Figure 6.4: Sustained throughput achieved by our single-threaded CPU implementation of the *string matching* module using the automata produced by the four pattern sets found in the **Synthetic ASCII** workload. As the number of signatures increases, the size of the automaton grows. Bigger automata result to an increased number of cache references, imposing a performance penalization. Note that the x-axis is in log-scale.

ual DFA separately. All the regular expressions of the same set are evaluated using the corresponding input trace file. We notice that the expressions of each set tend to have similar performance characteristics. Specifically, we observe that regular sets requiring more backtracking over the input data in order to locate the various matches tend to have lower throughput. Moreover, the regular expressions of the fourth set, which lead the *regex matching* module to constant and severe backtracking, achieve very low throughput. On the contrary, the maximum performance is achieved by the regular expressions of the first set, which require minimum or no backtracking on the data. Finally, in Figure 6.5(b) we can see the sustained throughput achieved by our engine when processing input data found on all the data traces used to evaluate the four regular expression categories. We notice that the overall performance is severely penalized since both DFAs contain regular expressions of the fourth regular expression set and the data input contains data chunks that trigger the constant backtracking. We can also see that the difference between the performance achieved using the two DFAs is not significant, considering that (i) the second automaton contains twice as much regular expressions from all four categories and (ii) its size is almost 45 times bigger that the size of the DFA containing 25 regular expressions.



(a) One regular expression per DFA (b) Multiple regular expressions per DFA

Figure 6.5: Sustained throughput achieved by our single-threaded CPU implementation of the *regex matching* module. Figure 6.5(a) displays the processing throughput using one hundred DFAs produced by the regular expressions of the **Huge Regular Expressions** workload, while Figure 6.5(b) displays the performance achieved using DFAs combining 25 and 50 random regular expressions of the same workload respectively.

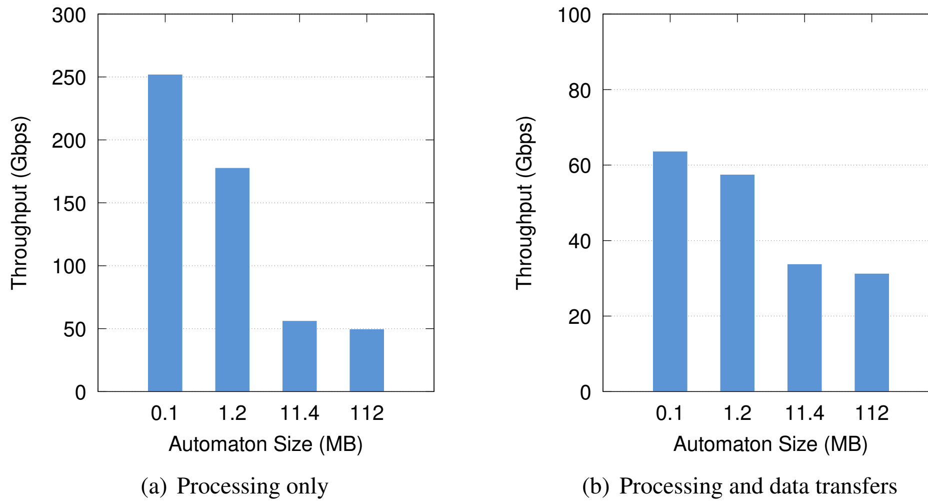
6.5 Graphics Accelerators

In this section we evaluate the performance of our parallel engine using graphics accelerators. In section § 6.5.1 we describe the performance characteristics of the CUDA flavors of the *string matching* and *regex matching* modules, executed on the NVIDIA GTX 980 GPU. In section § 6.5.2 we conduct the same experiments, as in section § 6.5.1, using the OpenCL version of both modules, executed on the same graphics accelerator for fair comparison.

6.5.1 CUDA

We start the evaluation of our system with the CUDA flavor of the *string matching* module, using the DFAs produced by the pattern sets described in section § 6.2.1. The results of this experiment are presented in Figure 6.6. Figure 6.6(a) depicts the sustained processing throughput achieved by the string matching module. Figure 6.6(b) shows the end-to-end throughput of our engine, including the data transfers from and to the GPU DRAM. We notice that the performance is decreased as the automaton grows in size from 0.1 MBytes to 1.26 MBytes due to the increasing number of cache references. We also observe that the sustained throughput stabilizes to 50Gbits/s for DFAs that do not entirely fit the size of the cache. Overall, the CUDA flavor of the *string matching* module can achieve end-to-end performance in the order of 60Gbits/s, for automata entirely fitting the cache, and over 35Gbits/s for larger automata. By comparing the maximum performance achieved by

the CUDA version against the one achieved by the single-threaded CPU implementation of the module, we can see that *our parallel engine can achieve up to 21 times higher performance*.

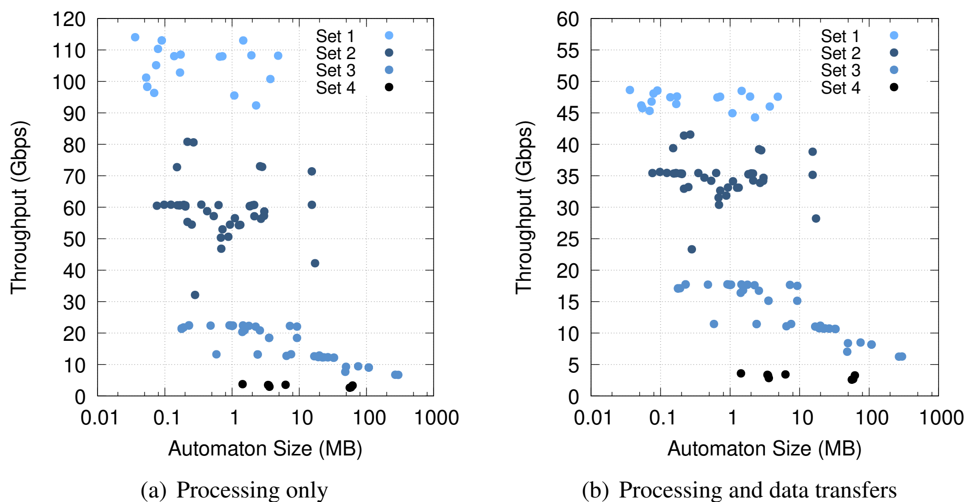


(a) Processing only

(b) Processing and data transfers

Figure 6.6: Sustained throughput achieved by the CUDA flavor of the *string matching* module using the automata produced by the four pattern sets found in the **Synthetic ASCII** workload. The x-axis in both plots represents the size of the DFAs produced by combining 10, 100, 1000 and 10000 patterns respectively. Figure 6.6(a) displays the throughput achieved by the GPU, while Figure 6.6(b) displays the end-to-end throughput, including the data transfers to and from the device. We notice that when the size of the automaton grows larger than the size of the cache, the performance is substantially decreased and remains consistent regardless of the size of the DFA.

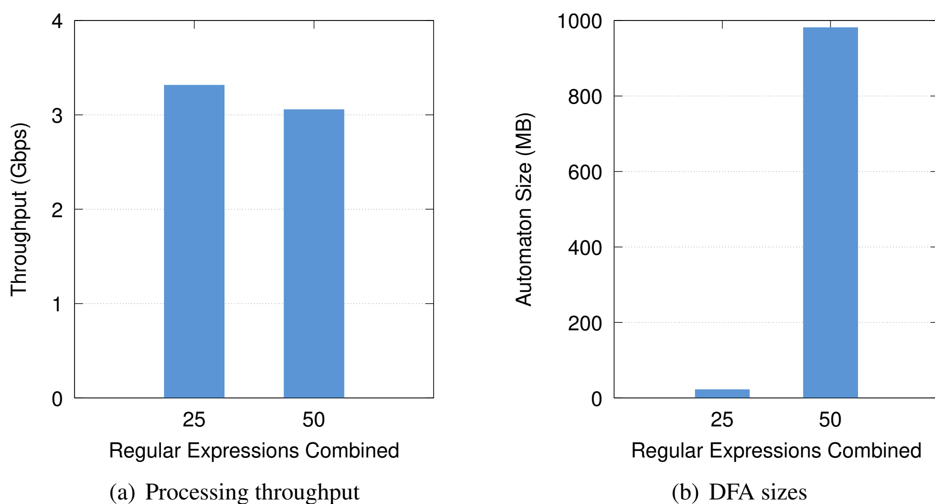
We proceed with the evaluation of the CUDA version of our engine by analyzing the performance of the *regex matching module*, using the various pattern and data sets found in the *Huge Regular Expressions* workload. In Figure 6.7, we present the performance results of the evaluation, using the 100 DFAs produced by the regular expressions of this workload. Figure 6.7(a) displays the processing throughput, while Figure 6.7(b) shows the end-to-end performance, including the data transfers via the PCIe to and from the GPU memory. As we can see, the performance characteristics of regular expressions belonging to the same sets tend to cluster around similar throughput values, with the first set achieving an average end-to-end throughput of almost 50Gbit/s. The throughput decreases as we move forward to the rest of the sets with the fourth and final set achieving an average of 4Gbps, due to the severe backtracking. In comparison to the single-threaded CPU implementation of the *regex matching* module, *the highly parallel CUDA version can achieve a performance increase in the order of 16 times*.



(a) Processing only

(b) Processing and data transfers

Figure 6.7: Sustained throughput achieved by the CUDA flavor of the *regex matching* module using one hundred DFAs produced by the regular expressions of the **Huge Regular Expressions** workload. The x-axis in both plots represents the size of the various regular expression DFAs. Figure 6.7(a) displays the throughput achieved by the GPU, while Figure 6.7(b) displays the end-to-end throughput, including the data transfers to and from the device. We notice that in most cases the size DFA does not directly affect the performance. The sustained throughput is function of the complexity of the regular expression and the amount of backtracking performed on the input.



(a) Processing throughput

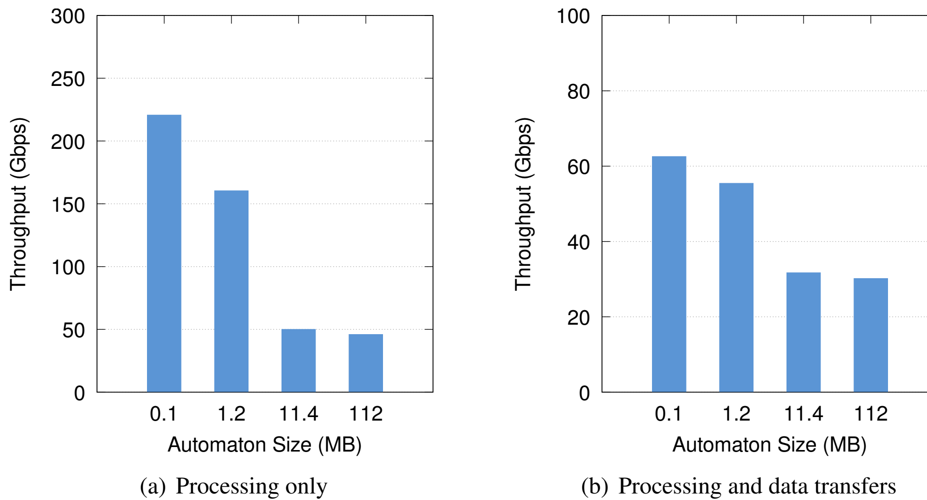
(b) DFA sizes

Figure 6.8: DFA sizes and sustained throughput achieved by the CUDA flavor of the *regex matching* module using combinations of 25 and 50 random regular expressions. Figure 6.8(b) displays the size of the DFAs, while Figure 6.8(a) displays the sustained throughput. We notice that the size difference of the DFAs is irrelevant to the number of regular expressions combined and does not affect the performance of the system

We conclude the evaluation of the CUDA flavor of the *regex matching* module by measuring the sustained throughput achieved when processing the workload containing the two DFAs combining 25 and 50 regular expressions respectively, with their paired input trace. As we can see in Figure 6.8(a), the average end-to-end throughput lies around 3.3Gbits/s and is not affected by the size of the two DFAs, displayed in Figure 6.7(b). The main reason of this performance degradation is the severe backtracking over the input, produced by the regular expressions of the fourth regular expression set compiled along the expressions of the other sets.

6.5.2 OpenCL

In this section, we present the performance assessment of the OpenCL flavor of our engine, executed on the NVIDIA GTX 980 Graphics Processing Unit, starting with the evaluation of the *string matching* module. Figure 6.9 presents the sustained throughput achieved by the module, using the four automata produced by the combination of 10, 100, 1000 and 10000 random ASCII patterns, found in the **Synthetic ASCII** workload. As we can see in Figure 6.9(a), the OpenCL flavor achieves a maximum throughput of 220Gbps, excluding the data transfers. By comparing the results of Figure 6.9(a) against those of Figure 6.6(a), we can see that the maximum performance of the CUDA flavor is almost 14% higher. The main reason behind this discrepancy is that CUDA code is optimized for the NVIDIA hardware, while OpenCL serves as a general purpose parallel computing platform. We can also observe that the "in-core" performance of both flavors stabilizes to 50Gbps, for DFAs that do not entirely fit in the cache.



(a) Processing only

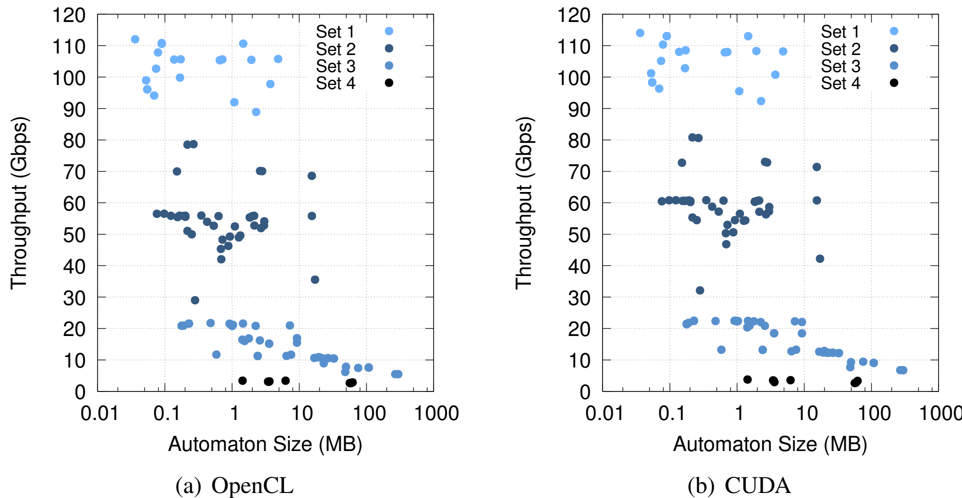
(b) Processing and data transfers

Figure 6.9: Sustained throughput achieved by the OpenCL flavor of the *string matching* module, using the pattern sets described in section § 6.2.1. The x-axis in both plots represents the size of the DFAs produced by combining 10, 100, 1000 and 10000 patterns respectively. Figure 6.9(a) displays the "in-core" throughput, while Figure 6.9(b) displays the end-to-end throughput, including the data transfers to and from the device.

Figure 6.9(b) displays the end-to-end string matching throughput achieved by the OpenCL implementation of our engine. The comparison of the results, presented in the figures 6.6(b) and 6.9(b), indicate that the difference between the performance of the two flavors is negligible, since the end-to-end throughput is normalized due to the overhead of the data transfers via the PCIe bus.

In the remaining of this section we compare the matching throughput achieved by the OpenCL version of the *regex matching* module versus the CUDA flavor, when processing the various regular expression sets described in section § 6.2.2.

Figures 6.10(a) and 6.10(b) present the "in-core" throughput achieved by the two implementations respectively. As we can see, the performance difference is less than 5%. This results comes in contrast to the 14% difference observed between the results of the two flavors of the *string matching* module. The main reason behind this behavior is that the performance gained by the optimizations performed by the NVIDIA compiler to the CUDA executable is hidden by the constant backtracking.

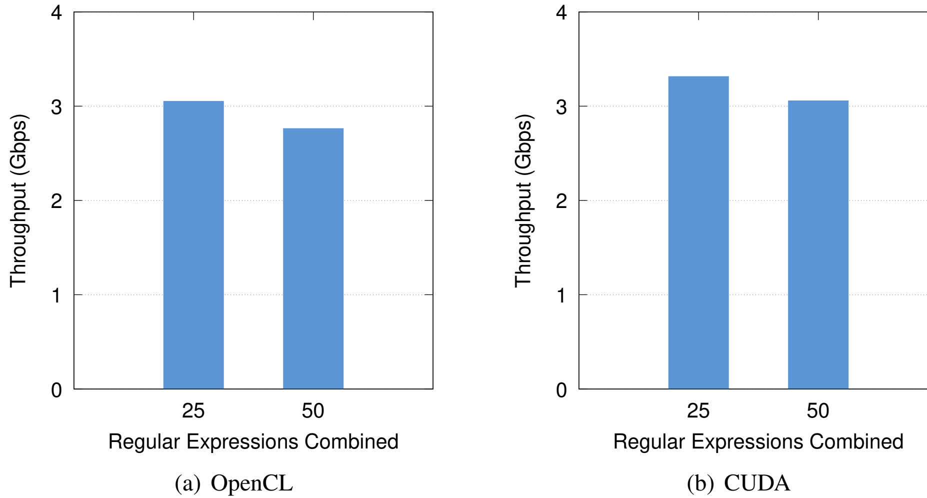


(a) OpenCL (b) CUDA

Figure 6.10: Comparison of the sustained throughput achieved by the OpenCL flavor of the *regex matching* module versus the CUDA implementation, using one hundred DFAs produced by the regular expressions of the **Huge Regular Expressions** workload. Figure 6.10(a) displays the "in-core" throughput achieved by the OpenCL flavor, while Figure 6.10(b) presents the results of the same experiment conducted using the CUDA version of the module. We notice that the difference between the throughput of the two implementations is insignificant, since the performance gained by the hardware optimized CUDA version is hidden due to backtracking.

We conclude the performance assessment of our system on Graphics Processing Units by evaluating the sustained throughput achieved by the OpenCL implementation of the *regex matching* module, using the two DFAs containing combinations of 25 and 50 random regular expressions found in the **Huge Regular Expressions** workload. Moreover, we compare these results to the ones obtained by the CUDA flavor of the module during

the same experiment. Figure 6.11(a) presents the results of the OpenCL implementation, while Figure 6.11(b) shows the performance achieved by the CUDA version. As we can see, the CUDA flavor is able to achieve slightly better throughput, due to the optimizations performed by the NVIDIA compiler.



(a) OpenCL (b) CUDA

Figure 6.11: Comparison of the sustained throughput achieved by the OpenCL flavor of the *regex matching* module, versus the CUDA flavor using combinations of 25 and 50 random regular expressions found in the **Huge Regular Expressions** workload. Figure 6.11(a) displays the performance of the OpenCL flavor, while Figure 6.11(b) displays the throughput achieved by the CUDA flavor. We notice that the CUDA version slightly outperforms the OpenCL version since it is optimized for the NVIDIA hardware.

6.6 OpenCL with Intel Processors

One of the main advantages of employing the OpenCL framework in the development of our engine is that, with slight modifications during the module compilation, we can exploit the SIMD characteristics of modern CPUs. The code is executed using multiple threads, handled by the OpenCL runtime. Each one uses the special SIMD registers and instructions provided by the hardware, in a manner similar to the execution model on SIMT devices, such as GPUs. In this section we evaluate the performance characteristics of our engine by executing the OpenCL version of our engine on the Intel Xeon E5-2697 CPU. The OpenCL framework tries to allocate all the available processing resources offered by the selected device during the task execution. For this reason, we conduct all the experiments described in this section imposing restrictions to the cores available to the framework. Specifically, we measure the performance of our engine using 6, 12, 18 and all 24 cores of the Xeon processor. The purpose of these configurations is to demonstrate the sustained performance of our engine when portions of the CPU capacity need to be reserved for other tasks running on the CPU.

We begin the evaluation by measuring the performance sustained by the *string matching* module, using the four random ASCII pattern sets found in the **Synthetic ASCII** workload. We conduct the same experiment four times, each time allowing the engine to execute on more CPU cores, as described in the beginning of this section. The sustained throughput of the *string matching* module, for all configurations, is displayed in Figure 6.12. As we can see, the OpenCL flavor of the module, when executed using all the available cores of the Intel Xeon processor, is capable of achieving a maximum performance similar to the ones achieved when running on the NVIDIA GTX 980 GPU. However, there are some major differences. First of all, executing the OpenCL code using the entire processing capacity of the processor means that the CPU will be on maximum load on all of the 24 cores, leaving no processing power for other tasks running on the host system. Moreover, since the CPU does not have all the hardware characteristics of a dedicated SIMT device, it is not able to sustain a minimum throughput. As shown in Figure 6.12, the throughput achieved by the *string matching* module decreases as the size of the DFA increases, due to the increasing number of cache misses. Moreover, the performance gained by each extra CPU core is not fixed, since more cores compete for the same memory space.

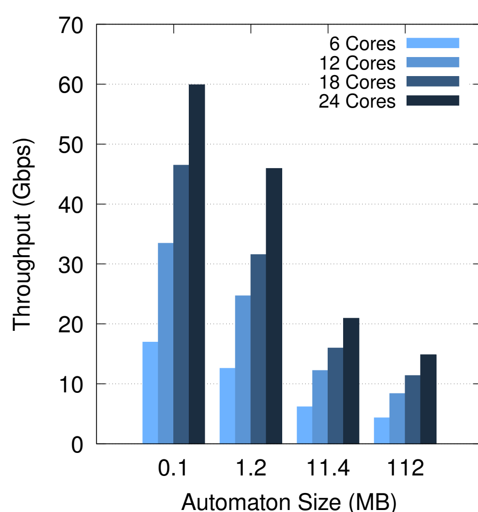
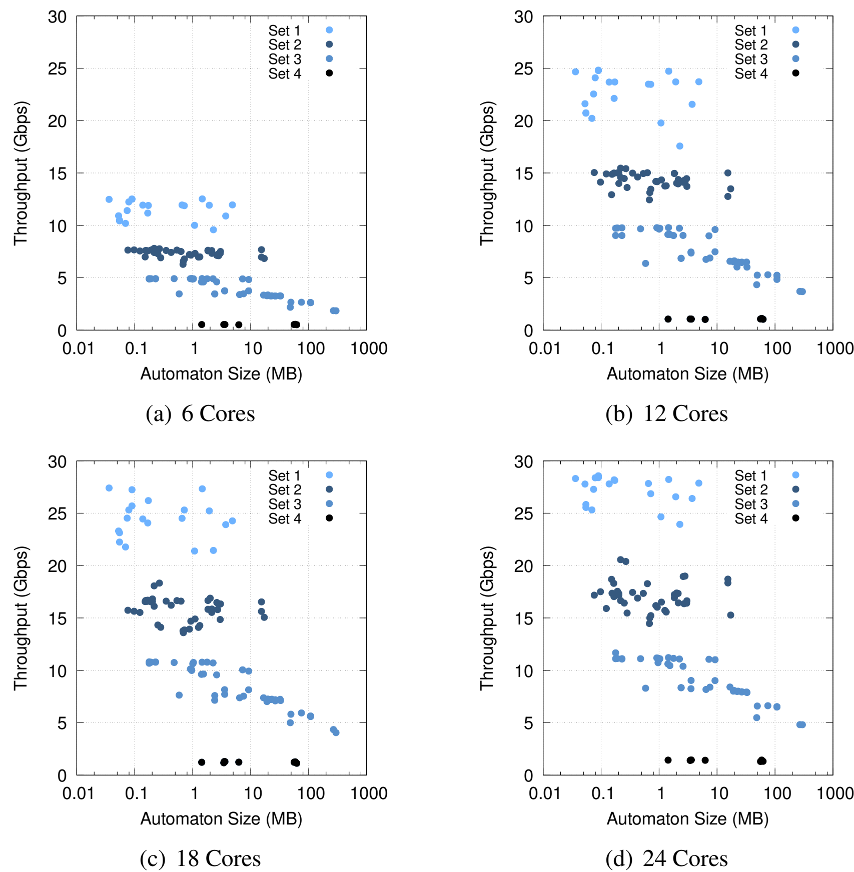


Figure 6.12: Sustained throughput achieved by the OpenCL flavor of the *string matching* module, executed on the Intel Xeon E5-2697 CPU, using the automata produced by the four pattern sets found in the **Synthetic ASCII** workload. The x-axis represents the size of the DFAs produced by combining 10, 100, 1000 and 10000 patterns respectively. Each pattern set is processed using 6, 12, 18 and 24 CPU cores. Bigger automata result to increased number of cache references, thus imposing a performance penalization. The processing throughput is increased as more cores are available to the engine. However, the performance gained by each extra CPU core is not fixed, since more cores compete for the same memory space.

In the next step of our evaluation, we measure the sustained throughput achieved by the OpenCL implementation of the *regex matching* module, executed on the Intel Xeon processor, using the automata produced by the regular expressions described in section § 6.2.2. Figure 6.13 displays the results of this experiment using 6, 12, 18 and all 24 cores of the Intel CPU. We notice that increasing the processing capacity from 1/4 to 3/4 of its maximum size, doubles the throughput. However, increasing the number of cores available to the engine from 18 to 24 does not significantly improve the performance. Moreover, even though the engine is capable of achieving throughput up to 30Gbits/s, it is not able to outperform the OpenCL and CUDA flavors executed on the Graphics Processing Unit.



(a) 6 Cores (b) 12 Cores

(c) 18 Cores (d) 24 Cores

Figure 6.13: Sustained throughput achieved by the OpenCL flavor of the *regex matching* module, executed on the Intel Xeon E5-2697 CPU, using one hundred DFAs produced by the regular expressions of the **Huge Regular Expressions** workload. Each regular expression is processed using 6, 12, 18 and 24 CPU cores. The x-axis in all plots represents the size of the various regular expression DFAs. The processing throughput is increased as more cores are available to the engine. We notice that the performance gained by each extra CPU core is not fixed, since more cores compete for the same memory space.

We conclude the evaluation of our system by measuring the performance of the *regex matching* module, using the automata produced by combining together 25 and 50 random regular expressions found in the **Huge Regular Expressions** workload. The experiment is repeated four times using the different core configurations described previously. As we can see in Figure 6.14, the OpenCL flavor of the module is able to achieve regular expression matching throughput in the order of almost 1.5Gbits/s, when all available cores are utilized. We notice that the performance gained by the OpenCL version of the module executed on the CPU is by far greater than the baseline, displayed in Figure 6.5(b). However, the engine executed on the CPU is not able to outperform the execution on the NVIDIA GTX 980 GPU, displayed in Figure 6.11.

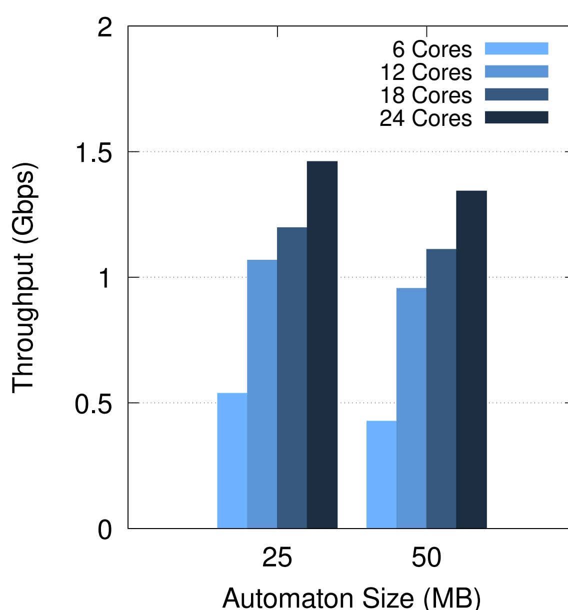


Figure 6.14: Sustained throughput achieved by the OpenCL flavor of the *regex matching* module, executed on the Intel Xeon E5-2697 CPU, using combinations of 25 and 50 random regular expressions found in the **Huge Regular Expressions** workload. Each pattern set is processed using 6, 12, 18 and 24 CPU cores. The x-axis indicates the number of regular expressions combined in a singe DFA. The processing throughput is increased as more cores are available to the engine. We notice that the performance gained by each extra CPU core is not fixed, since more cores compete for the same memory space.

Chapter 7

Related Work

Pattern matching algorithms are highly used in various applications, not only in computer science (e.g. intrusion detection, spam filtering, digital forensics, L7 traffic classification, named data networking, text mining) but also in other relevant fields, such as computational biology, chemistry and nanotechnology. A wide portion of the research community has focused over the years on either developing new efficient algorithms or even improving the performance of already existing ones. A significant amount of information regarding the evolution of the string searching field can be found in surveys, such as [48, 49].

Researchers tend to tailor and adjust the traditional string pattern searching algorithms (algorithms like those that are already discussed in § 2) to the current trends, such as parallel programming. What is more, string searching is a field that is extensively studied over the years, mostly due to the diverse areas that it can be applied as core operation. The following sections contain only some of the works that are based on string searching, since enumerating every single work in the field is not feasible. Firstly, we discuss about the various applications where string matching can be applied (such as digital forensics). In parallel, we present the related work on each specific topic, and when reasonable, we compare our work against the presented one.

7.1 Pattern matching applications

The core of intrusion detection systems lies on signature matching. Currently, researchers have assisted in favor of improving the IDS performance, by developing efficient software approaches like [50, 51, 52] in terms of memory, throughput, latency and other metrics. In addition, works like `Gnort` and `MIDeA`, focus on improving the performance of intrusion detection systems by exploiting the advantages of accelerators. Specifically, they use GPUs appropriately in order to handle the variable network traffic and to parallelize the workloads [53, 13, 14]. Moreover, `GASSP` is a GPU-accelerated stateful packet processing framework that can be used for building network packet processing applications, such as network intrusion detection systems [54, 55]. These works take advantage of the processing capabilities of NVIDIA graphics processors, unlike our framework that also targets multi-core CPUs and other many-core accelerators. What is more, our framework

can be used for a diverse set of applications, in comparison to the works discussed above, that are mainly tailored to network packet processing applications, such as intrusion detection systems.

`Hyperscan` is a high-performance multiple regular expression matching library, developed by Intel's open source project [15, 16]. Unlike this work, we mainly target accelerators using the CUDA and OpenCL frameworks, or multi-core CPUs using the OpenCL implementation.

In the field of digital forensics, there is a wide variety of research activity. String matching in digital forensics, for instance, is used in order to identify similarities among digital artifacts. Several approaches have been used due to their adequate performance and various tools are widely utilized for reasons of digital forensic investigations. For example, the `sdhash` tool [56, 57] produces a variable-length similarity digest of files, based on statistically-identified features, packed into Bloom filters [58, 59]. The `saHash` tool is used as a similarity hashing approach that operates in linear time [60].

String pattern matching is also being used in order to filter spam e-mails. However, exact string matching is not enough, since malicious users distributing spam e-mails have found many ways in order to bypass these filters. For instance, they use methods that include character level substitutions, repetitions, and insertions, for obfuscation of words that normally would be blacklisted [61, 62]. Moreover, approximate string pattern matching is used for text classification in favor of SMS spam filtering [63, 64], real-time filtering systems for detecting spam messages on social networks [65] or compromised accounts [66]. These works offer dedicated solutions, tailored for specific applications, in comparison to our framework that can be used for heterogeneous operations.

In addition, in [67], the authors implement a memory-efficient algorithm –called "leaf-attaching"– tailored for large-scale string pattern matching on ASIC and FPGA for network intrusion detection systems. This procedure not only contributes to high performance, but also targets the optimization of NIDSs in terms of memory efficiency. This algorithm scales well to support larger dictionaries.

Another interesting work studies the name lookup issue for fast packet forwarding using the longest prefix matching technique. Also, they exploit the processing power of accelerators, in order to build a GPU-based name lookup engine for high throughput and large-scale name tables [68].

In addition, techniques have been proposed in order to search multiple key patterns over encrypted data outsourced to the cloud, without increasing the search complexity, by exploiting the locality-sensitive hashing technique [69]. This kind of application is very specific and enables efficient pattern matching over encrypted data. In our work, we have not address this kind of applications, however, with minimum extensions we can expand our functionality in order to be able to include it.

Other works propose solutions that expand the already existing functionality and algorithms, in order to outperform the current techniques. A great example is the Cuckoo Filter [70] that demonstrates better performance than the traditional and widely used Bloom Filter [70].

Furthermore, pattern matching is the core operation in image recognition and image understanding systems. This kind of systems can be applied in many applications that de-

mand high speed, flexibility, error tolerance, low complexity and low cost. Thus, the field of nanotechnology can significantly benefit from those systems, since it strictly requires high speed, small footprint and low-power consumption. Specifically, in [71] and [72], the authors use approximate string matching in order to understand real-world sensor errors.

Finally, string pattern matching is a significant factor for string analysis and comparison, which are widely used in the analysis of biological sequences. DNA and protein sequences are often handled as long texts with a specific alphabet (e.g. the alphabet of RNA sequences is the {A, C, G, U}). Searching for specific sequences that are included in these big amounts of text is an elemental operation. Many works provide this functionality and focus on optimizing it further. In [73], the authors present efficient algorithms for exact and approximate matching between two RNA sequences and a method to optimally align a given RNA sequence with an unknown secondary structure to another, with known sequence and structure. The [74] introduces a structural suffix array and structural longest common prefix array for the purposes of addressing key pattern matching problems in RNA secondary structures, using the notion of structural strings. In [75] the authors present some applications of affix trees to exact and approximate pattern matching and thus, to the discovery in RNA sequences. Specifically, by allowing bidirectional search for symmetric patterns in sequences, affix trees permit to discover and locate in the sequences patterns, describing not only sequence regions, but also containing information about the secondary structure that a given region could form, something that offers significant optimizations.

7.2 String searching tools

There is a variety of state-of-the-art string searching tools. These tools are mainly based on a single or even multiple algorithms taking advantage of their diverse nature, in order to be applied in a more optimal manner. `grep` [76] and other sibling tools (e.g. `egrep`, `fgrep` and `pgrep`) are the most commonly used string searching tools in UNIX and UNIX-like systems. Their core operations vary from version to version and that is the reason that each different variant normally corresponds to a sub-category of applications. The `GNU grep` searches input files for lines containing a match to a given pattern list. When it finds a match in a line, it copies the line to standard output (by default), or produces whatever other sort of output the user requested via the options. Though `grep` expects to do the matching on text, it has no limits on input line length other than available memory, and it can match arbitrary characters within a line. If the final byte of an input file is not a newline, `grep` silently supplies one. Since newline is also a separator for the list of patterns, there is no way to match newline characters in a text [77]. The `egrep` tool scans a specified file line by line, returning the lines that contain a pattern that match a given regular expression [78]. The `fgrep` tool searches for simple strings in a file (or files) [79]. `pgrep` is used to look up processes, based on name and other attributes [80]. Other tools, such as `PowerGREP` [81], are offered with similar functionalities for Windows operating systems.

Furthermore, there is a number of tools and libraries tailored for approximate pat-

tern matching. agrep is a widely distributed tool [10, 26, 5, 82] and TRE is a POSIX compliant regex matching library, with features that include but are not limited to approximate matching [83]. Regular expression tools, such as testers and debuggers, are also widely available online [84, 85, 86, 87]. SMART is a string matching algorithms research tool [88]. Although the tools and techniques discussed above can be applied to most applications, when used in demanding environments, such as high performance network intrusion detection systems, the performance is significantly deficient, in comparison to our framework.

Moreover, pattern searching is extensively used in computational biology and chemistry. Known sequence analysis tools and search engines are BLAST [89] –a basic local alignment search tool– and FASTA [17] –a protein similarity search tool. Finally, StemSearch is a RNA search tool based on stem identification and indexing [90].

Chapter 8

Conclusions and Future Work

In this section we present a summary of the contributions of this work (§ 8.1) and some thoughts on future work (§ 8.2).

8.1 Summary of Contributions

In this work, we develop a string pattern matching framework that performs simultaneous matching of multiple fixed strings and binary patterns against multiple input data streams with a single pass over the input bytes. More specifically, the engine is able to achieve simultaneous matching of multiple POSIX Extended Regular Expressions against multiple input data streams. In addition, our framework can be executed on the vast majority of data parallel platforms, such as OpenCL enabled CPUs as well AMD and NVIDIA integrated and discrete GPUs. Furthermore, we extend the syntax of common string matching in order to support readable binary signatures and their combination with clear-text. Moreover, we offer a syntax for matching rules that allows events to be triggered when a match is achieved.

8.2 Future Work

As future work, we plan on exploring new techniques in order to generate more compact DFAs. In this way, the memory footprint of the DFAs would decrease and our approach would become more memory efficient. In addition, as shown in the evaluation (Chapter 6), the size and the creation time of a single automaton depends on the nature of the regular expressions being preprocessed. We want to study further approaches for faster DFA compilation. Also, identifying cases of catastrophic backtracking is very important. By discovering such cases, we will be able to notify the user or even exclude the fatal regular expressions from the pattern set. Finally, we plan on utilizing multiple GPUs –or even a combination of CPUs-GPUs– instead of a single one. Scaling our system this way, would offer further performance gain.

8.3 Conclusion

Having numerous applications in diverse areas, string pattern matching is a very popular field of study. Many approaches have been proposed over the years in order to provide the optimal results for each application. To cope with the current trends on parallel programming, we propose a string matching framework, provided either as a C- or Java-like API, able to be executed on commodity parallel hardware architectures, such as GPGPUs. Our framework is capable of simple string matching along with regular expressions. In addition, we achieve the simultaneous matching of multiple simple strings and binary patterns against multiple data streams as input. The framework, also, manages to simultaneously match large sets of regular expressions against multiple data streams.

The evaluation of our system shows significant performance capabilities. The sustained throughput of our massively parallel engine can achieve x15 more throughput, while processing regular expressions, and x21 more throughput, while processing simple strings – in comparison to the single-threaded CPU implementations of the algorithms used by the engine.

Bibliography

[1] A. Klapuri, "Pattern induction and matching in music signals," in *Exploring Music Contents*. Springer, 2010, pp. 188–204.

[2] A. V. Aho and M. J. Corasick, "Efficient string matching: an aid to bibliographic search," *Communications of the ACM*, vol. 18, no. 6, pp. 333–340, 1975.

[3] B. Commentz-Walter, *A string matching algorithm fast on the average*. Springer, 1979.

[4] R. M. Karp and M. O. Rabin, "Efficient randomized pattern-matching algorithms," *IBM Journal of Research and Development*, vol. 31, no. 2, pp. 249–260, 1987.

[5] S. Wu, U. Manber *et al.*, "A fast algorithm for multi-pattern searching," 1994.

[6] S. Wu, U. Manber, and E. Myers, "A subquadratic algorithm for approximate regular expression matching," *Journal of algorithms*, vol. 19, no. 3, pp. 346–360, 1995.

[7] R. Baeza-Yates and G. Navarro, "New and faster filters for multiple approximate string matching," *Random Structures & Algorithms*, vol. 20, no. 1, pp. 23–49, 2002.

[8] D. E. Knuth, J. H. Morris, Jr, and V. R. Pratt, "Fast pattern matching in strings," *SIAM journal on computing*, vol. 6, no. 2, pp. 323–350, 1977.

[9] R. S. Boyer and J. S. Moore, "A fast string searching algorithm," *Communications of the ACM*, vol. 20, no. 10, pp. 762–772, 1977.

[10] S. Wu and U. Manber, "Fast text searching: allowing errors," *Communications of the ACM*, vol. 35, no. 10, pp. 83–91, 1992.

[11] E. W. Myers, "A sublinear algorithm for approximate keyword searching," *Algorithmica*, vol. 12, no. 4-5, pp. 345–374, 1994.

[12] G. Myers, "A fast bit-vector algorithm for approximate string matching based on dynamic programming," *Journal of the ACM (JACM)*, vol. 46, no. 3, pp. 395–415, 1999.

[13] G. Vasiliadis, M. Polychronakis, S. Antonatos, E. P. Markatos, and S. Ioannidis, "Regular expression matching on graphics hardware for intrusion detection," in *Recent Advances in Intrusion Detection*. Springer, 2009, pp. 265–283.

[14] G. Vasiliadis, M. Polychronakis, and S. Ioannidis, “Parallelization and characterization of pattern matching using gpus,” in *Workload Characterization (IISWC), 2011 IEEE International Symposium on*. IEEE, 2011, pp. 216–225.

[15] “Hyperscan: a high-performance multiple regex matching library,” [Accessed: 31-Jan-2017]. [Online]. Available: https://01.org/hyperscan

[16] “Hyperscan on github,” [Accessed: 31-Jan-2017]. [Online]. Available: https://github.com/01org/hyperscan

[17] “Protein similarity search,” [Accessed: 31-Jan-2017]. [Online]. Available: http://www.ebi.ac.uk/Tools/sss/fasta/

[18] “NECOMA: Nippon-european cyberdefense-oriented multilayer threat analysis,” [Accessed: 31-Jan-2017]. [Online]. Available: http://www.necoma-project.eu/

[19] “RAPID: Heterogeneous secure multi-level remote acceleration service for low-power integrated systems and devices,” [Accessed: 31-Jan-2017]. [Online]. Available: http://www.rapid-project.eu/

[20] “SHARCS: Secure hardware-software architectures for robust computing systems,” [Accessed: 31-Jan-2017]. [Online]. Available: http://http://www.sharcs-project.eu

[21] “Delivarable D3.5: Countermeasure Application - Results,” NECOMA Project, 2015, [Accessed: 19-Jan-2017]. [Online]. Available: http://www.necoma-project.eu/m/filer_public/70/fd/70fd1052-cf95-4b58-bc47-61a75be64d19/necoma-d35r2577.pdf

[22] “Delivarable D2.4: Antivirus Ported on RAPID,” RAPID Project, 2016, [Accessed: 19-Jan-2017]. [Online]. Available: http://rapid-project.eu/_docs/RAPID_D2.4.pdf

[23] R. N. Horspool, “Practical fast searching in strings,” *Software: Practice and Experience*, vol. 10, no. 6, pp. 501–506, 1980.

[24] R. A. Baezayates and G. H. Gonnet, “Fast string matching with mismatches,” *Information and Computation*, vol. 108, no. 2, pp. 187–199, 1994.

[25] V. I. Levenshtein, “Binary codes capable of correcting deletions, insertions, and reversals,” in *Soviet physics doklady*, vol. 10, no. 8, 1966, pp. 707–710.

[26] S. Wu and U. Manber, “Agrep–a fast approximate pattern-matching tool,” *Usenix Winter 1992*, pp. 153–162, 1992.

[27] R. W. Hamming, “Error detecting and error correcting codes,” *Bell System technical journal*, vol. 29, no. 2, pp. 147–160, 1950.

[28] “What Regular Expressions Are Exactly - Terminology,” [Accessed: 19-Jan-2017]. [Online]. Available: http://www.regular-expressions.info/tutorial.html

[29] Wikipedia, "Stephen Cole Kleene — Wikipedia, The Free Encyclopedia," 2017, [Accessed: 19-Jan-2017]. [Online]. Available: https://en.wikipedia.org/wiki/Stephen_Cole_Kleene

[30] "sed, a stream editor," [Accessed: 19-Jan-2017]. [Online]. Available: https://www.gnu.org/software/sed/manual/sed.html

[31] "The GNU Awk User's Guide," [Accessed: 19-Jan-2017]. [Online]. Available: https://www.gnu.org/software/gawk/manual/gawk.html

[32] "Snort - Network Intrusion Detection & Prevention System," [Accessed: 19-Jan-2017]. [Online]. Available: https://www.snort.org/

[33] "ClamAV," [Accessed: 19-Jan-2017]. [Online]. Available: https://www.clamav.net/

[34] "ed, A line-oriented text editor," [Accessed: 19-Jan-2017]. [Online]. Available: https://www.gnu.org/software/ed/

[35] "Unix, — An Open Group Standard," [Accessed: 19-Jan-2017]. [Online]. Available: http://www.unix.org/

[36] Wikipedia, "Grep — Wikipedia, The Free Encyclopedia," 2017, [Accessed: 19-Jan-2017]. [Online]. Available: https://en.wikipedia.org/wiki/Grep

[37] "POSIX — IEEE Standards Association," [Accessed: 19-Jan-2017]. [Online]. Available: https://standards.ieee.org/develop/wg/POSIX.html

[38] "Basic Regular Expressions," [Accessed: 19-Jan-2017]. [Online]. Available: http://pubs.opengroup.org/onlinepubs/009695399/basedefs/xbd_chap09.html#tag_09_03

[39] "Extended Regular Expressions," [Accessed: 19-Jan-2017]. [Online]. Available: http://pubs.opengroup.org/onlinepubs/009696899/basedefs/xbd_chap09.html#tag_09_04

[40] "The Single UNIX ® Specification, Version 2 — Regular Expressions," [Accessed: 19-Jan-2017]. [Online]. Available: http://pubs.opengroup.org/onlinepubs/7908799/xbd/re.html

[41] M. J. Flynn, "Some computer organizations and their effectiveness," *IEEE transactions on computers*, vol. 100, no. 9, pp. 948–960, 1972.

[42] "NVIDIA's Next Generation CUDA™ Compute Architecture: Fermi™," [Accessed: 24-Jan-2017]. [Online]. Available: http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf

[43] L. Koromilas, G. Vasiliadis, I. Manousakis, and S. Ioannidis, "Efficient software packet processing on heterogeneous and asymmetric hardware architectures," in *Proceedings of the tenth ACM/IEEE symposium on Architectures for networking and communications systems*. ACM, 2014, pp. 207–218.

[44] E. Papadogiannaki, L. Koromilas, G. Vasiliadis, and S. Ioannidis, "Efficient software packet processing on heterogeneous and asymmetric hardware architectures," *IEEE/ACM Transactions on Networking*, 2017.

[45] "Intel xeon phi coprocessors," [Accessed: 24-Jan-2017]. [Online]. Available: http://www.intel.com/content/www/us/en/processors/xeon/xeon-phi-coprocessor-overview.html

[46] "Cuda parallel computing platform," [Accessed: 24-Jan-2017]. [Online]. Available: http://www.nvidia.com/object/cuda_home_new.html

[47] "The opencl framework," [Accessed: 24-Jan-2017]. [Online]. Available: http://www.khronos.org/opencl/

[48] G. Navarro, "A guided tour to approximate string matching," *ACM computing surveys (CSUR)*, vol. 33, no. 1, pp. 31–88, 2001.

[49] V. Saikrishna, A. Rasool, and N. Khare, "String matching and its applications in diversified fields," *International Journal of Computer Science Issues*, vol. 9, no. 1, pp. 219–226, 2012.

[50] S. Vakili, J. P. Langlois, B. Boughzala, and Y. Savaria, "Memory-efficient string matching for intrusion detection systems using a high-precision pattern grouping algorithm," in *Proceedings of the 2016 Symposium on Architectures for Networking and Communications Systems*, ser. ANCS '16. New York, NY, USA: ACM, 2016, pp. 37–42. [Online]. Available: http://doi.acm.org/10.1145/2881025.2881031

[51] "Dfc: Accelerating string pattern matching for network applications," in *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*. Santa Clara, CA: USENIX Association, Mar. 2016. [Online]. Available: https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/choi

[52] A. Tumeo, O. Villa, and D. G. Chavarría-Miranda, "Aho-corasick string matching on shared and distributed-memory parallel architectures," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 23, no. 3, pp. 436–443, 2012.

[53] G. Vasiliadis, S. Antonatos, M. Polychronakis, E. P. Markatos, and S. Ioannidis, "Gnort: High performance network intrusion detection using graphics processors," in *Recent Advances in Intrusion Detection*. Springer, 2008, pp. 116–134.

[54] G. Vasiliadis, L. Koromilas, M. Polychronakis, and S. Ioannidis, "Gaspp: A gpu-accelerated stateful packet processing framework." in *USENIX Annual Technical Conference*, 2014, pp. 321–332.

[55] ——, "Design and implementation of a stateful network packet processing framework for gpus," *IEEE/ACM Transactions on Networking*, vol. PP, no. 99, pp. 1–14, 2016.

[56] "sdhash tool," http://roussev.net/sdhash/sdhash.html.

[57] "sdhash on GitHub," https://github.com/sdhash/sdhash.

[58] V. Roussev, "Data fingerprinting with similarity digests," in *Advances in digital forensics vi*. Springer, 2010, pp. 207–226.

[59] ——, "An evaluation of forensic similarity hashes," *digital investigation*, vol. 8, pp. S34–S41, 2011.

[60] F. Breitinger and H. Baier, "Similarity preserving hashing: Eligible properties and a new algorithm mrsh-v2," in *Digital forensics and cyber crime*. Springer, 2012, pp. 167–182.

[61] D. Sculley, G. Wachman, and C. E. Brodley, "Spam filtering using inexact string matching in explicit feature space with on-line linear classifiers." in *TREC*, 2006.

[62] R. Verma and J. Dhar, "Online spam filter for duplicate or near duplicate message content detection scheme," *Journal of Convergence Information Technology*, vol. 9, no. 4, p. 23, 2014.

[63] S. J. Delany, M. Buckley, and D. Greene, "Sms spam filtering: methods and data," *Expert Systems with Applications*, vol. 39, no. 10, pp. 9899–9908, 2012.

[64] T. Almeida, J. M. G. Hidalgo, and T. P. Silva, "Towards sms spam filtering: Results under a new dataset," *International Journal of Information Security Science*, vol. 2, no. 1, pp. 1–18, 2013.

[65] H. Gao, Y. Chen, K. Lee, D. Palsetia, and A. N. Choudhary, "Towards online spam filtering in social networks." in *NDSS*, 2012.

[66] M. Egele, G. Stringhini, C. Kruegel, and G. Vigna, "Compa: Detecting compromised accounts on social networks." in *NDSS*, 2013.

[67] H. Le and V. K. Prasanna, "A memory-efficient and modular approach for large-scale string pattern matching," *Computers, IEEE Transactions on*, vol. 62, no. 5, pp. 844–857, 2013.

[68] Y. Wang, Y. Zu, T. Zhang, K. Peng, Q. Dong, B. Liu, W. Meng, H. Dai, X. Tian, Z. Xu *et al.*, "Wire speed name lookup: A gpu-based approach," in *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, 2013, pp. 199–212.

[69] B. Wang, S. Yu, W. Lou, and Y. T. Hou, "Privacy-preserving multi-keyword fuzzy search over encrypted data in the cloud," in *INFOCOM, 2014 Proceedings IEEE*. IEEE, 2014, pp. 2112–2120.

[70] V. Gupta and F. Breitinger, "How cuckoo filter can improve existing approximate matching techniques," in *Digital Forensics and Cyber Crime*. Springer, 2015, pp. 39–52.

[71] V. Annampedu and M. D. Wagh, "Reconfigurable approximate pattern matching architectures for nanotechnology," *Microelectronics Journal*, vol. 38, no. 3, pp. 430–438, 2007.

[72] ——, "Approximate pattern matching in nanotechnology," *Proc. of Nanotech 2006*, vol. 3, pp. 316–319, 2006.

[73] V. Bafna, S. Muthukrishnan, and R. Ravi, "Computing similarity between rna strings," in *Combinatorial Pattern Matching*. Springer, 1995, pp. 1–16.

[74] R. Beal and D. Adjeroh, "Efficient pattern matching for rna secondary structures," *Theoretical Computer Science*, vol. 592, pp. 59–71, 2015.

[75] G. Mauri and G. Pavesi, "Algorithms for pattern matching and discovery in rna secondary structure," *Theoretical Computer Science*, vol. 335, no. 1, pp. 29–51, 2005.

[76] *grep - Linux man page*, [Accessed: 31-Jan-2017]. [Online]. Available: https://linux.die.net/man/1/grep

[77] "The gnu grep 3.0 manual," [Accessed: 19-Jan-2017]. [Online]. Available: https://www.gnu.org/software/grep/manual/grep.html

[78] *egrep - Linux man page*, [Accessed: 31-Jan-2017]. [Online]. Available: https://linux.die.net/man/1/egrep

[79] *fgrep - Linux man page*, [Accessed: 31-Jan-2017]. [Online]. Available: https://linux.die.net/man/1/fgrep

[80] *pgrep - Linux man page*, [Accessed: 31-Jan-2017]. [Online]. Available: https://linux.die.net/man/1/pgrep

[81] "Powergrep," [Accessed: 31-Jan-2017]. [Online]. Available: http://www.powergrep.com/

[82] "agrep GitHub," https://github.com/Wikinaut/agrep.

[83] "TRE regexp matching library," http://laurikari.net/tre/.

[84] "Regular expressions 101," [Accessed: 31-Jan-2017]. [Online]. Available: https://regex101.com/

[85] "Regexr," [Accessed: 31-Jan-2017]. [Online]. Available: http://regexr.com/

[86] "Online regex tester," [Accessed: 31-Jan-2017]. [Online]. Available: http://www.regextester.com/

[87] "Debuggexbeta," [Accessed: 31-Jan-2017]. [Online]. Available: https://www.debuggex.com

[88] "Smart: String matching algorithms research tools," [Accessed: 31-Jan-2017]. [Online]. Available: https://www.dmi.unict.it/~faro/smart/index.php

[89] "Blast: Basic local alignment search tool," [Accessed: 31-Jan-2017]. [Online]. Available: https://blast.ncbi.nlm.nih.gov/Blast.cgi

[90] N. Milo, S. Yogev, and M. Ziv-Ukelson, "Stemsearch: Rna search tool based on stem identification and indexing," *Methods*, vol. 69, no. 3, pp. 326–334, 2014.