High Performance Pattern Matching and Data Remanence on Graphics Processing Units

Xavier J. A. Bellekens

A thesis submitted for the degree of Doctor of Philosophy to the
Centre for Intelligent and Dynamic Communications,
University of Strathclyde.

April 2016

This thesis is the result of the author's original research. It has been composed by the author and has not been previously submitted for examination which has led to the award of a degree.

Signed:

Date:

‘Power is in tearing human minds to pieces and putting them together again in new shapes of your own choosing.’

—George Orwell,
“1984”, 1984.

Abstract

Pattern matching is an important task in a plethora of different fields ranging from computer science to medical application, but is also a resource consuming problem. With the increase in network link speed, and the tremendous amounts of data generated, serial pattern matching on Central Processing Unit (CPU) is close to being rendered obsolete. The ubiquitous Graphics Processing Unit (GPU) have become the focus of much interest within the scientific community due to their highly parallel computing capabilities, and cost effectiveness offered by the hardware.

This thesis presents an empirical investigation of massively parallel single and multi pattern matching algorithms, as well as security and privacy concerns for data processing on GPUs. This thesis demonstrates a trie reduction algorithm that reduces the size of the data stored in GPU memory. GPUs have a limited amount of memory and with the increasing number of patterns to be searched for, space complexity is of great importance. This work addresses these challenges and investigates different memory hierarchies for different matching problems increasing the overall performance.

This work also presents a Digital Forensic (DF) and Reverse Engineering (RE) methodology based on an evaluation of the different memory hierarchies present in a GPU.

The results of the investigation presented in this thesis show that single and multi-pattern matching algorithms can benefit of the massively parallel capabilities of GPUs when implemented with hardware design in mind. Furthermore, it is demonstrated that data offloaded to the GPU is subject to data leaks.

Acknowledgements

First and foremost I would like to thank my supervisor Dr. Robert Atkinson, for giving me the opportunity to undertake this work and for his advice and guidance when I needed them the most. Secondly I would like to thank Dr. Christos Tachtatzis for the lengthy discussions, the numerous advices, and his patience during those 4 years.

I would also like to thank everybody at Agilent Technologies and Keysight Technologies for the opportunity to undertake this research, their support and advices.

I would also like to thank my internal and external examiners, Dr. Louise Crockett and Dr. Allan Tomlinson for their valuable suggestions and advice which further helped enhance my thesis.

I am also grateful for my friends, for their unconditional friendship, and believing I could achieve this much. Thank you — In alphabetical order — Iwan Briquemont, Grégoire Del Lisse, Quentin Franssen, Dr. Andrew Hamilton, Dr. Jakub Konka, Dominique Labrut, Benjamin Lefebvre, Kamila Nieradzinska, Dr. Amar Seeam, Nadezhda Shivarova, Dr. Neil Sinclair, and Olivier Vencencius.

Finally I would like to thank my parents, for their support throughout my studies, my sister for all the laughter and advices, and my grand-dad who came to visit me in Scotland despite being a centenarian.

Contents

List of Publications

[1] X. Bellekens, C. Tachtatzis, A. Hamilton, P.-L. Dubouilh, K. Nieradzinska, and R. C. Atkinson, "A node expansion allegory for gpu accelerated pattern matching," in *IEEE Transactions on Parallel and Distributed Systems*, IEEE, April 2016 (Under Submission).

[2] X. Bellekens, G. Paul, J. M. Irvine, C. Tachtatzis, and R. C. Atkinson, "Strategies for protecting intellectual property when using graphics processing units CUDA applications," in *Proceedings of the 9th International Conference on Security of Information and Networks*, SIN '16, (New York, NY, USA), ACM, 2016.

[3] X. Bellekens, G. Paul, J. M. Irvine, C. Tachtatzis, R. C. Atkinson, T. Kirkham, and C. Renfrew, "Data remanence and digital forensic investigation for CUDA graphics processing units," in *Integrated Network Management (IM), 2015 IFIP/IEEE International Symposium on*, pp. 1345–1350, May 2015.

[4] X. J. A. Bellekens, C. Tachtatzis, R. C. Atkinson, C. Renfrew, and T. Kirkham, "A highly-efficient memory-compression scheme for GPU-accelerated intrusion detection systems," in *Proceedings of the 7th International Conference on Security of Information and Networks*, SIN '14, (New York, NY, USA), pp. 302:302–302:309, ACM, 2014.

[5] X. J. A. Bellekens, C. Tachtatzis, R. C. Atkinson, C. Renfrew, and T. Kirkham, "Glop: Enabling massively parallel incident response through GPU log processing," in *Proceedings of the 7th International Conference on Security of Information and Networks*, SIN '14, (New York, NY, USA), pp. 295:295–295:301, ACM, 2014.

[6] X. Bellekens, R. Atkinson, C. Renfrew, and T. Kirkam, "Investigation of GPU-based pattern matching," in *14th Annual Post Graduate Symposium on the Convergence of Telecommunications, Networking and Broadcasting*, (Liverpool, UK), PGNET13, June 2013.

Acronyms

AC Aho-Corasick

ALU Arithmetic-Logic Unit

API Application Program Interface

ASCII American Standard Code for Information Interchange

BMH Boyer-Moore-Horspool

CPU Central Processing Unit

CUDA Compute Unified Device Architecture

DAFSA Deterministic Finite State Automaton

DAWG Directed Acyclic Word Graph

DFA Deterministic Finite Automaton

DF Digital Forensic

DMA Direct Memory Access

DNA Deoxyribonucleic Acid

DRAM Dynamic Random Access Memory

DOS Denial of Service

DPI Deep Packet Inspection

DRAM Dynamic random-access memory

DtoH Device to Host

ECC Error-correcting code memory

ELF Executable and Linkable Format

GLoP GPU Log Processing

GPGPU General-Purpose Graphics Processing Unit

GPU Graphics Processing Unit

HEPFAC Highly Efficient Parallel Failure-less Aho-Corasick

HPC High Performance Computers

HtoD Host to Device

IDS Intrusion Detection Systems

IP Intellectual Property

JIT Just-In-Time

KMP Knuth-Morris-Pratt

LCU Logic Control Units

MIMD Multiple Instructions Multiple Data

MISD Multiple Instruction Single Data

MT Mersene Twister

OS Operating System

PCB Printed Circuit Board

PCI Peripheral Component Interconnect

PFAC Parallel Failure-less Aho-Corasick

PTX Parallel Thread Execution

RAM Random Access Memory

RE Reverse Engineering

SASSM Shader ASSeMbly

SDK Software Development Kit

SFU Special Function Units

SIMD Single Instruction Multiple Data

SIMT Single Instruction Multiple Thread

SISD Single Instruction Single Data

SM Streaming Multiprocessors

SRAM Static random-access memory

Chapter 1

Introduction

Data processing requirements and computer networks have been growing in size, speed and complexity over the last decade. According to Cisco, as many as 50 billion devices will be interconnected and generating data by 2020 [1]. These trends will also affect greatly Intrusion Detection Systems (IDS) [2], malware analysis [3] and digital forensics [4]. The increasing number of interconnected devices often have poor security and become targets of choice for numerous attacks. High throughput traffic for fast file analysis is therefore crucial. Furthermore, other fields such as medical sciences require significant processing power in order to analyse genomes in reasonable times [5]. Database engines are the core of the web, storing information generated and require fast and efficient ways to retrieve data [6]. As biological, textual, and security threats continue to grow at exponential rates, the costs involved in terms of execution time and memory resources increases accordingly, requiring faster software and hardware [7] [8]. There has been an increasing interest pattern matching in order to solve problems of different nature over the years. As such, the field of pattern matching is divided into single and multi-pattern matching algorithms. To this end the work presented in this thesis investigates parallel implementation of both single and multi-pattern algorithms as well as security and privacy concerns of offloading data to a GPU, allowing applications to increase their matching throughput.

1.1 Research Objectives

This thesis explores the feasibility of off-the-shelf massively parallel single and multi-pattern matching as well as security and privacy aspects of GPU accelerated applications. The concept of offloading pattern matching to GPU was initially proposed by Jacob *et al.* in 2006 [9]. The work presented made use the *Cg* language in order to offload the pattern matching engine to a graphics card and used the Knuth-Morris-Pratt (KMP) algorithm to increase the throughput of an IDS. With the hardware advancement in recent years, the release of new GPUs and the novel Compute Unified Device Architecture (CUDA) framework, General-Purpose Graphics Processing Unit (GPGPU) processing is becoming more common. Pattern matching and data analytics can benefit from off-the-shelf massively parallel computation. It is for this reason that this thesis explores the problem of pattern matching on GPUs as well as the security and privacy implications of offloading sensitive data for computation. More specifically, the main objectives of this theses are to:

1. Review and investigate the potential of single pattern matching algorithms on GPUs;

2. Analyse multi-pattern matching algorithms for large scale data processing using different memory hierarchies;

3. Create a memory scheme, allowing reduced memory requirements of patterns stored on the limited memory of the GPU;

4. Investigate data remanence and privacy of data stored in different memory hierarchies of the GPU as well as identify information yielded by the CUDA binaries;

5. Establish a Digital Forensic (DF) and Reverse Engineering (RE) methodology, to enable data acquisition and facilitate memory and CUDA binary analysis;

It is important to recognise that pattern matching applications require a high level of reliability, throughput and flexibility. Therefore algorithms that are scalable and approaches that take advantage of massively parallel hardware to increase data processing will be the key to the future of pattern matching and data processing in numerous fields [10] [11].

1.2 Thesis Statement

Pattern matching is a complex task requiring increasing processing power. This work investigates massively parallel hardware to increase the throughput of single and multi-pattern matching algorithms. Moreover, with the increase of cloud computing and GPU-as-a-service, an analysis of security and privacy highlighting data remanence and digital forensic challenges is provided.

1.3 Main Contributions

The work presented in this thesis complements the existing state of the art through the following contributions:

- A novel implementation of different single-pattern matching algorithms on GPU. By offloading the pattern matching operations to the GPU, the implementations takes advantage of techniques such as loop unrolling and memory hierarchies in order to increase their performances on off-the-shelf massively parallel hardware [12].

- A GPU Log Processing (GLoP) algorithm is introduced, using a variant of the Parallel Failure-less Aho-Corasick (PFAC) algorithm for high performance log monitoring. The scalability of the algorithm is analysed, along with two different memory implementations, allowing for evaluating the performance of different memory hierarchies *in situ*. Additionally, a state table reduction algorithm based on the Boyer-Moore-Horspool (BMH) bad character table is implemented [13].

- A Highly Efficient Parallel Failure-less Aho-Corasick (HEPFAC) algorithm is presented: the first version takes advantage of global memory and stores the trie using a one-dimensional breadth-first / row major ordered array and makes use of bitmapped nodes and a trie reduction algorithm. A second version of the HEPFAC algorithm is implemented using a two-dimensional node expansion allegory. The algorithms takes advantage of texture memory, increasing the throughput of the algorithm by a factor of 2.5 when applied to large alphabets. The algorithm is evaluated against different alphabet sizes and multiple optimisation techniques (memory locality, loop unrolling, coalesced memory access). Furthermore, the algorithm is evaluated on Amazon EC2 cloud allowing other researchers to compare their results on the same hardware and drive more efficient conclusions [14].

- A methodology for forensic investigation on GPUs and RE of CUDA based application is presented. The methodology describes ways to acquire remanent data stored in the different types of memories, making use of 3 types of GPUs (High End, Consumer and Mobile). Furthermore, the methodology highlights privacy and intellectual property concerns as well as data leakage in CUDA based applications, whilst providing a mean to identify and analyse their content. The methodology is derived and subsequently validated using different CUDA-based applications in order to preserve the integrity of the obtained data [15].

1.4 Thesis Outline

This thesis consists of six chapters and one appendix. Chapter 2 provides details and concepts on sequential single and multi-pattern matching algorithms, defining the core algorithms of the field. This chapter also introduces key work, such as the Deterministic Finite State Automaton (DAFSA) and the Directed Acyclic Word Graph (DAWG) algorithms, demonstrating branch reduction techniques. Furthermore it outlines the Aho-Corasick (AC) algorithm which is a direct extension of the single pattern matching algorithm KMP and is a popular multi-pattern algorithm.

Chapter 3 introduces the concepts of GPU architecture and presents parallel implementation of single and multi-pattern matching algorithms. It starts with the CUDA programming model, defines the Single Instruction Multiple Thread (SIMT) model and then moves onto a brief description of memory hierarchies and optimisations. An overview of GPU architecture fundamentals is presented to highlight GPU limitations and memory optimisations employed in parallel programming. Furthermore, prior work in the field is outlined, describing different parallel implementation of single and multi-pattern matching algorithms. A novel implementation of the KMP algorithm is then presented making use of optimisations in order to increase the processing throughput of the algorithm. A similar approach is then presented for the BMH algorithm. The chapter concludes with the presentation of an innovative modified version of the PFAC algorithm, adapted to GLoP, demonstrating possible solutions to the impediments of massively parallel single-pattern matching algorithms.

In Chapter 4 the studied problem transforms into a memory storage requirement. An overview of the state of the art is presented and a novel HEPFAC algorithm is presented. The algorithm reduces the size of a trie for efficient data transfer to and from the GPU as well as reduced data storage requirements. The algorithm takes advantage of different memory hierarchies and is evaluated against different alphabet sizes, memory requirements and optimisation. The chapter concludes with a performance analysis on public cloud architectures in order to facilitate future research.

Whilst considering the numerous algorithm applicabilities of novel single and multi-pattern matching algorithms, Chapter 5 is dedicated to security and privacy, highlighting potential problems such as data remanence and data leaks on GPUs. This chapter demonstrates data remanence in different memory hierarchies and on different GPU architectures. The work described builds upon prior research to evaluate and develop a DF and RE methodology, allowing digital forensic investigators to investigate crimes committed using massively parallel off-the-shelf hardware.

Finally Chapter 6 draws conclusions, brings together all research aspects proposed in this thesis and discusses future work.

Chapter 2

Pattern Matching

This chapter explains the concepts and importance of pattern matching. To this end, firstly the most common single pattern matching algorithms are described. Furthermore multi-pattern matching algorithms and different types of deterministic finite automatons are outlined.

2.1 Single Pattern Matching

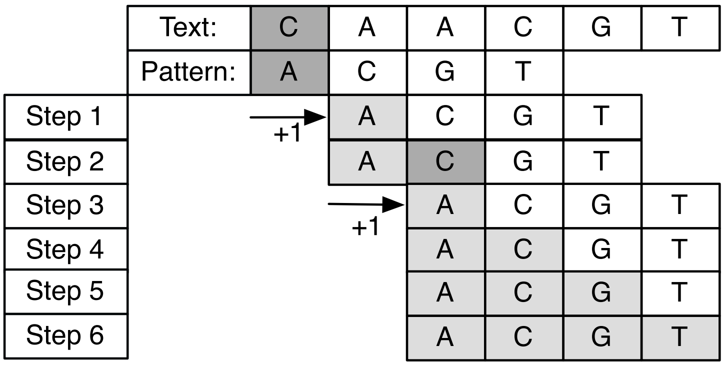
Single pattern matching plays a key role in theoretical computer science. It is widely used for text processing, intrusion detection systems, anti-virus engines, information retrieval and Deoxyribonucleic Acid (DNA) sequencing [16].

Apostolico *et al.* [17] describe the problem of single pattern matching as finding all occurrences of a pattern $P = \{p_0, p_1, p_{m-1}\}$ of length m in a text string $T = \{t_0, t_1, t_{n-1}\}$ of length n, where the patterns and the texts are expressed in an alphabet Σ. For example, DNA is represented by an alphabet of four characters $|\Sigma| = 4$, while American Standard Code for Information Interchange (ASCII) is represented by an alphabet of two-hundreds and fifty six characters $|\Sigma| = 256, \Sigma = a, b, c, ...$.

Single pattern matching algorithms can be as simple as the naive pattern matching algorithm which is described first, to lay the cornerstone of the field, or more sophisticated such as the Knuth-Morris-Pratt algorithm which is described subsequently highlighting the efficiency of shifting the pattern to speed the matching process up. Finally the Boyer-Moore-Horspool algorithm is outlined demonstrating the performance of the bad character table.

2.1.1 Naive Pattern Matching Algorithm

The naive algorithm also known as brute-force algorithm is the foundation of the pattern matching field. The idea behind the algorithm is to align the pattern P with the text T and compare each character against the text T.



	Text:	C	A	A	C	G	T
	Pattern:	A	C	G	T		
Step 1		+1	A	C	G	T	
Step 2			A	C	G	T	
Step 3			+1	A	C	G	T
Step 4				A	C	G	T
Step 5				A	C	G	T
Step 6				A	C	G	T

Figure 2.1
Naive Pattern Matching Algorithm Workflow

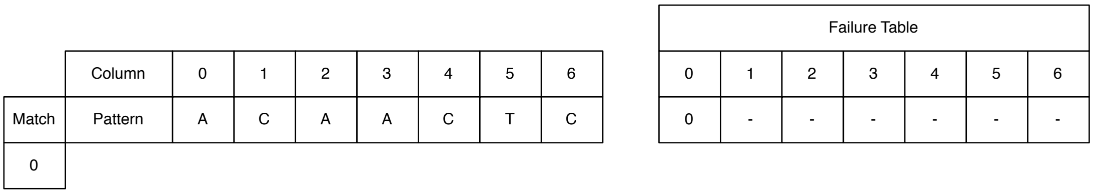
Figure 2.1 demonstrates the steps the naive algorithm requires to match the pattern P against the text T. When a mismatch occurs (Step 1), the pattern P is shifted to the right by one position and the comparison restarts from the first letter of the pattern. When a match occurs (Step 2), the next characters of the pattern $P[1]$ is compared against the next character of the text $T[2]$ until a mismatch occurs (Step 3), or until the pattern P is fully matched (Step 6).

The naive method is simple but inefficient, in particular when the pattern contains repeated characters, or when the pattern is matched against a large text. The worst case running time of the algorithm is $\mathcal{O}(nm)$ where m denotes the length of the pattern P and n denotes the length of the text T.

2.1.2 Knuth-Morris-Pratt Algorithm

The Knuth-Morris-Pratt (KMP) algorithm improves upon the brute-force algorithm, by introducing and improving it with a failure table [18] [19].

The failure table is constructed prior to any searches, in order to permit portions of the text T to be skipped during the matching process.Constructing the failure table essentially requires to match the pattern against itself. This is done by iterating through the pattern P from the start towards the end, identifying the longest prefix of the pattern [19].

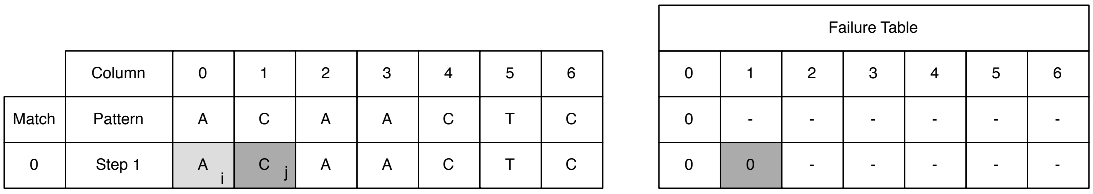


	Column	0	1	2	3	4	5	6
Match	Pattern	A	C	A	A	C	T	C
0								

Failure Table						
0	1	2	3	4	5	6
0	-	-	-	-	-	-

Figure 2.2
Knuth-Morris-Pratt Failure Table Construction

The procedure to compute the failure table is shown in Figure 2.2. Prior to any computation the cell at column 0 of the failure table is assigned to 0 (Figure 2.2, Right) and the value of *Match* is set to 0 (Figure 2.2, Left). Match identifies the longest prefix currently identified in the pattern. To compute the failure table the algorithm undertakes the following steps:

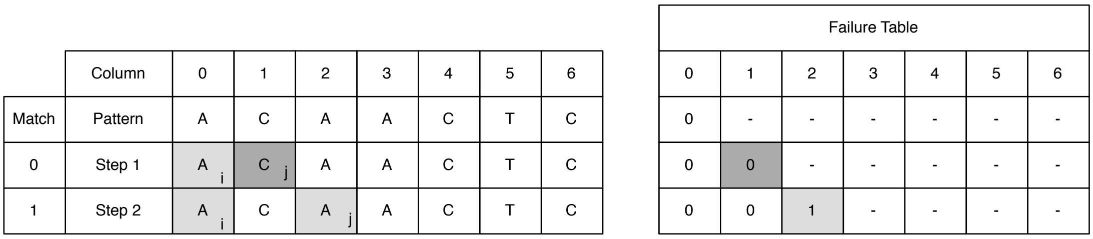


	Column	0	1	2	3	4	5	6
Match	Pattern	A	C	A	A	C	T	C
0	Step 1	A i	C j	A	A	C	T	C

Failure Table						
0	1	2	3	4	5	6
0	-	-	-	-	-	-
0	0	-	-	-	-	-

Figure 2.3
Knuth-Morris-Pratt Failure Table Construction Step 1

Step 1: Figure 2.3 shows that the two characters at column $i = 0$ and $j = 1$ are compared, since "A" is different from "C", a mismatch has occurred and the cell at column 1 of the failure table is set the to the current value of *Match*. The column pointer $i = 0$ remains unchanged while the column pointer j is incremented by 1.

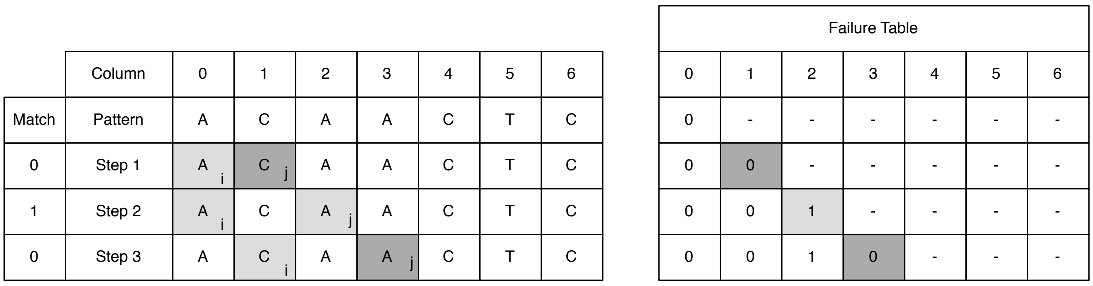


	Column	0	1	2	3	4	5	6
Match	Pattern	A	C	A	A	C	T	C
0	Step 1	A i	C j	A	A	C	T	C
1	Step 2	A i	C	A j	A	C	T	C

Failure Table						
0	1	2	3	4	5	6
0	-	-	-	-	-	-
0	0	-	-	-	-	-
0	0	1	-	-	-	-

Figure 2.4
Knuth-Morris-Pratt Failure Table Construction Step 2

Step 2: In the next step shown in Figure 2.4, since j was incremented by one,the character at column $i = 0$ and $j = 2$ are compared; since the characters are the same, a match has occurred, the value of *Match* is incremented by 1. This value corresponds to the current longest prefix matched. The cell in the failure table corresponding to the index $j = 2$ is set to the value of *Match*. Finally, the column indexes i and j are both incremented by 1.

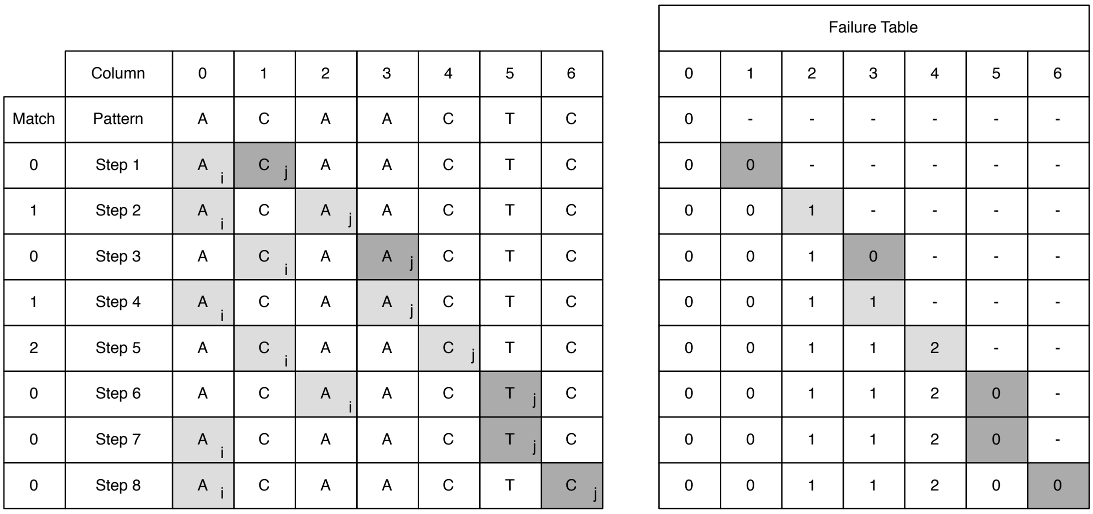


	Column	0	1	2	3	4	5	6
Match	Pattern	A	C	A	A	C	T	C
0	Step 1	A i	C j	A	A	C	T	C
1	Step 2	A i	C	A j	A	C	T	C
0	Step 3	A	C i	A	A j	C	T	C

Failure Table						
0	1	2	3	4	5	6
0	-	-	-	-	-	-
0	0	-	-	-	-	-
0	0	1	-	-	-	-
0	0	1	0	-	-	-

Figure 2.5
Knuth-Morris-Pratt Failure Table Construction Step 3

Step 3: The characters at column $i = 1$ and $j = 3$ are compared; Since “C” and “A” are different, a mismatch has occurred and the value of *Match* is reset to 0. The cell corresponding to index j in the failure table is set to the value to the current value of *Match* as shown in Figure 2.5. Furthermore, the column pointer i is set to the value of the cell at column $i - 1$ in the failure table. Hence, i is now equal to 0, while j remains unchanged as shown in Figure 2.6.



	Column	0	1	2	3	4	5	6
Match	Pattern	A	C	A	A	C	T	C
0	Step 1	A i	C j	A	A	C	T	C
1	Step 2	A i	C	A j	A	C	T	C
0	Step 3	A	C i	A	A j	C	T	C
1	Step 4	A i	C	A	A j	C	T	C
2	Step 5	A	C i	A	A	C j	T	C
0	Step 6	A	C	A i	A	C	T j	C
0	Step 7	A i	C	A	A	C	T j	C
0	Step 8	A i	C	A	A	C	T	C j

Failure Table

0	1	2	3	4	5	6
0	-	-	-	-	-	-
0	0	-	-	-	-	-
0	0	1	-	-	-	-
0	0	1	0	-	-	-
0	0	1	1	-	-	-
0	0	1	1	2	-	-
0	0	1	1	2	0	-
0	0	1	1	2	0	-
0	0	1	1	2	0	0

Figure 2.6
Final Failure Knuth-Morris-Pratt Failure Table Construction

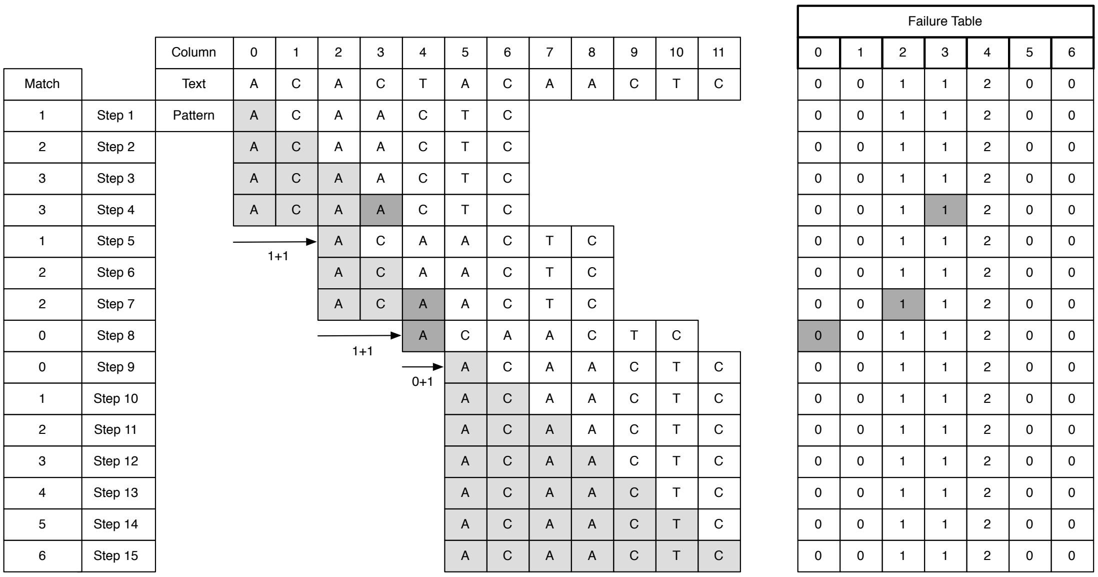
The process continues until the column pointer *j* has traversed all characters in the pattern as shown in Figure 2.6. Table 2.1 depicts the successful computed failure table for the assumed pattern. It is worth noting that the variable *Match* is either incremented by 1 whenever a prefix is identified, or, otherwise, reset to 0.

Table 2.1
Failure Table

P	A	C	A	A	C	T	C
Failure table	0	0	1	1	2	0	0

When the failure table is successfully computed, the algorithm initiate its second phase, the matching process.

The matching algorithm starts by aligning the pattern with the text string. The first character of the pattern is then matched against the first character of the text string. For each match, the text to pattern comparison continues until a full match occurs. In the event of a mismatch, the column index corresponding to the mismatch is looked up in the failure table and the pattern is shifted from the left to the right corresponding to the value obtained in the failure table plus 1.



		Column	0	1	2	3	4	5	6	7	8	9	10	11
Match		Text	A	C	A	C	T	A	C	A	A	C	T	C
1	Step 1	Pattern	A	C	A	A	C	T	C					
2	Step 2		A	C	A	A	C	T	C					
3	Step 3		A	C	A	A	C	T	C					
3	Step 4		A	C	A	A	C	T	C					
1	Step 5				A	C	A	A	C	T	C			
2	Step 6				A	C	A	A	C	T	C			
2	Step 7				A	C	A	A	C	T	C			
0	Step 8						A	C	A	A	C	T	C	
0	Step 9							A	C	A	A	C	T	C
1	Step 10							A	C	A	A	C	T	C
2	Step 11							A	C	A	A	C	T	C
3	Step 12							A	C	A	A	C	T	C
4	Step 13							A	C	A	A	C	T	C
5	Step 14							A	C	A	A	C	T	C
6	Step 15							A	C	A	A	C	T	C

Failure Table						
0	1	2	3	4	5	6
0	0	1	1	2	0	0
0	0	1	1	2	0	0
0	0	1	1	2	0	0
0	0	1	1	2	0	0
0	0	1	1	2	0	0
0	0	1	1	2	0	0
0	0	1	1	2	0	0
0	0	1	1	2	0	0
0	0	1	1	2	0	0
0	0	1	1	2	0	0
0	0	1	1	2	0	0
0	0	1	1	2	0	0
0	0	1	1	2	0	0
0	0	1	1	2	0	0
0	0	1	1	2	0	0
0	0	1	1	2	0	0

Figure 2.7
Knuth-Morris-Pratt Matching Algorithm Workflow.

Figure 2.7 illustrates the pattern matching workflow for the KMP algorithm, using the failure table computed previously and represented in Table 2.1. Prior to the matching phase, the pattern P is aligned with the text T and the value of *Match* is set to 0.

Step 1: The first letter of the pattern P, is matched against the first letter of the text T. Since “A” is equal to “A” a match has occurred and the matching process continues. The $Match$ variable is incremented by 1 (*match* = 1).

Step 2: The second letter of the pattern P is matched against the second letter of the text T. Since “C” is equal to “C” a match has occurred and the matching process continues. The $Match$ variable is incremented by 1 (*match*= 2).

Step 3: The third letter of the pattern P is matched against the third letter of the text T. Since “A” is equal to “A” a match has occurred and the matching process continues. The $Match$ variable is incremented by 1. This process continues until a mismatch has occurred.

Step 4: The fourth letter of the pattern P is compared against the fourth letter of the text T . Since "A" is not equal to "C" a mismatch occurs. Note that the value of the variable "*Match*" is equal to 3. The algorithm now looks up the cell in the failure table whose column is equal to the value of $Match$. This is the cell at column 3 of the failure table which holds the value of 1. The algorithm then shifts the pattern from left to right by the value retrieved from the failure table plus 1; that is, the pattern is shifted by 2 positions with respect to the text. Similarly to the procedure for computing the failure table. Since a mismatch has occurred in this step, $Match$ is reset to 0 before the algorithm progresses.

This process continues until the entire text has been traversed. Note that due to the pre-computation of the failure table, the algorithm is able to find a full match in the text in less steps than a naive algorithm.

The worst and average complexity for the for the searching phase of the algorithm are $\Theta(n)$ and $\mathcal{O}(m+n)$ respectively, where n represents the length of the text T and m represents the length of the pattern P [20].

2.1.3 Boyer-Moore-Horspool Algorithm

The BMH algorithm uses a similar technique as the one used in the KMP algorithm; however, the comparisons, are made from the right to the left [21] [22]. The algorithm uses a bad character table to compare the suffix of the pattern against the text string, allowing the matching process to avoid numerous unnecessary comparisons.

The bad character table is constructed prior to the searching process in order to reduce the number of comparisons required to match a pattern P against a text string T. This is done by identifying the alphabet $\Sigma = \{A, C, T\}$ inside the pattern and by computing the longest shift that can be applied for a particular letter in the alphabet during the searching phase.

Patten Column Index	0	1	2	3	4	5	6
Pattern	A	C	A	A	C	T	C

		Bad Character Table			
		0	1	2	3
	Step 0	A	C	T	*
7	Step 1	-	-	-	7
7 - 0 -1 =	Step 2	6	-	-	7
7 - 1 -1 =	Step 3	6	5	-	7
7 - 2 -1 =	Step 4	4	5	-	7
7 - 3 -1 =	Step 5	3	5	-	7
7 - 4 -1 =	Step 6	3	2	-	7
7 - 5 -1 =	Step 7	3	2	1	7
-	Step 8	3	2	1	7

Figure 2.8
Boyer-Moore-Horspool Bad Character Table Construction

Figure 2.8 presents the procedural steps required to create the bad character table. Prior to the construction, the length of the pattern is calculated, let m be the length of the pattern, $m = 7$.

Step 0: The alphabet Σ of the pattern is identified programatically. The alphabet represents individual occurrences of every letter present in the pattern, all other letters are represented by the "$*$" symbol. The alphabet Σ represents the header of the table in Figure 2.8 (Step 0).

Patten Column Index	0	1	2	3	4	5	6
Pattern	A	C	A	A	C	T	C

		Bad Character Table			
		0	1	2	3
	Step 0	A	C	T	*
7	Step 1	-	-	-	7

Figure 2.9
Boyer-Moore-Horspool Bad Character Table Construction; First Step

Step 1: The length of the pattern ($m = 7$) is assigned to the last column of the bad character table (Column 3) as shown in Figure 2.9.

Step 2: The bad character shift is computed for the first letter of the pattern at column index 0 ("A") by using Equation 2.1.

$$m - PatternColumnIndex - 1 = shift \tag{2.1}$$

The result is then placed in the column corresponding to the letter "A" of the bad character table (Column 0) as shown in Figure 2.10

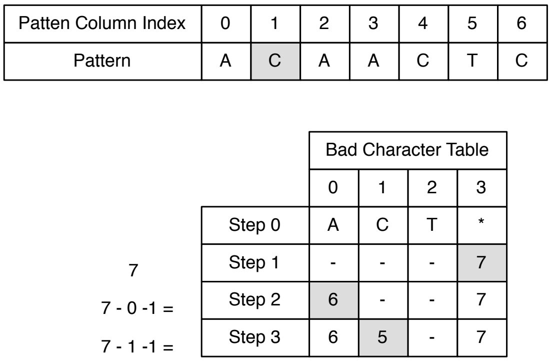
Patten Column Index	0	1	2	3	4	5	6
Pattern	A	C	A	A	C	T	C

		Bad Character Table			
		0	1	2	3
	Step 0	A	C	T	*
7	Step 1	-	-	-	7
7 - 0 -1 =	Step 2	6	-	-	7

-

Figure 2.10
Boyer-Moore-Horspool Bad Character Table Construction

Step 3: The bad character shift is computed for the second letter of the pattern at column 1 ("C") by using Equation 2.1. The subsequent result is then placed in the column corresponding to the letter "C" of the bad character table (Column 1) Figure 2.11

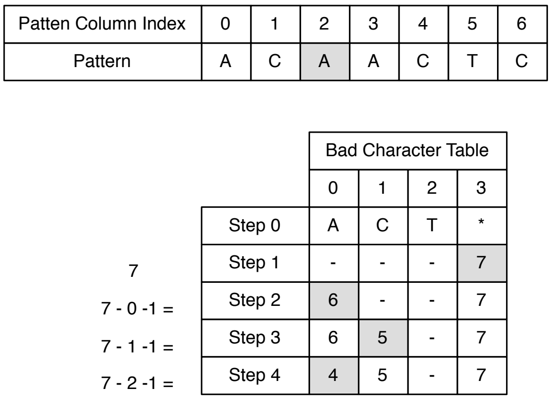


Patten Column Index	0	1	2	3	4	5	6
Pattern	A	C	A	A	C	T	C

		Bad Character Table			
		0	1	2	3
	Step 0	A	C	T	*
7	Step 1	-	-	-	7
7 - 0 -1 =	Step 2	6	-	-	7
7 - 1 -1 =	Step 3	6	5	-	7

Figure 2.11
Boyer-Moore-Horspool Bad Character Table Construction

Step 4: The bad character shift is computed for the third letter of the pattern at column 3 ("A") by using Equation 2.1. The subsequent result is then placed in the column corresponding to the letter "A" of the bad character table (Column 0), as shown in Figure 2.12



Patten Column Index	0	1	2	3	4	5	6
Pattern	A	C	A	A	C	T	C

		Bad Character Table			
		0	1	2	3
	Step 0	A	C	T	*
7	Step 1	-	-	-	7
7 - 0 -1 =	Step 2	6	-	-	7
7 - 1 -1 =	Step 3	6	5	-	7
7 - 2 -1 =	Step 4	4	5	-	7

Figure 2.12
Boyer-Moore-Horspool Bad Character Table Construction

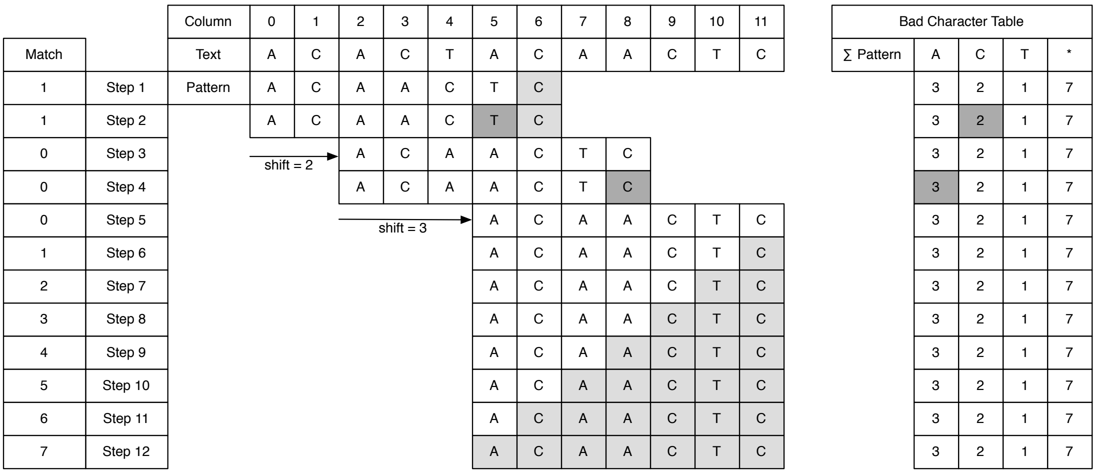
The process is repeated until the end of the pattern. Note that the last letter of the pattern will keep its existing value if it has previously been defined, or will be given the length of the pattern if the letter had not been encountered earlier in the process. Table 2.2 depicts the successful computation of the bad character table for the assumed pattern.

Table 2.2
Bad Character Table

Σ	A	C	T	*
Bad Character Table	3	2	1	7

With the bad character table computed, the algorithm can initiate the matching process.

The matching process of the BMH algorithm starts by aligning the pattern with the text string. The last character of the pattern is then matched against the corresponding character of the text string. For each match the two previous corresponding characters are compared against each other until a full match occurs. In the event of a mismatch, the bad character table indicates the number of position the pattern shall be shifted from the left to the right, however, note that the matching process starts from the right to the left.



		Column	0	1	2	3	4	5	6	7	8	9	10	11	Bad Character Table				
Match		Text	A	C	A	C	T	A	C	A	A	C	T	C	∑ Pattern	A	C	T	*
1	Step 1	Pattern	A	C	A	A	C	T	C							3	2	1	7
1	Step 2		A	C	A	A	C	T	C							3	2	1	7
0	Step 3		shift = 2		A	C	A	A	C	T	C					3	2	1	7
0	Step 4				A	C	A	A	C	T	C					3	2	1	7
0	Step 5				shift = 3			A	C	A	A	C	T	C		3	2	1	7
1	Step 6							A	C	A	A	C	T	C		3	2	1	7
2	Step 7							A	C	A	A	C	T	C		3	2	1	7
3	Step 8							A	C	A	A	C	T	C		3	2	1	7
4	Step 9							A	C	A	A	C	T	C		3	2	1	7
5	Step 10							A	C	A	A	C	T	C		3	2	1	7
6	Step 11							A	C	A	A	C	T	C		3	2	1	7
7	Step 12							A	C	A	A	C	T	C		3	2	1	7

Figure 2.13
Boyer-Moore-Horspool Pattern Matching Workflow

The searching process of the BMH algorithm is demonstrated in Figure 2.13. Prior to the search the pattern P is aligned with the text T and the value of *Match* is set to 0.

Step 1: The last letter of the pattern P at column 6 is compared against the corresponding letter of the text string. As a match has occurred *Match* is incremented by 1 and the column index is decremented by 1.

Step 2: As a match occurred previously, the letter of the pattern P at column 5 is compared against the corresponding letter of the text string. Since "T" is not equal to "A" a mismatch occurrs. The value of *Match* is reset to 0 and the bad character table is consulted to determine the number of places the pattern must be shifted to the right. As the "C" character was previously matched successfully, the lookup occurs in column 2 ("C") of the bad character table (Figure 2.13, Right).

Step 3: The pattern is shifted from the left to the right by the value retrieved from the bad character table in column "C".

Step 4: The last letter of the pattern P (Column 8) is compared against the corresponding letter of the text string. Since "C" is different from "A" a mismatch has occurred. As there are no prior matches, the bad character table is consulted at the column corresponding to the letter "A" of the text string.

Step 5: The pattern is shifted from the left to the right by the value retrieved from the bad character table in column "A".

The process continues until the entire text has been traversed. The complexity in the average case for the searching phase of the algorithm is $\Theta(n)$ where n represents the length of the text and $\mathcal{O}(nm)$ in the worst case where m represents the length of the pattern. The complexity of the pre-processing phase is $\mathcal{O}(m + \sigma)$ where σ represents the length of the alphabet Σ.

2.2 Multi-Pattern Matching

Multi-pattern matching is a natural extension of the single pattern matching problem. It is described by Crochemore *et al.* [23] as the problem of finding all occurrences of patterns in a finite set $X = \{P_1, P_2, P_n\}$ in a given text string T of length n.

In this section the core-concepts of multi-pattern matching are outlined. Firstly, different types of deterministic finite automata are presented. The Aho-Corasick algorithm is described highlighting its similarities and improvements over the single pattern matching KMP algorithm.

2.2.1 Deterministic Finite Automaton

Deterministic Finite Automaton (DFA) is a finite state machine accepting or rejecting a finite number of states and producing only one computation for each pattern [24]. DFAs are commonly used to model the behaviour of physical machines [24], patterns recognition and regular expressions [25].

A finite automaton can be defined as set of states Q, having only one initial state and one or multiple accepting states. Its initial state is denoted as $I \in Q$ while the accepting states are denoted $F \subseteq Q$.

The matching is done by starting at state I and matches the text string $T = \{t_1, t_2, ..., t_n\}$ against the different states of the automaton. The transition function between all states $q \in Q$ is denoted Δ and the alphabet used in the matching is denoted Σ. The full automaton A is represented in Equation 2.2 [20] [26].

$$A = (Q, \Sigma, I, F, \Delta) \tag{2.2}$$

The transition relation between states is defined by Equation 2.3 [20].

$$\Delta = Q \times \Sigma \rightarrow Q \tag{2.3}$$

Figure 2.14 represents a deterministic finite automaton. DFAs can only have one state active at a time, allowing only one path of the DFA to be followed when matching patterns.

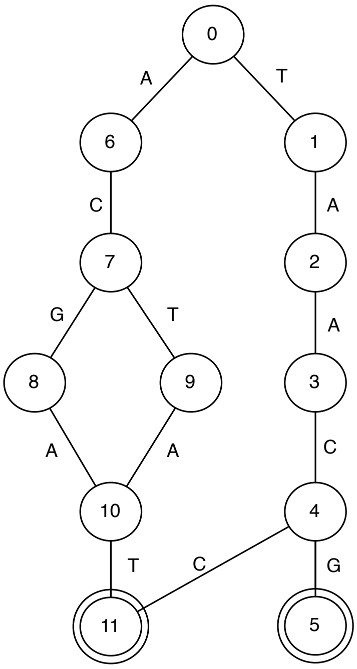


Figure 2.14
Deterministic Finite State Automaton

The states of the DFA are organised as a directed graph such as shown in Figure 2.14. Transitions are followed when the letter of the pattern is matched against the text string. For example, the DFA represented in Figure 2.14 contains four different patterns, $X(Q) = \{A, C, G, A, T\}, \{A, C, T, A, T\}, \{T, A, A, C, C\}, \{T, A, A, C, G\}$, it also contains two accepting states F represented by node 11 and 5. Accepting states or final states define a set of valid transitions through the trie. A pattern is matched, only if $q \in F$ is reached. In case of a mismatch, the matching restarts either from the starting state I (node 0) or follows a transition link (e.g. Transition link between node 11 and 4).

The complexity of the searching phase of the algorithm is linear and is represented as follows, $\mathcal{O}(n)$ where n represents the length of the input string, as there is only one path possible from the starting state I to the ending state F.

2.2.2 Deterministic Acyclic Finite State Automatons

DAFSA are also known as DAWG and are space efficient representations of tries [1] [26] while allowing to search for a string in $\mathcal{O}(m)$ where m represents the length of the input string, DAFSA reduce the overall number of nodes required for the representation of the trie [27].

Daciuk *et al.* [26] describes DAFSA as a 5 tuple automaton represented in Equation 2.2, containing only acyclic transition functions Δ. The size of the automaton is defined as $|A|$ and is equal to the number of states. $|Q|.X(\Sigma)$ is the set of all patterns over Σ, where Q represents the set of states, X the set of patterns and Σ the alphabet. The properties of a minimal automaton are defined by Watson *et al.* [28] [29].

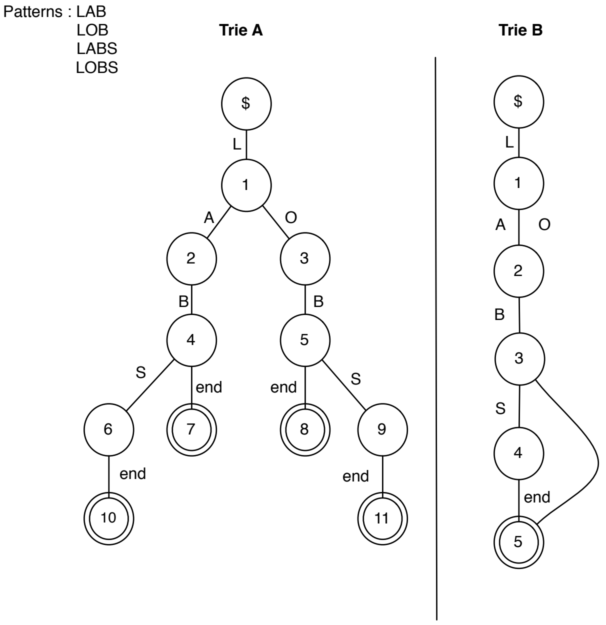


Figure 2.15
Conversion of a Trie to a Deterministic Acyclic Finite State Automaton.

[1] The word "trie" comes from the word re*trie*ve and is used to designate a tree where each transition represents the letter of a given alphabet.

Figure 2.15 demonstrates the properties of two tries, “A” and “B”, both containing the same four patterns, $X = \{L, A, B\}, \{L, A, B, S\}, \{L, O, B\}, \{L, O, B, S\}$ however the number of nodes and vertices required for trie A is much higher than the one required by the DAFSA represented by trie B. The primary difference between both tries, is the reduction of prefixes and suffixes at all levels of the trie, while allowing multiple vertices leading to a particular state. Note that by convention the $ sign represents the root node.

The construction of the DAFSA requires two steps, the first one is the creation of the trie, while the second one will travel the trie merging similar suffixes and prefixes, as well removing and adding vertices.

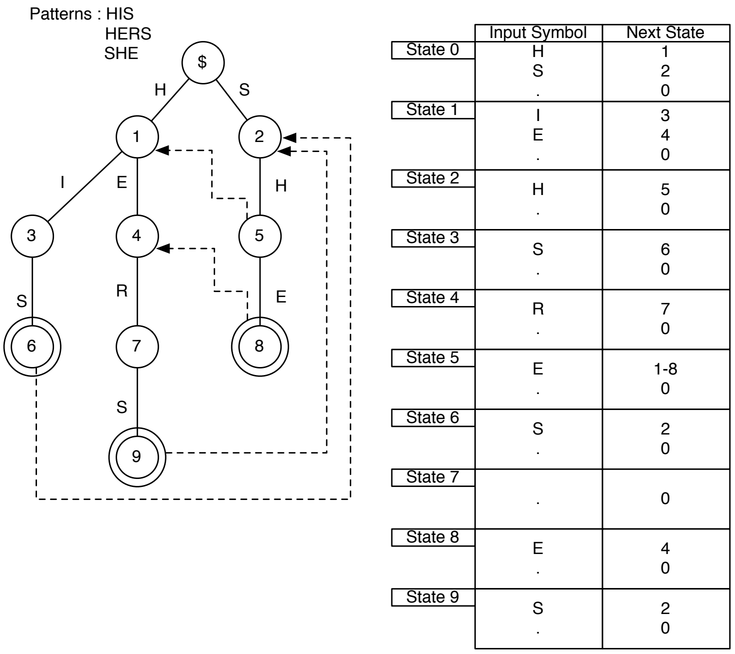
A variant of the directed acyclic finite state automaton has been described by Fredriksson *et al.* [30] optimising further the space required by the DAFSA by modifying the data structures required by each node.

2.2.3 Aho-Corasick Algorithm

The AC algorithm [31] is an extension of the Knuth-Morris-Pratt algorithm [20] that builds a deterministic finite automaton, or trie, with the sets of patterns X. Additional vertices also known as failure links added to the trie facilite the transition from one pattern to another.

The failure links, allow the matching process to search every pattern present in the trie in a single pass against the text string. This feature makes the AC algorithm one of the most common multi-pattern matching algorithms used.

The AC trie is built in two stages, the first stage consists of building the trie such as $A = (Prefix(X), \epsilon, X, \delta, Q)$ where A represents the automaton, $Prefix(X)$ the prefixes of the set of states Q, ϵ the empty character, X the set of patterns and δ the transition function. The transition function is defined by $\delta(p, p_n) = pp_n$ where p represents a pattern in the set of patterns X and p_n represents a character of the pattern p. Hence the trie transition can be defined as $pp_n \in Prefix(X)$. The second stage consists of adding failure vertices to the nodes at any state q, such as $\delta(q, p_n)$ represents the longest suffix of $Prefix(X)$ [32].



	Input Symbol	Next State
State 0	H S .	1 2 0
State 1	I E .	3 4 0
State 2	H .	5 0
State 3	S .	6 0
State 4	R .	7 0
State 5	E .	1-8 0
State 6	S .	2 0
State 7	.	0
State 8	E .	4 0
State 9	S .	2 0

Figure 2.16
Aho-Corasick Automaton

Figure 2.16 shows an Aho-Corasick trie for the patterns $X = \{H, I, S\}, \{H, E, R, S\}, \{S, H, E\}$ and its failure links represented by dashed-lines. Note that the failure links from the nodes to the root have been omitted intentionally to avoid confusion and that double circled states represent accepting states. The failure links to the root represents a mismatch at any state q. These links are represented in the state table (Figure 2.16, right) with the input symbol "." leading to the state 0 of the tree.

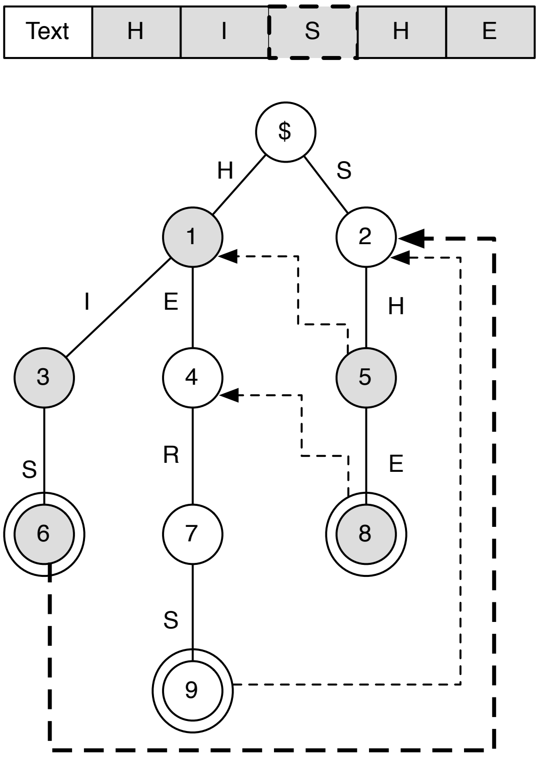


Figure 2.17
Aho-Corasick Searching Process

Figure 2.17 shows how the AC automaton is able to match multiple patterns in one pass over a text string by using the failure links. When reaching the terminal state of the first pattern in node 6, the automaton can then follow the failure link to match the next word and complete a subsequent match.

The computational complexity of the Aho-Corasick search process is $\mathcal{O}(n+k)$ where n is the length of the text string and k the number of occurred patterns appearing in the text string. The AC algorithm time complexity is linear and has contributed to its widespread adoption in multiple fields, ranging from pattern matching, through DNA sequencing [33] to intrusion detection systems.

2.3 Summary

Pattern matching is the dominant method to identify strings in numerous applications such as intrusion detection systems and DNA sequencing, in this chapter different single and multi-pattern matching algorithms were introduced. Firstly the two main single pattern matching algorithms were explained. The Knuth-Morris-Pratt algorithm introduced the failure tables and demonstrated major improvements over the naive pattern matching algorithm and its main competitor, the Boyer-Moore-Horspool algorithm introducing the bad character table as well as left-matching.

Secondly, different algorithms of multi-pattern matching were described, such as DFAs and DAFSAs. The Deterministic Finite State Automaton introduced the concept of state reductions in order to create Directed Acyclic Word Graphs. The popular Aho-Corasick algorithm was also described and presented as an extension of the KMP single pattern matching algorithms.

These algorithms represent the cornerstone of the field of single and multi -pattern matching. Parallel versions of these algorithms will be extensively studied, in future chapters of this thesis, with a focus optimisation in order to increase the overall throughput, using off-the-shelf hardware.

Chapter 3

Pattern Matching on GPU

3.1 Problem Statement

Computing models can be categorised into serial and parallel computing, resulting in two major processing approaches. Serially executed algorithms are essentially running on CPU. This execution strategy assigns a time interval called "quantum" to each process running on the CPU and executes them in a round robin fashion. At the end of the quantum, the CPU is preempted and is assigned to another process [34]. On the other hand, parallel computing utilises multiple processors to parallelise and simultaneously compute independent given tasks. This process is known as task parallelism. However, when given multiple independent data chunks to process concurrently, the process is known as data parallelism. The latter process, is well suited to GPUs, which are mapping data elements to parallel threads.

With the recent increase in processing requirements due to the amount of data being generated (e.g. search engines, DNA/RNA sequencing, data mining and IDS) and the increasing complexity of data processing, manufacturers resorted to increasing the number of processing cores to increase the performance of CPUs [35]. Given the comprehensive instruction set and the ability to execute independent instructions per thread CPUs are unable to scale to hundreds of cores per chip and have been unable to cope with the current processing requirements [36]. With the exponentially increasing data processing demand and CPU hardware limitations, GPUs are fulfilling the processing requirements as their architecture is well suited to address data parallelism challenges by performing massively parallel processing with the drawback of a simplified instruction set.

Traditional pattern matching algorithms run on a CPU and require a large amount of processing power due to the increasing pattern complexity and the expanding amount of data. High performance pattern matching applications can therefore benefit from multi-core off-the-shelf devices such as GPUs [37]. Their relatively small number of Logic Control Units (LCU) and high number of Arithmetic-Logic Unit (ALU) allow applications such as pattern matching to benefit from their massively parallel performance through the CUDA framework.

To this end, this chapters starts with an overview of the CUDA programming model and outlines the GPU architecture. It then moves onto a description of the current state of the art. Finally the contributions of this chapter are presented, in the form of a novel implementation and empirical investigation of two single-pattern matching algorithms and followed by the implementation and analysis of a novel GPU Log Processing library.

3.2 CUDA Programming Model

In 2006, Nvidia introduced the CUDA framework, a parallel computing platform and programming model allowing researchers to take advantage of the General Purpose Graphics Processing Unit (GPGPU) computations. GPGPU programming allows to take advantage of the massively parallel multithreading architecture by sending data from the host computer to the device GPU, allowing data to be processes at high rates. These applications, performing at supercomputer level, range from financial computing, fluid dynamics to DNA sequencing.

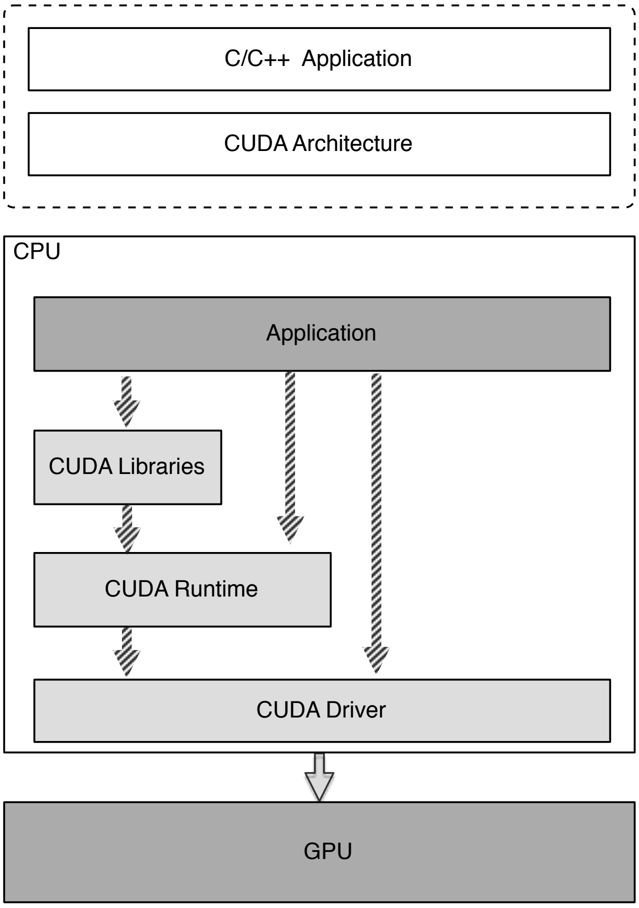


Figure 3.1
The CUDA Architecture and Programming Pipeline

GPUs can be represented as a massively-parallel many-core-processor. Therefore applications requiring data chunks to be processed concurrently can benefit from the multi-threaded environment, allowing each thread to process different chunks of data. The computation is done through the CUDA Software Development Kit (SDK). The CUDA SDK allows the programmer to communicate with the GPU using the ANSI C/C++ programming language which has been extended with a new set of directives and libraries. The code is compiled with the `nvcc` compiler on the CPU to the instruction set of the target architecture as shown in Figure 3.1.

Algorithms taking advantage of GPU processing power consist of one or multiple phases executed either on the CPU or the GPU. The segments of code exhibiting data parallelism are delegated to the GPU. The code that runs on the GPU is called a `Kernel` and is executed in six distinct steps: (I) The host allocates space on the GPU to transfer data; (II) The data is transferred from the host memory to the GPU memory using the Direct Memory Access (DMA) controller; (III) The host instructs the GPU to execute the kernel; (IV) The GPU executes the code in a massively parallel fashion; (V) The results are retrieved from the GPU by the host; (VI) The GPU memory is released from the host.

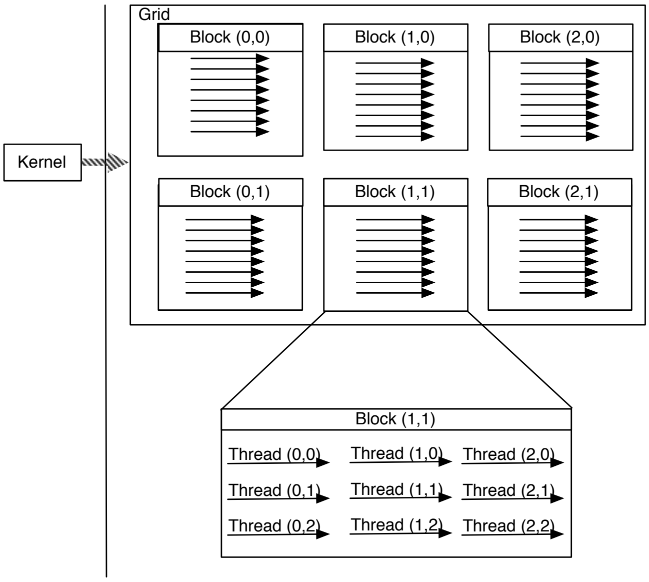


Figure 3.2
Grid and Threading Model

`Kernels` are executed in parallel, launching a finite number of threads to exploit the data parallelism capabilities of the GPU. Threads are organised in thread blocks, which are further organised into a grid as shown in Figure 3.2. All threads of a thread block are scheduled on the same streaming multiprocessor, however, more than one block can occupy the streaming multiprocessor at the same time. The threads within a thread block are organised into groups of 32 threads called `warps`. These are executed in a round robin fashion by the warp scheduler allowing multiprocessors to maximise the resources available on the GPU. This resource allocation is also based on the thread model, defining the operation of each thread on data points.

3.2.1 Single Instruction Multiple Thread Model

The earliest taxonomy of parallel execution was presented by Flynn [38] and is divided in 4 types. These types are based on the number of instructions available and the number of data streams the architecture can handle.

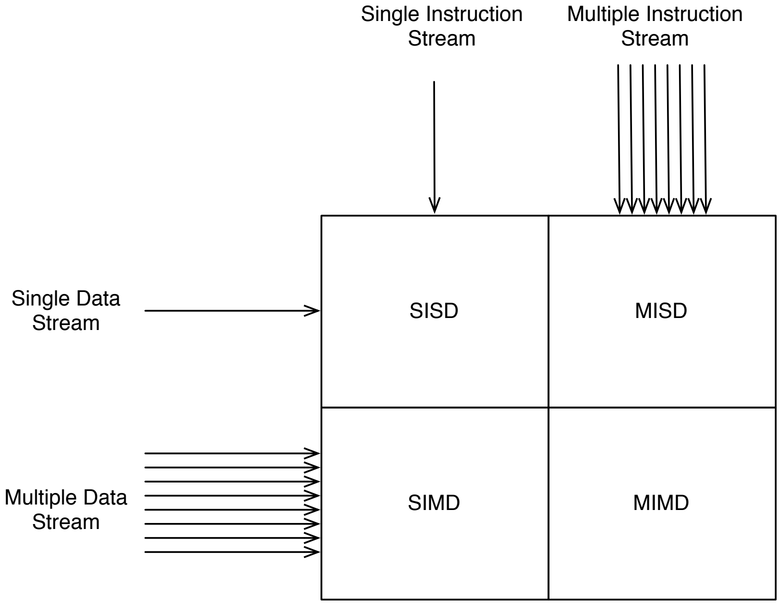


Figure 3.3
Flynn's Taxonomy

Figure 3.3 shows the four types, Single Instruction Single Data (SISD) describes the execution of a sequential algorithm, executing one instruction at the time. Single Instruction Multiple Data (SIMD) implies the execution of multiple data streams against the same instruction. Multiple Instruction Single Data (MISD) involves the execution of multiple instructions performing operations on a single data stream. Finally Multiple Instructions Multiple Data (MIMD) requires multiple instructions to operate on multiple data streams [38].

Nvidia's parallel programming standard relies on the Single Instruction Multiple Thread (SIMT) model. Both the SIMD and the SIMT are related and work in a similar fashion. Both models broadcast the same instruction to multiple execution units. However, the difference relies in the increased flexibility of the SIMT model, which allows the hardware and the software to take advantage of three key differences. The first one is the multiple register set for a single instruction. By using a “Scalar” spelling, the CPU is allowed to launch a thread per element. All the threads are then executed by the warp scheduler at a given cycle executing the same instruction, hence Single Instruction Multiple Threads. Each thread however processes different data as it possesses its own registers. The second factor is the possibility to use multiple memory addresses for a single instruction, allowing threads to access a different memory address via a single instruction without any latency costs required by the “address computation” required in a SIMD architecture. Finally the SIMT architecture allows the threads to follow multiple flow paths for a single instruction; the downside is that this operation has tremendous cost at runtime as only one flow path is executed at a time, hence, requiring some threads to “wait”, introducing further slow down, through randomised memory accesses [39]. Nevertheless these key differences allow researchers to design efficient algorithms on GPGPUs by designing code for operations required by a single thread working on one data element. This is ultimately extended by the CUDA framework at runtime for multiple data elements based on the thread block and grid configurations [40] [41] [42].

3.3 GPU Architecture

Three distinct GPU architectures are able to run CUDA. The first one was the Tesla generation brought along with the first version of the CUDA framework in 2006. The second generation was Fermi in 2010 and the third one, used in this research, is the Kepler architecture released in late 2012.

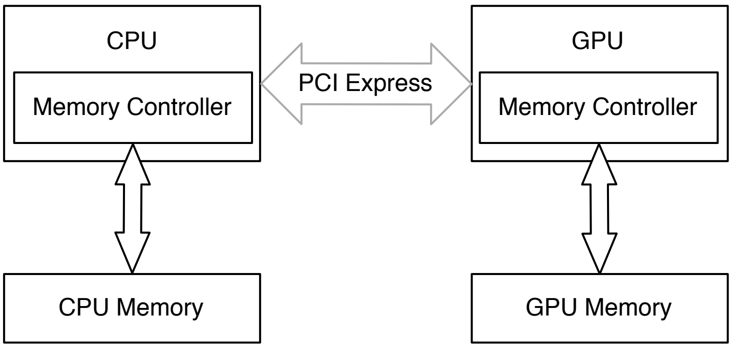


Figure 3.4
CPU and GPU Simplified Architecture

Figure 3.4 shows a simplified architecture of the CPU and the GPU. Note that both the CPU and the GPU have their own memory controllers and communicate together through the 16-lane Peripheral Component Interconnect (PCI) slots. Both the CPU and the GPU have their own virtual memory spaces. As a result the CPU is not able to read or write to the device memory. Applications must therefore explicitly allocate, copy and transfer data from one to another in order to process it, or to retrieve results [43].

The computational power of GPUs comes from Streaming Multiprocessors (SM). Each CUDA-enabled GPU architecture provides from two to a dozen SMs. More specifically each streaming multi-processor contains multiple execution units for 32-bit integer for single and double-precision floating point operations, a warp scheduler coordinating the executions units, dedicated hardware for texture mapping and Special Function Units (SFU) allowing the approximation of different mathematical functions (log, exp, sin, cos, etc). The streaming multiprocessors also contain two on-chip memory blocks: 48 KB of constant cached memory broadcasting data to the SMs and 64 KB of shared memory allowing data to be shared between multiple threads [43][44].

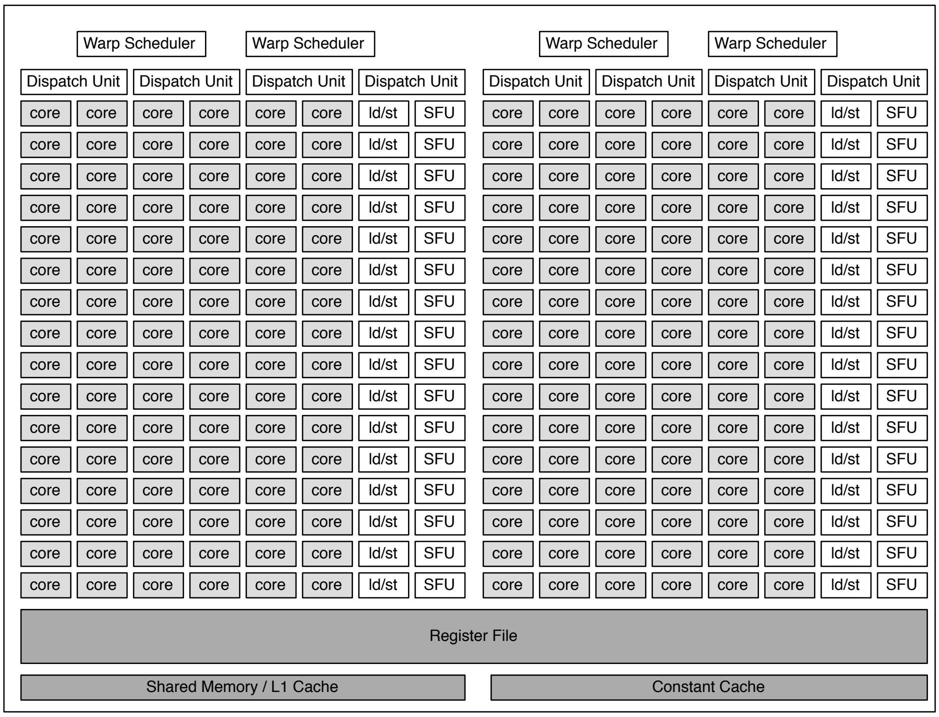


Figure 3.5
Kepler Streaming Multiprocessor

Figure 3.5 shows one streaming multiprocessor for the Kepler-class architecture. The number of streaming processors also known as CUDA cores, is increased by a factor of 6 compared to the latest Fermi architecture and features 192 CUDA cores. This architecture also possesses 13 streaming multi-processors, for a total of 2496 CUDA cores. The Nvidia Kepler K20m also amounts up to 7.1 billion transistors on a die area of 561 mm^2 and achieves a theoretical memory bandwidth of 208 GB/s.

3.3.1 GPU Memory Model

The data processed by the GPU can be stored in five different types of memories on the GPU. Each memory type can be accessed through the CUDA framework, and each has it own strength and weakness, trading complexity for speed. Highly optimised code takes into consideration all memory type and uses the appropriate memory, in order to increase the throughput of the algorithm.

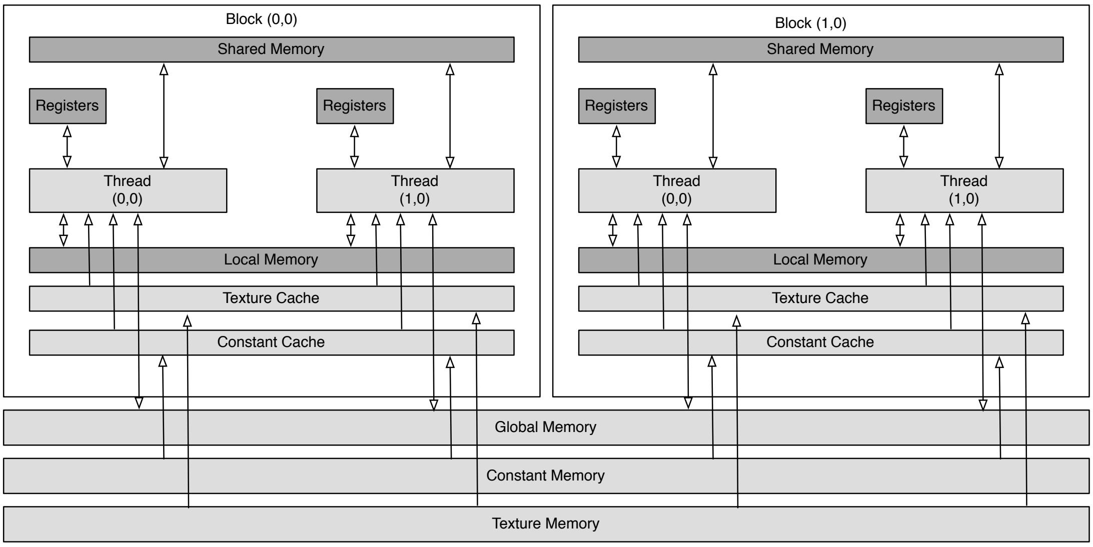


Figure 3.6
On-chip and Off-chip Device Memory Space

Global memory is the main off-chip memory abstraction used by the kernel. It is used to read and write to the device memory. Global memory requires up to 600 cycles to be accessed and is therefore considered slow. However, it is the largest memory space available on the GPU. To achieve the best performance when reading or writing data, `warps` must operate in a coalesced fashion and perform contiguous memory accesses. Algorithm design must take this into consideration for speed critical applications. Otherwise, un-coalesced memory access leads to a high penalty depending on the chip implementation. Lastly global memory is also accessible to all threads launched by the kernel [43] [44] [45].

Constant memory is a read-only memory optimised to broadcast information to multiple threads. The compiler uses constant memory to store “constant” variables defined either by the researcher in the code, or to store values that are not easily computed. The memory is a partition of the device memory, accessed through a specific set of instructions. The GPU accesses the memory over a constant cache of 64 KB located on the streaming multiprocessor. The constant cache is able to deliver a single value to every thread in a `warp` in a single clock cycle. However, this performance is lost if threads require different values and the process takes up to 600 clock cycles to complete [43] [44].

Local memory is off-chip memory, which contains the stack of every thread launched by the CUDA kernel. The memory also contains variables that cannot be contained by the registers, is not cached and can be read and written by each thread. Local memory can take up to 600 clock cycles to access [43] [45].

Texture memory is off-chip memory and like constant memory, texture memory is accessed through read instructions. These are required to use a texture cache located in the streaming multiprocessor. The cache is optimised for read only-accesses and does not require the accesses to be coalesced unlike global memory. The memory layout is optimised for 1D, 2D and 3D spatial locality [46]. Texture calls defined in the code define the type of data stored, as well as the access scheme (type of data stored i.e float, double, int). The memory can be read by all threads in a block and is optimised to speed up regularly executed operations through caching, as texture memory requires the same amount of clock cycles as global memory [43] [44].

Shared memory is a fast on-chip memory. It is primarily used to exchange data between threads within a block. Shared memory only requires up to 2 clock cycles to be accessed and is often used to store resources that need fast access [45]. The total size of shared memory is 64 KB. However, the default configuration is set to 48 KB of shared memory and 16 KB is assigned to the L1 cache. Shared memory can be read and written to by any thread [43] [47].

3.3.2 Experimental System Architecture

The different algorithms proposed have been assessed on a Supermicro server supplied with two Intel Xeon E5-2620 six core CPUs running at 2.0 GHz. The server as a total of 64 GB Random Access Memory (RAM) and is running Ubuntu Server 11.10 (kernel version 3.0.0-12-server). It is also composed of a Nvidia Tesla K20m graphic card. The card consists of a single Printed Circuit Board (PCB), composed of 192 CUDA cores distributed over 13 multiprocessors. The K20m has 5 GB of global memory and permits up to 26,624 active threads, with each streaming processor handling up to 2048 active threads.

3.4 Background

Pattern matching is the art of finding one or multiple patterns in a text string, regardless of the complexity of the pattern and the length of the text. Pattern matching has diverse applications ranging from medical to computer security.

Peng and Chen discussed the use of GPUs for large-scale pattern matching in *CUgrep: A GPU-based High Performance Multi-string Matching System* [48]. They implemented a parallelised version of the Backward Nondeterministic Dawg Matching algorithm (BNDM) in order to create a web page matching system. They made use of three different data structures and created two "B-tables", the first one containing for each character a bit-mask and the second one containing compressed bit-masks allowing the algorithm to take advantage of small memory spaces. During the matching process, the text representing the web pages was split into chunks of 2 KB, and the number of launched threads was equal to the number of pattern. The chunks of data were analysed sequentially. The algorithm achieved a speedup of 40 times compared to the sequential equivalent when using a Tesla C1060 over its CPU counterpart [48].

Schatz *et al.* [49] presented "Cmatch" in *Fast Exact String Matching on the GPU*. The algorithm is a GPU-based exact string matching algorithm for DNA sequences. It takes advantage of data structure, organised as a suffix-tree. The tree is constructed on the CPU and organised in two arrays, the node array and the children array. Both arrays are stored in texture memory of the GPU. The children nodes are organised in adjacent cells for locality. The DNA sequence and the two tree arrays are sent to the GPU prior to the matching phase. The DNA sequence is then divided into multiple chunks and processed in parallel. The algorithm is 35 times faster than its equivalent CPU-based version by using an Nvidia GTX 8800 and a 3.0 GHz Intel Xeon processor.

CUDA AGrep was presented in *A Fast CUDA Implementation of Agrep Algorithm for Approximate Nucleotide Sequence Matching* by Li *et al.* [50]. The algorithm aims at matching patterns in DNA and RNA sequences. The text string (DNA/RNA genome) uses a binary representation, where each nucleotide is represented by only 2 bits. The genome is then stored in global memory. During matching, the genome is divided into subgenomes, were each thread is used to process 4,096 nucleotides. Nucleotides are shuffled to achieve coalesced memory accesses during matching. By using an NVIDIA GeForce GTX285 graphics card, the authors achieved 70 fold and 36 fold speedups over the CPU-version of the k-difference AGrep algorithm depending on the maximum length of the patterns.

Lin *et al.* [51] proposed a multi-GPU Knuth-Morris-Pratt algorithm. The patterns are aggregated in a 1D array and are then divided by the number of GPUs in use, to be transferred to their respective GPUs along with the text string. The respective failure tables of each pattern is calculated on the GPU. The resulting matches are stored in a one dimensional array returned to the CPU. The GPU cards used were an NVIDIA Tesla S1060 and C1060, the authors achieved 97 times speed up over the counterpart Intel Xeon CPU.

An Intrusion Detection System on GPU was presented by Hung *et al.* [52]. The algorithm is divided into three phases. Firstly the packets are concatenated into a 1D array and the patterns are processed into hierarchical hash tables. The hash tables are stored by levels into 1 dimensional arrays; the number of levels is dependent on the length of the pattern. Each cell of the hash table contains either, a final state, a next state (in the next hash table) or a null pointer. Both are then sent to the GPU for the second phase, the matching process. During the matching process, each thread will process the same amount of data. When a match is found the corresponding match is written into a match table stored in the GPU global memory. Finally, the match table is transferred from the GPU to the CPU memory. The algorithm achieves a throughput of 2.3 Gbit/s when using an NVIDIA GeForce GTS 450, demonstrating 11 fold speedup over a sequential Aho-Corasick algorithm running on an Intel i3 CPU processor.

In *Efficient Pattern Matching on GPUs for Intrusion Detection Systems* Tumeo *et al.* [53] present a modified version of the Aho-Corasick algorithm for GPUs. The algorithm was designed to assign one single TCP/IP packet to each thread, allowing more threads to be launched and avoid overlapping between text chunks being matched. The algorithm was designed with a buffering mechanism allowing packets to be transferred during the matching process to produce a continuous feed. The results were organised in an multidimensional array, containing the packet index, the packet symbol and the matching state. The algorithm uses memory improvements and benefits from a binary representation of the Aho-Corasick states to reduce the overall memory requirements. The input structure was also redesigned by the author to reduce the loading memory transaction times. The authors demonstrated their algorithm on two Nvidia cards, a GeForce 9500M GS and a Tesla C1060, the optimised code shows 2.85 times improvements on the GeForce card and 6.67 times improvement on the Tesla card over the CPU implementation on an Intel Xeon E5345.

A GPU file carving algorithm has been presented by Marziale *et al.* [54] in *Massive threading: Using GPUs to increase the performance of digital forensics tools*. File carving algorithms are used by forensic investigators to retrieve files from an hard disk drive image. This is performed by looking for file headers and footers, also known as magic numbers. In this paper, they presented a parallelised version of an earlier work, the Scalpel carving tool. However, unlike the CPU algorithm running a modified version of the Boyer-Moore algorithm, the GPU runs a simple sequential searching algorithm. The algorithm works as follows. Firstly the searching rules (patterns) are copied to constant memory on the GPU prior to the search. Secondly, 10MB (text strings) blocks of data are sent to the GPU sequentially. The current block is stored in the GPU Dynamic random-access memory (DRAM); each thread is assigned 160 bytes to process and overlaps the next chunk of data by the length of the longest rule minus 1. The simple sequential algorithm implemented by Marziale *et al.* yields 2.22 times improvement over the CPU version of the Scalpel algorithm, with a disk image of 20 GB on an Nvidia 8800 GTX graphic card.

Lin *et al.* [55] presented a Parallel Failure-Less Aho-Corasick (PFAC) algorithm to accelerate string matching on GPU in *Accelerating String Matching Using Multi-Threaded Algorithm on GPU*. The idea behind the library, is to remove the failure-functions from the Aho-Corasick algorithm to reduce the thread divergence, as well as the number of pointers required in each trie node. The matching process works as follows; each thread is assigned one letter and travels the trie until a match is detected. When a match is detected, the thread ID is sent back to the CPU. However, when a mismatch is detected the thread computational process is terminated. The PFAC library was further improved in [56] by binding the state transition table to texture memory to benefit from the 6 kb cache reducing the memory transaction from global memory. A second optimisation lies in caching text string data in shared memory during the matching process to benefit from the low clock cycle access during the thread matching overlapping. The final improvement comes from introducing perfect hashing to reduce the overall memory requirements of PFAC, storing only valid transition for the failure-less AC trie [57]. The library demonstrated 3.9 Gbps throughput on an Nvidia GeForce GTX 295 and the implementation of perfect hashing required 4 times less memory.

Prior work described in this section was realised on older versions of the CUDA framework using outdated GPU hardware, increasing the throughput of the different algorithms by increasing the numbers of GPUs. The following Chapters presented in this thesis take advantage of the latest GPUs hardware and CUDA framework in order to improve the performances of pattern matching by using techniques such as loop unrolling, dynamic parallelism, and stacked memory, offered in the latest software and hardware version. Moreover, the algorithms are specifically designed with the hardware architecture in mind taking full advantage of the possibilities offered.

3.5 GPU-Accelerated Knuth-Morris-Pratt Algorithm

The Knuth-Morris-Pratt algorithm presented in section 2.1 has been chosen for its simplicity and efficiency on CPU. Furthermore, it is the precursor of the multi-pattern Aho-Corasick algorithm. The algorithm also presents opportunities for parallelisation strategies on GPUs and the ability to store its failure table efficiently in shared memory. Only a limited number of studies have considered improving upon the run-time performance of crucial single pattern matching algorithms, often improving its speed by running it concurrently on multiple GPUs [58] [59].

One solution proposed by Lin *et al.* in [59] allows 97 times speed ups over their CPU counterpart, but the methodology used presents multiple drawbacks. In particular, the algorithm requires the use of the OpenMP Application Program Interface (API) increasing considerably the complexity of the algorithm. On the other hand the improvements of the implementation rely on the number of GPUs used to boost the overall throughput. A more suitable solution would be to improve and parallelise the KMP algorithm by taking advantage of the different memory types of the GPU and best coding practices as well as improving and modifying the core of the serial version, demonstrating the high efficiency of the algorithm running on a single GPU, while keeping the complexity of the algorithm to a minimum.

3.5.1 Pre-processing Engine

As described in Section 2.1.2 the first phase of the KMP algorithm consists of a pre-processing phase. This step is realised before the patterns and text string are transferred to the GPU. The patterns and failure table are serialised by the pre-processing engine. The serialisation of patterns offers a simple way to keep track and arrange the patterns in a two dimensional array. A second two dimensional array containing the failure table is also computed for each of the patterns.

As the KMP algorithm is a single-pattern algorithm the processing engine is required to analyse the text as often as there are patterns. The process is time consuming and greatly benefits from the parallel nature of the GPU as described later in Section 3.5.4.

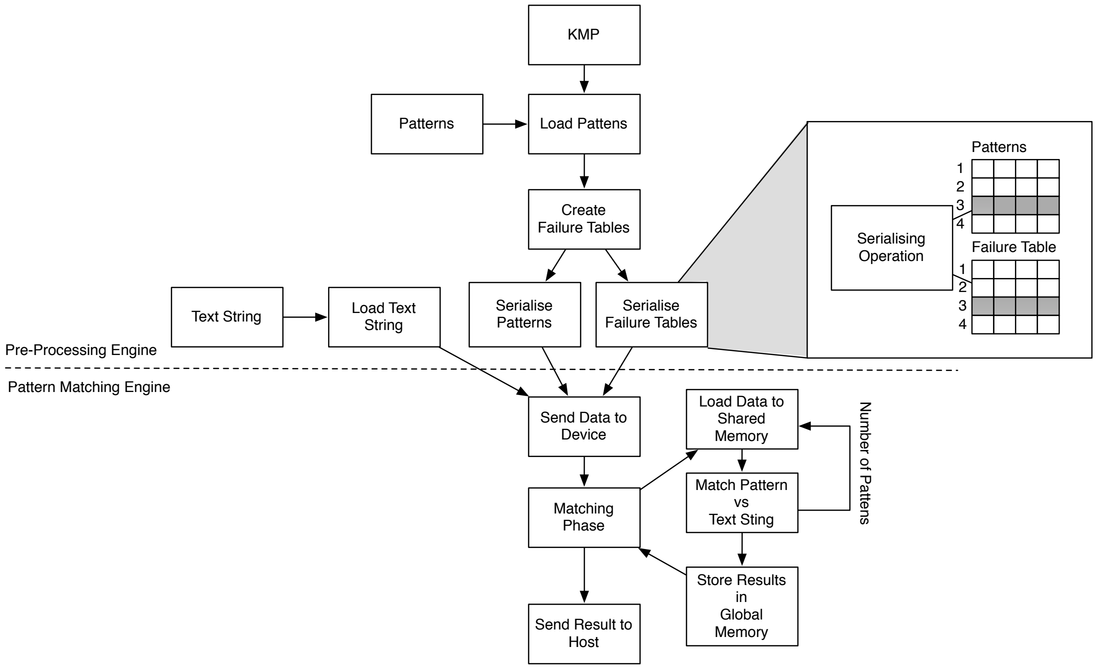


Figure 3.7
Flow Chart for the Pre-Processing and Pattern Matching Engine of the KMP algorithm

The pre-processing engine follows the steps described in Figure 2.7 in order to create failure-tables corresponding to every patterns, before storing them in the 2D array. Figure 3.7 demonstrates how the pre-processing engine aggregates the patterns and the failure tables. The failure tables and corresponding patterns can be found at the row ID of each table, avoiding the need for hash table linking patterns and failure tables as shown in Figure 3.7 (Top-Right – serialisation operation: Pattern Table RowID3 corresponds to the Failure Table RowID3). Figure 3.7 (Bottom) also shows the steps undertaken by the pattern matching engine explained in the next section.

3.5.2 Pattern Matching Engine

In order to parallelise the algorithm and maximise the transfer from the host to the device, the failure tables are serialised into a single array. This technique is recommended in order to avoid synchronisation delays between the CPU and the GPU for small batch transfers, hence, reducing the overall transfer time.

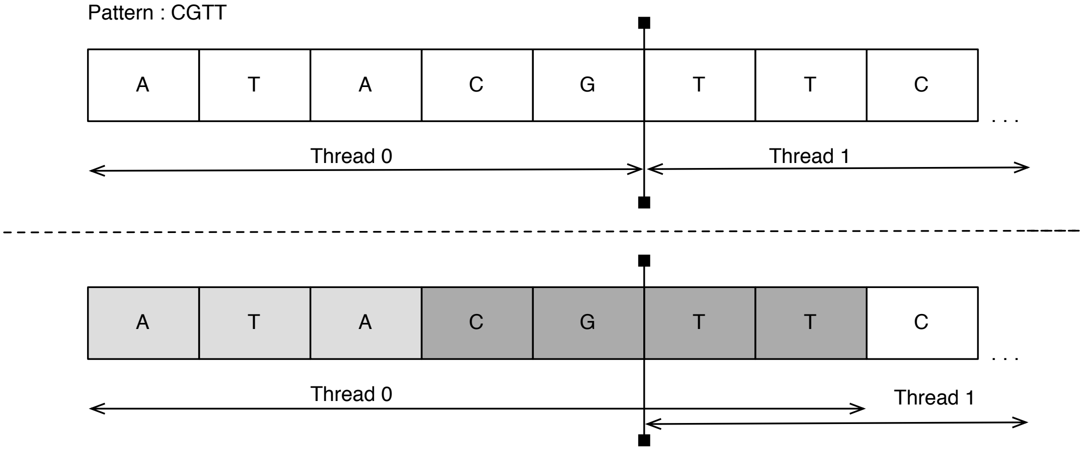


Figure 3.8
Patterns Divided Between Two Chunks of Data

The searching phase on the other hand can be parallelised by ordering each thread to process a number of characters, equal to the length of the longest pattern P in the failure table. As each single thread is looking for a given number of characters, the threads must overlap the searching area of their right neighbouring thread to avoid missing patterns split over two chunks. A text chunk represent a portion of text analysed by a single thread.

Figure 3.8 (Top) shows that the pattern $P = CGTT$ is split over two different chunks searched by two neighbouring threads. The problem is resolved in Figure 3.8 (Bottom) by purposely requesting $Thread_0$ to overlap the searched chunk of $Thread_1$ by the length of the longest pattern in the failure table minus 1. This technique allows every pattern in the text to be matched regardless of the size of the chunks processed by individual threads. The chunk sizes are calculated as shown in Equation 3.1, let N be the length of the text, M be the length of the pattern and $ChunkLength$ be the total length of the chunks matched by every thread.

$$ChunkLength = \frac{N}{Total_{ThreadNumber}} + (M - 1) \tag{3.1}$$

Figure 3.9 demonstrates how the processing engine stores the text and patterns before being loaded in the processing engine.

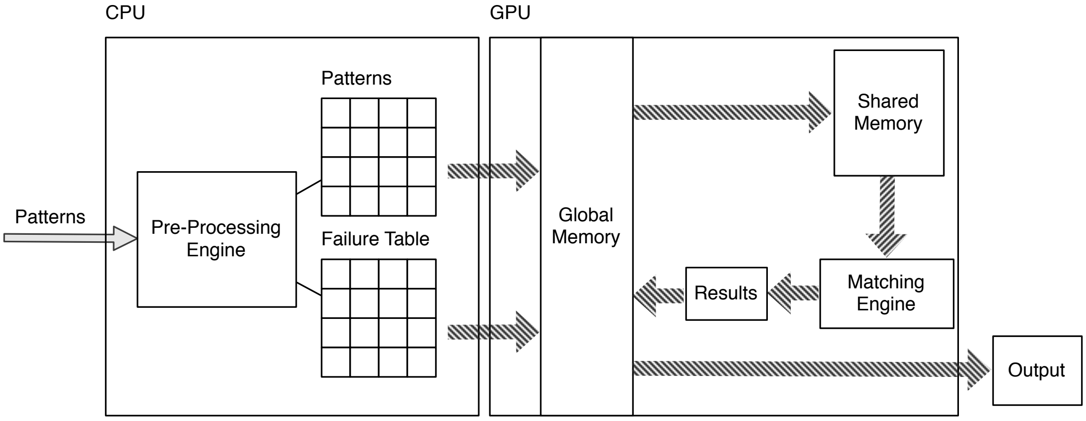


Figure 3.9
Pattern Processing Engine

Both the patterns and failure table are stored in global memory, before being transferred to shared memory before the matching. The results of the matches are stored in global memory, before being retrieved by the host computer and displayed to the user.

3.5.3 Code Optimisations

When launched by the CPU, the failure tables are stored on the GPU as 2 dimensional arrays in global memory; the text string is also loaded into global memory. Prior to the matching process of the first pattern, the failure table is loaded into shared memory along with the pattern shown Figure 3.10. The searching phase benefits from the shared memory as each thread searches the same pattern at the same time, resulting in faster memory accesses and higher throughput.

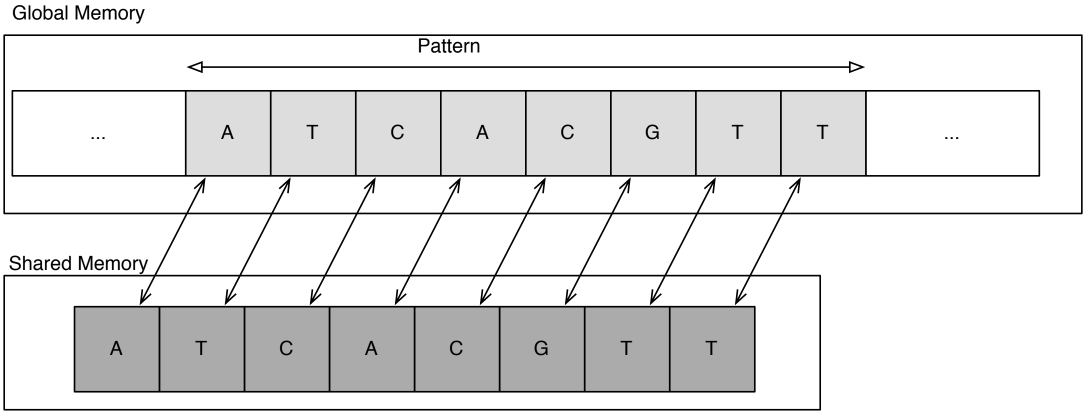


Figure 3.10
Threads loading patterns from global memory into shared memory simultaneously

The matching section of the implementation benefits from loop unrolling. This technique transforms the matching code "`for`" loop, allowing the optimisation of the execution [60]. This is done by avoiding a testing operation at each iteration of the loop, removing control instructions and branching penalties. The loop unrolling is achieved by re-writing the "`for`" loop in a number of individual statements [61].

3.5.4 Experimental Evaluation

To measure the improvements of the algorithm over its sequential version, different data sets have been used. In this way, the performance of the algorithm was evaluated against different type of alphabets length. Variations in the number of patterns, pattern length and number of threads were also considered. The performance of the algorithm was recorded using the standard time measurement API offered by the CUDA framework, however, in order to avoid errors introduced by background processes competing for resources, the results have been averaged over 500 runs, as this is standard practice [62] [63].

The first dataset used is the *Yersnia Pestis* bacteria's DNA, with an alphabet $\Sigma = 4$ and a size of 4.6 Mb. The second dataset used is a password file leaked on the internet with an alphabet $\Sigma = 26$ and a total size of 4.0 Mb. These datasets were chosen as they illustrate common problems encountered in different applications such as DNA sequencing, single string searching software, or more specific such as intrusion detection systems and digital forensic applications.

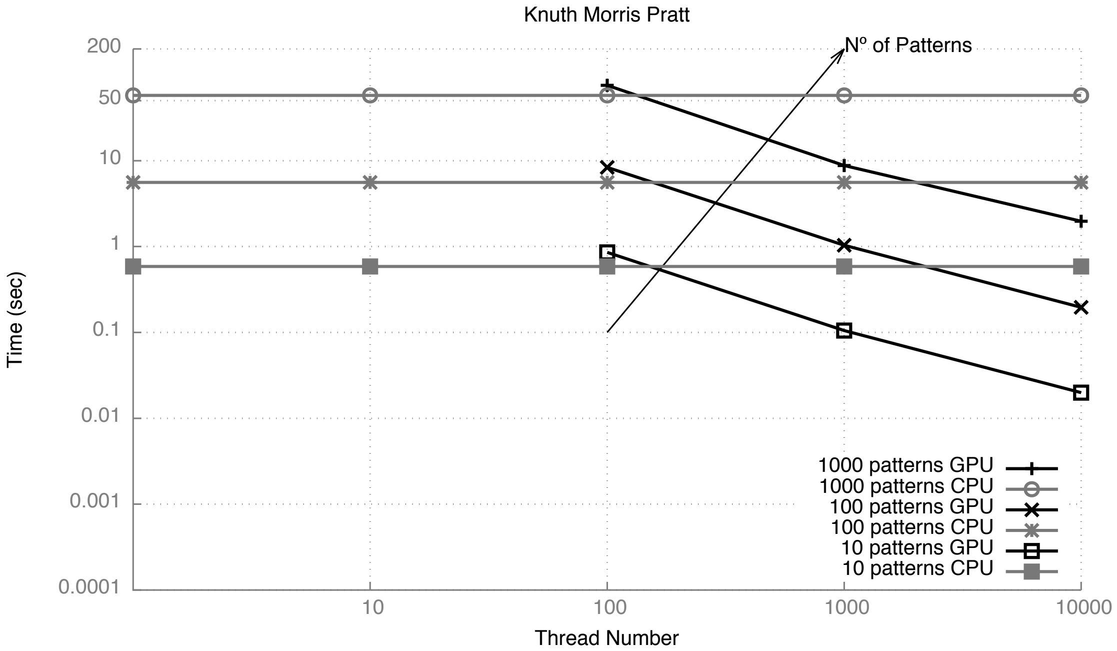


Figure 3.11
Results over Multiple Pattern Sizes and Thread Numbers

Figure 3.11 demonstrates the execution time of the algorithm running on CPU and GPU. The figure depicts different scenarios and demonstrates the impact of GPU multithreading over the CPU serial implementation. When a small number of threads are launched on the GPU, the CPU demonstrates better performance. This demonstrates the need for the GPU to be able to cache and use round robin techniques between the threads and warps launched in order to hide latencies and exhibit good performance.

Following the properties exhibited in Figure 3.11, it is possible to ascertain that in order to benefit from the power of the GPU and benefit from parallelisation, a significant number of threads is required by the kernel. Note that Figure 3.11 is represented using a logarithmic scale.

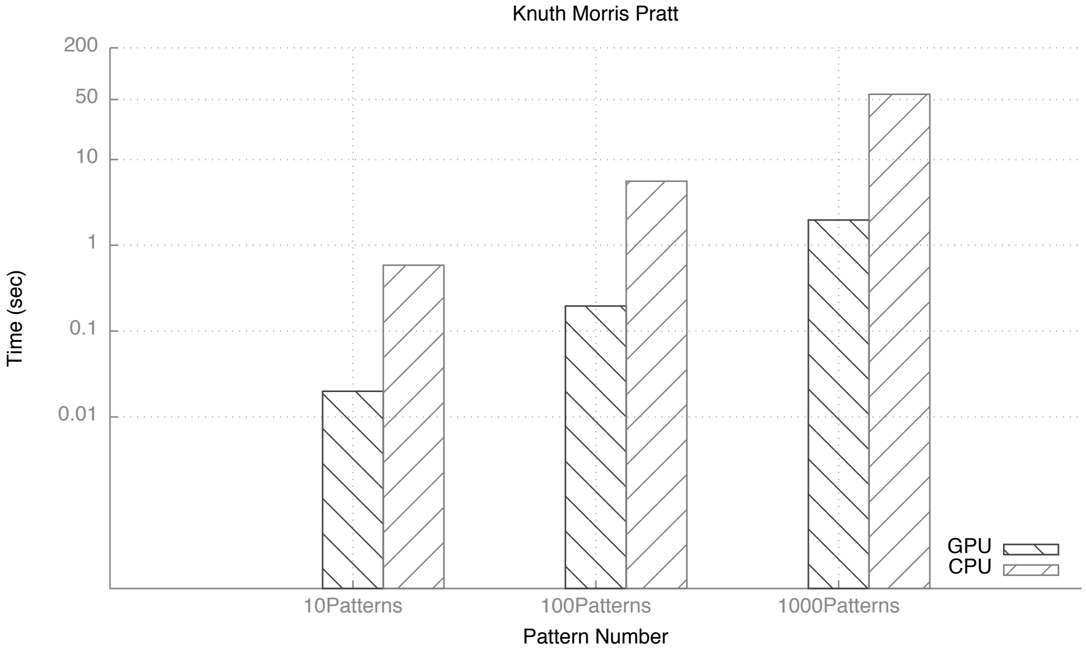


Figure 3.12
Execution time of Knuth-Morris-Pratt on CPU Vs GPU over different number of patterns

Figure 3.12 depicts the performance of the KMP algorithm over different number of patterns. The patterns used in the experiment have a constant length of 20 characters chosen pseudo-randomly from the *Yersinia Pestis* DNA test string. As shown the performance of the algorithm is decreasing according to the number of patterns being matched. Note that Figure 3.12 is represented using a logarithmic scale.

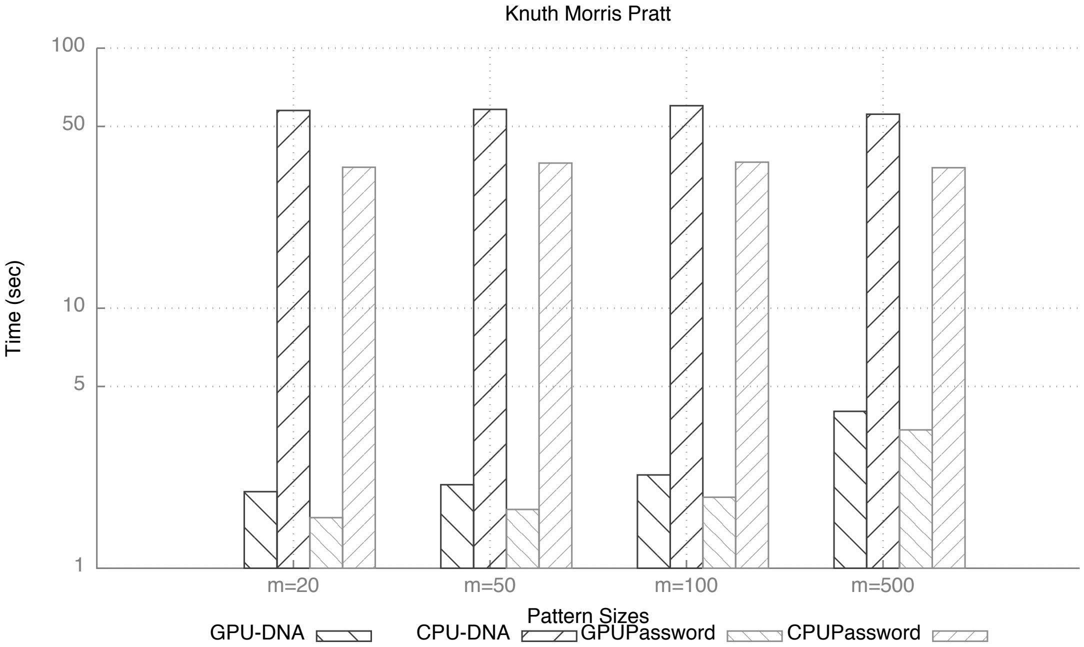


Figure 3.13
Speedups comparisons over multiple file sizes and pattern sizes

Figure 3.13 exhibits the comparison of the Knuth-Morris-Pratt algorithm over two text, the *Yersinia Pestis* DNA data and the leaked *Password* file. Both datasets are evaluated against different pattern lengths on the CPU and the GPU. As shown in Figure 3.13 the algorithm demonstrates a speedup of 14 fold over the CPU version of the algorithm, with a pattern length varying from $m = 20$ on text string with alphabet $\Sigma = 4$ and demonstrates performances of 10.2 folds over the CPU version of the algorithm against a text string of alphabet $\Sigma = 26$ with pattern length of $m = 20$.

As demonstrated in Figure 3.13, threads mismatch earlier with larger alphabets, hence reducing the thread divergence, making the matching process more efficient. At the same time, increasing the pattern size requires more comparisons to occur, increasing the divergence, hence slowing down the matching process [49].

Table 3.1
Throughput Comparison in Gbps

File	Threads	Number of Patterns	Throughput GPU in Gbps	Throughput CPU in Gbps
DNA	10 000	1	19.827875	0.47744
DNA	10 000	10	1.98144	0.06703
DNA	10 000	100	0.18429	0.00618
DNA	10 000	1000	0.01824	0.000624
Password	10 000	1	18.5337	0.62786
Password	10 000	10	1.8932	0.08896
Password	10 000	100	0.19854	0.008957
Password	10 000	1000	0.019975	0.0007941

The Knuth-Morris-Pratt algorithm demonstrates good performances with different alphabet sizes and demonstrates up to 19.8 Gbps of throughput when matching one pattern of size $m = 20$, with an alphabet of $\Sigma = 4$ and up to 18.5 Gbps with a pattern of size $m = 20$ and an alphabet of $\Sigma = 26$. However, when increasing the number of patterns to be matched the overall performance of the algorithm decreases. The algorithm also slows down with the increasing pattern length, demonstrating modest performance with larger number of patterns and larger pattern sizes as demonstrated in Table 3.1.

On the other hand, the Boyer-Moore-Horspool serial algorithm is known to perform better than the KMP in practice [21] [22]. These properties lead to an investigation of a massively parallel version of the BMH algorithm, in order to draw a comparison between the two most popular single pattern matching algorithms.

3.6 GPU-Accelerated Boyer-Moore-Horspool Algorithm

The serial version of the Boyer-Moore-Horspool algorithm is known to perform better than its KMP counterpart due to the large shifts it can make as it starts matching from the end of the pattern, as explained in [64] [65]. The algorithm is also know to perform better with smaller alphabets [17] [66].

One solution proposed by Zhou *et al.* in [67] shows a 40 time speed up over their counterpart CPU. The methodology presented, makes use of shared memory to store the text string. Each chunk of text string is stored separately in shared memory, this technique requires the authors to use a bank conflict solution and clear the chunks currently in use before copying new chunks in shared memory. A more suitable solution would be to store the chunks in global memory, reduce the coding complexity and modify the core of the sequential version to increase the efficiency of the algorithm. The implementation presented in this section follows this approach and stores the text string in global memory, while storing the patterns in shared memory. Furthermore, the implementation takes advantage of loop unrolling to increase the overall matching throughput and reduce branching.

3.6.1 Pre-Processing Engine

The serial BMH algorithm consists of two phases, the first is a pre-processing phase which is realised before the pattern matching occurs, as described in Section 2.1.3. In the parallel version, the pre-processing phase is itself divided in two. The first step is to create a bad character table for all the patterns being searched for. The second step requires the patterns to be serialised in a 2D array. The serialisation phase allows the user to transfer all patterns and their corresponding bad character tables in a single transfer.

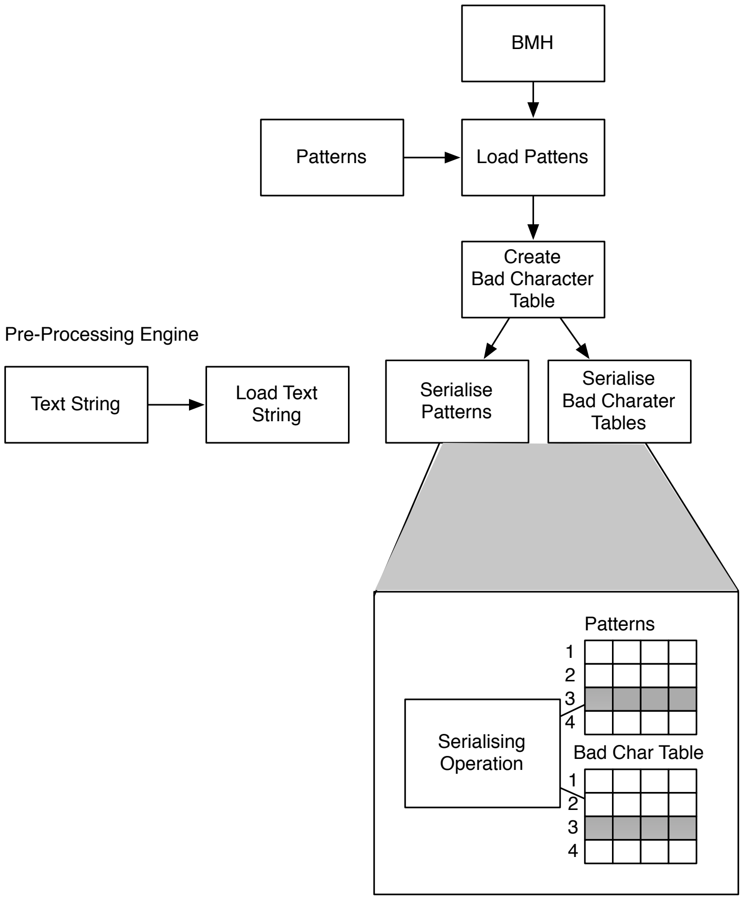


Figure 3.14
Boyer-Moore-Horspool Pre-processing Flowchart

Figure 3.14 illustrates the pre-processing steps undertaken before the matching process takes place. These steps are solely realised on the CPU. As for the KMP algorithm, the pattern and its corresponding bad character table can be found at the same rowID of both tables. This technique avoids the use of a tertiary table, or hash table to link the patterns and their bad character tables.

3.6.2 Pattern Matching Engine

The pattern matching engine is the core of the matching algorithm. In order to be parallelised, the patterns and the bad character table are sent in one batch from the host to the device. This ultimately limits the number of requests sent to the GPU.

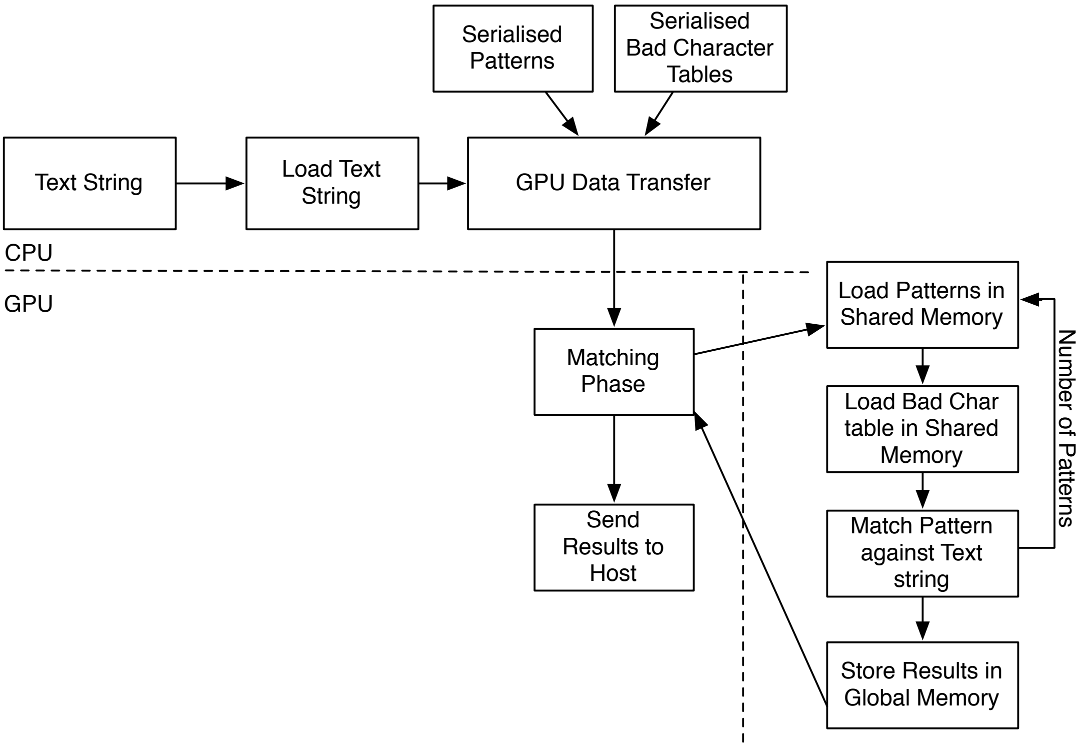

Serialised Patterns
Serialised Bad Character Tables
Text String
Load Text String
GPU Data Transfer
CPU
GPU
Load Patterns in Shared Memory
Matching Phase
Load Bad Char table in Shared Memory
Number of Patterns
Send Results to Host
Match Pattern against Text string
Store Results in Global Memory

Figure 3.15
Boyer-Moore-Horspool Matching Engine Flowchart

Figure 3.15 displays the behaviour of the pattern matching engine. As the data is stored in global memory, the patterns and corresponding bad character tables are stored in shared memory. The algorithm repeats this task according to the number of pattern to be searched for. This ultimately limits the throughput of the algorithm. Following the matching phase, the results are stored in global memory along with the matching indices. When the matching phase finishes, the results are sent back to the host and displayed to the user.

Unlike the KMP algorithm, the BMH algorithms starts to match backwards, but, the chunks of data analysed by the BMH algorithms are of similar length, and threads overlap each other, in order to match split patterns, as shown in Figure 3.8.

As with the KMP algorithm the loop unrolling technique is used, allowing the optimisation of the code execution. The loop unrolling technique helps in reducing the branching penalties by re-writing the loop in a number of individual statements.

3.6.3 Experimental Evaluation

In order to analyse the performance of the algorithm and measure the improvements over its sequential version, two datasets are used. These allow evaluation of the performance of the algorithm against various alphabet sizes, number of patterns and pattern lengths. The dataset used in this experiments is the same as the one described in Section 3.5.4, allowing a thorough comparison of both the KMP and BMH algorithm.

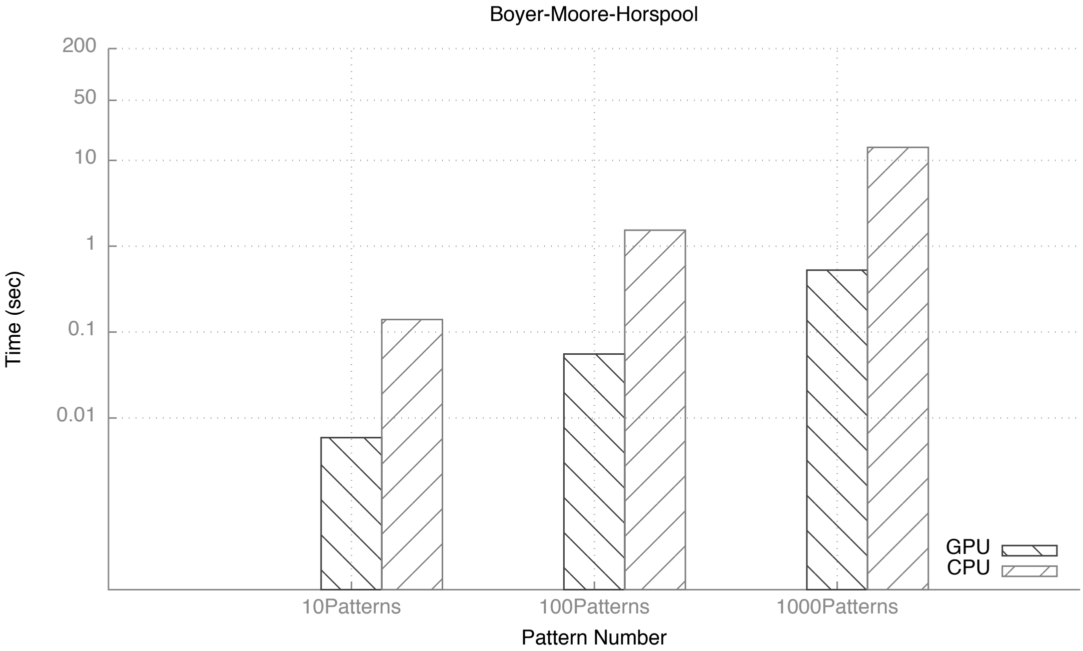


Figure 3.16
Execution time of Boyer-Moore-Horspool on CPU Vs GPU over different number of patterns

Figure 3.16 displays the performance of the BMH algorithm when exposed to different numbers of patterns. The patterns are uniformly chosen from the *Yersinia Pestis* DNA test string. The throughput decreases with the increasing number of patterns. This behaviour is expected, as the algorithm is required to skim through the text string as many times as there are patterns to be matched, hence the throughput decreases with the increasing number of patterns.

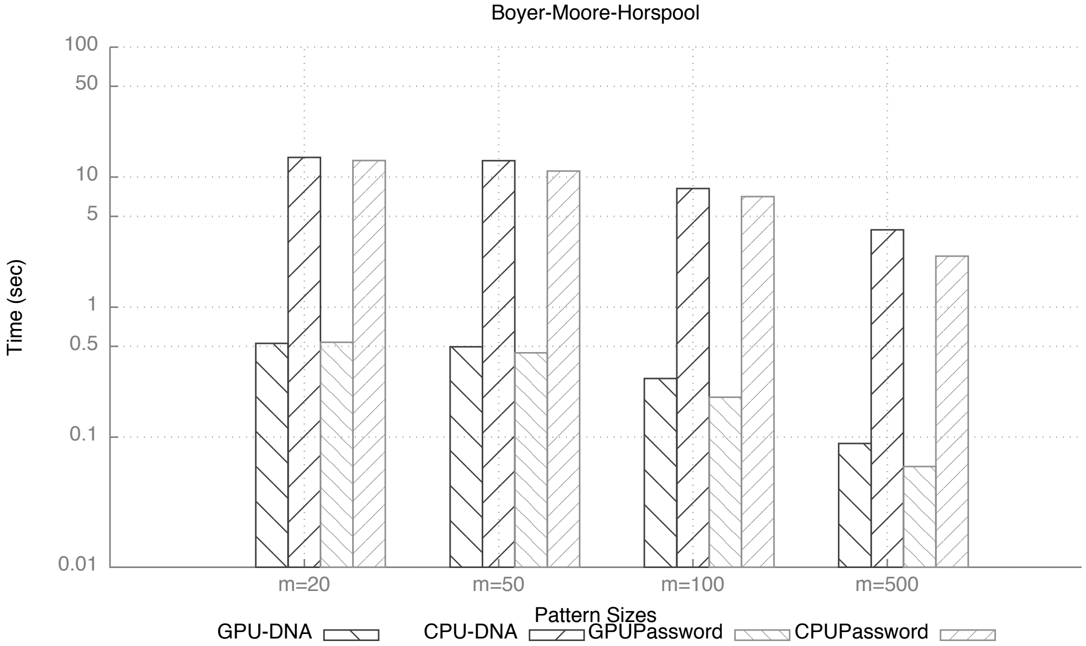


Figure 3.17
Speedups comparisons over multiple file sizes and pattern length

Figure 3.18 demonstrates the behaviour of the algorithm when used against patterns varying in length from $m = 20$ to $m = 500$ and against two different datasets. It is also shown that the BMH algorithm reacts counterintuitively with the increasing length of patterns. With the increasing length of the patterns and more characters to match, the overall throughput of the algorithm increases for the same number of patterns being matched. This phenomenon is due to the length of the pattern and the numerous mismatches occurring during the matching phase, inherently speeding the matching process, this phenomenon is described in [66] and in [68] where the same phenomenon is observed on a CPU implementation.

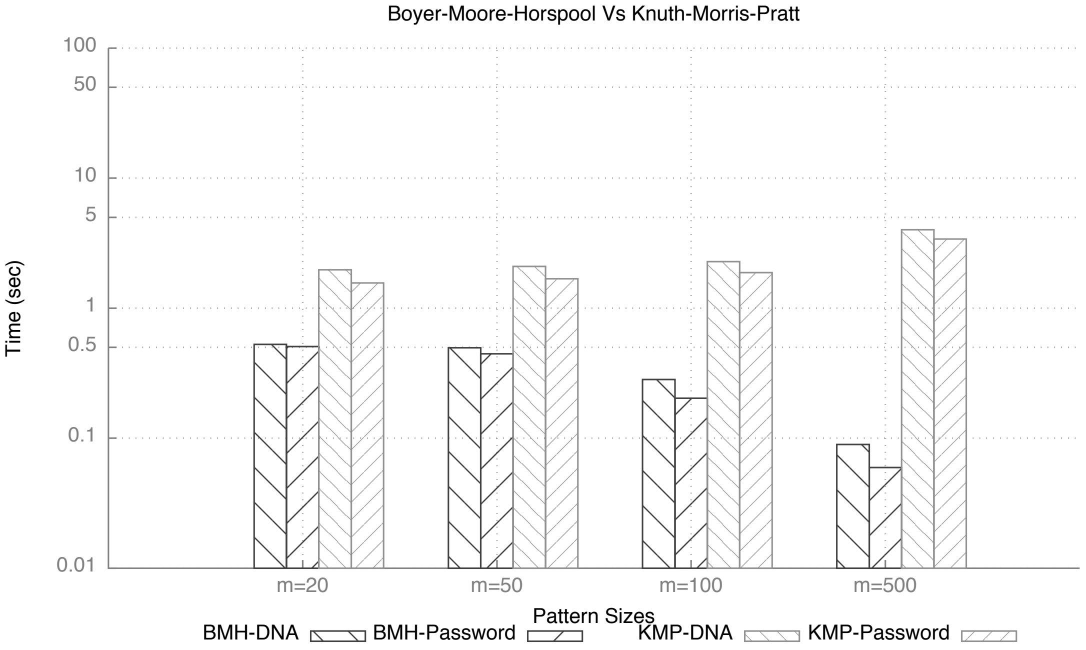


Figure 3.18
Summary comparisons of the Boyer-Moore-Horspool and Knuth-Morris-Pratt algorithm over multiple data files and pattern sizes

Figure 3.18 compares the throughput of the BMH and KMP algorithm over different pattern lengths. As shown, the BMH performs better when matching both the text string of alphabet size $\Sigma = 4$ and of size $\Sigma = 26$. The behaviour of both algorithm regarding the length of the pattern are antithetical, the KMP throughput decreases with the growing length of the patterns while the BMH algorithm displays the opposite behaviour.

As explained earlier, this is due to the divergence occurring during the matching phase, as well as the number of characters the BMH algorithm is able to skip by starting the matching from the end of the pattern, while the KMP algorithm is required to continue its matching process towards the end of the pattern, hence requiring more comparisons in order to match the same pattern.

Table 3.2
Throughput Comparison in Gbps

File	Threads	Number of Patterns	Throughput GPU in Gbps	Throughput CPU in Gbps
DNA	10 000	1	24.56211	1.01128
DNA	10 000	10	2.49981	0.09532
DNA	10 000	100	0.24735	0.009302
DNA	10 000	1000	0.02424	0.000929
Password	10 000	1	26.24363	1.23796
Password	10 000	10	2.68947	0.12620
Password	10 000	100	0.25902	0.01095
Password	10 000	1000	0.024875	0.0009788

In Table 3.2 the BMH algorithm is evaluated against two datasets of alphabet size $\Sigma = 4$ and $\Sigma = 26$, as well as against different number of patterns. The GPU implementation demonstrates 24 Gbps throughput when matching a single pattern with an text string and alphabet of size $\Sigma = 4$, and as expected the throughput increases when matching a single pattern against a text string of alphabet size $\Sigma = 26$. This behaviour is observable both on the CPU and the GPU and has previously been observed on a CPU [68]. The results also demonstrates over 26 fold increase in processing time over its CPU counterpart while matching a single pattern, and over 25 fold speedups when matching a thousand patterns at the same time.

This section demonstrated empirically the possibilities of single pattern matching on GPU and demonstrated the improvements of the massively parallel versions of the algorithm against their CPU counterpart, however, it was also highlighted that single-pattern matching algorithms do not cope well with matching increasing numbers of patterns. The next section evaluates a multi-pattern algorithm approach in order to increase throughput when matching one or more patterns, taking advantage of tree like structures.

3.7 GPU-Accelerated Failure-Less Aho-Corasick Algorithm

The Aho-Corasick algorithm presents multiple advantages over single-pattern matching algorithms, the main one being its processing throughput due to it's ability to match numerous patterns while traversing the text string only once, increasing drastically the matching throughput.

3.7.1 Engine Architecture

The parallel version of the Aho-Corasick however introduces the same problem as encountered by the KMP and BMH algorithms and requires boundary checking and overlapping. This requires the execution time of the thread to rise as shown in Equation 3.2 [57].

$$Total_{ThreadTime} = Chunck + LongestPatternLength_{Size} - 1 \tag{3.2}$$

When matching a text string with the parallel version of the Aho-Corasick algorithm, each thread is assigned to a chunk of the text string and is required to travel the cyclic trie. Due to the non-deterministic transitions of the AC algorithm the threads may travel through different path leading to divergence as they backtrack through the trie using the failure transitions to match different patterns in a single pass, as shown in Figure 3.19 [69].

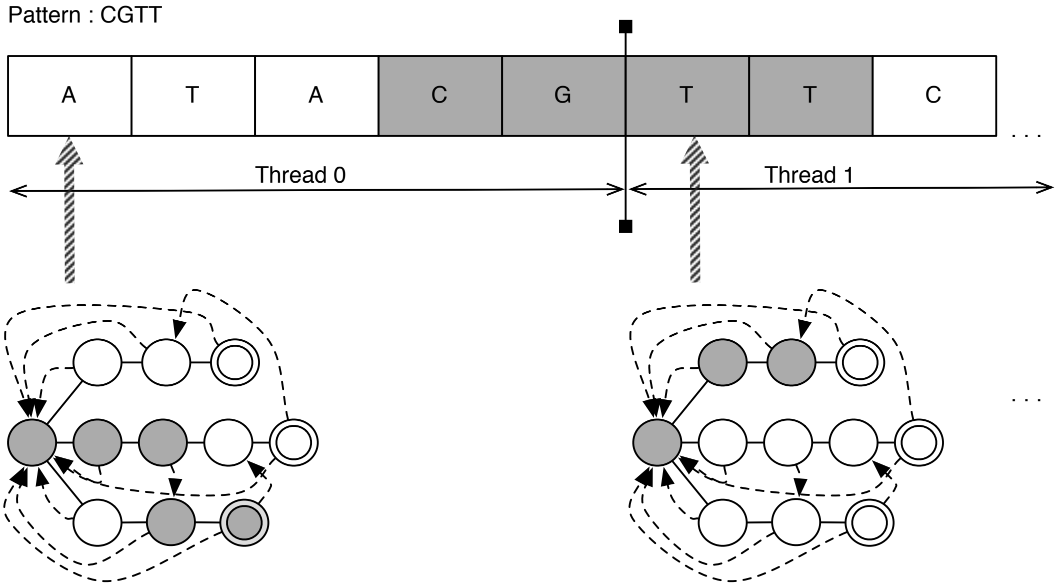


Figure 3.19
Overlapping Patterns with the parallelised Aho-Corasick algorithm

Figure 3.19 depicts the boundary problems introduced by allocating a chunk to each thread [53] [70]. When matching the pattern $CGTT$, thread 0 and thread 1 will scan the text and compare the letters against the automata stored in memory. In order to fully match the searched pattern thread 0 will have to overlap the chunk attributed to thread 1 by the size of the pattern-1.

The boundary and divergence problems can be hindered by using the PFAC algorithm introduced by Lin *et al.* in [56]. The failure transitions of the AC are removed from the trie, reducing drastically the number of valid transitions, hence reducing thread divergence [71].

This technique is also used by Memeti *et al.* in [72] in *Accelerating DNA Sequence Analysis using Intel Xeon Phi* where the authors propose to remove the failure links of the AC algorithm to allow one valid transition for each states of the automaton and guarantee a constant number of operations for each thread.

Furthermore, the PFAC algorithm can be adapted to different environments, as demonstrated by Aragon *et al.* [73] in *Pattern matching in OpenCL: GPU vs CPU energy consumption on two mobile chipsets* where the PFAC algorithm is implemented using openCL [74] on CPU and GPU mobile devices demonstrating the reduced power consumption and capabilities of the PFAC pattern matching algorithm.

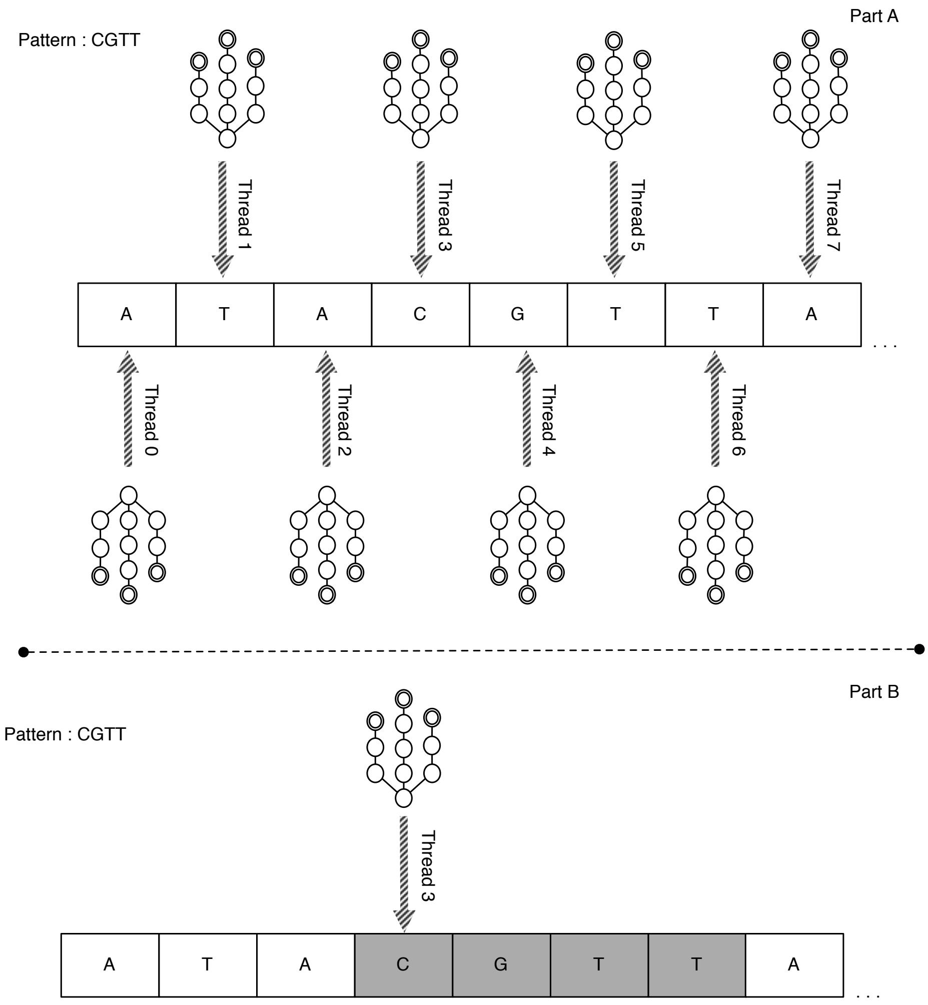


Figure 3.20
Parallel Failure-less Aho-Corasick

Figure 3.20 (Part A) shows how the threads are assigned to each letter of the text string. This technique allows the first access to memory to be coalesced, hence each thread in a warp to access contiguous memory. As the threads start matching, a number of them will mismatch and will be terminated liberating resources. As shown in Figure 3.20 (Part B), thread 3 will fully match the pattern “CGTT” and will overlap the letters allocated to thread 4, 5 and 6 which have mismatched and have thus been terminated during the first matching step. This technique reduces the work of each thread to the minimum, as the threads are terminated when no valid transition is found. However, the technique is subject to load imbalance as some threads will match more characters and require more time than their counterparts [57] [75].

In this work, the PFAC trie is improved upon by combining it with a trie reduction scheme, limiting the total depth of the trie to the minimum number of nodes required. This technique has previously been implemented by Vasiliadis *et al.* [76], on a standard Aho-Corasick algorithm, achieving a false positive rate below 0.0001% with a trie depth of 8.

This technique combined with the PFAC trie reduced the divergence by limiting the work of each thread to achieve a full match, while reducing dramatically the memory footprint required to store the trie, on the GPU. The use of a failure-less trie successfully exploits the massively parallel nature of the GPU by assigning each thread to one byte of memory to start the matching process as demonstrated in Figure 3.20.

Left

Pattern : ACCGA

States	A	C	G	T
0	1	-	-	-
1	-	2	-	-
2	-	3	-	-
3	-	4	-	-
4	-	5	-	-
5	-1	-	-	-

Right

States	A	C	G
0	1	-	-
1	-	2	-
2	-	3	-
3	-	4	-
4	-	5	-
5	-1	-	-

Figure 3.21
Reduced Trie Storage

In order to improve the memory footprint of the algorithm, the trie is stored in a compact 2D array. Traditionally, patterns are stored in state tables containing the entire alphabet as shown in Figure 3.21 (Left), however, the number of sparse cells increases exponentially. Therefore, it is possible to decrease the memory footprint by only allocating cells with the alphabet required by the patterns as shown in Figure 3.21 (Right). An application of the algorithm presented in this work is provided subsequently, highlighting the architecture of the implementation and the throughput achieved.

3.7.2 GLoP Architecture

The GLoP algorithm developed in this work and presented in this thesis supports two types of GPU memory architectures. The first one stores the patterns and text strings directly in global memory, while the second one, stores the patterns in global memory and the text strings in texture memory, allowing the algorithm to take full advantage of the texture cache. Both memory storage techniques present advantages and disadvantages. The global memory implementation requires coalesced memory accesses in order to provide efficient throughput, however, allows read and write memory operations. At the same time, texture memory provides cached data, but only allows read operations.

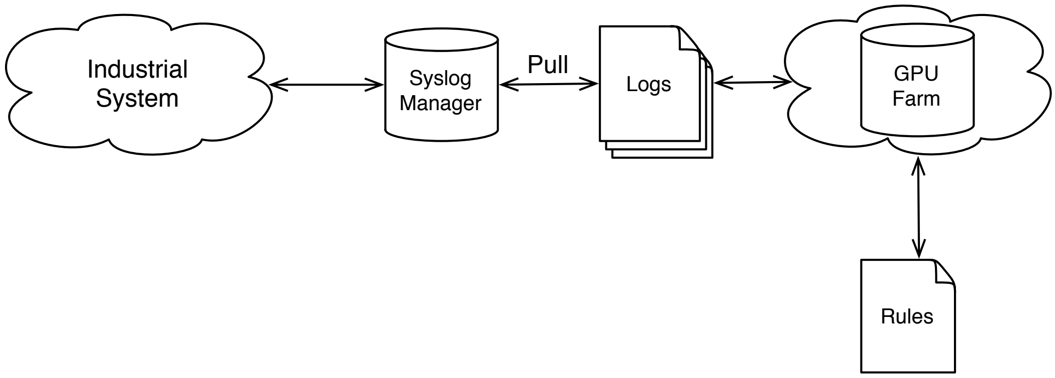


Figure 3.22
GLoP Algorithm Applied to a Massively Parallel Log Management System

As shown in Figure 3.22, the GLoP algorithm was implemented along with a log management system, allowing the use of memory efficient massively parallel pattern matching algorithm, to find patterns for deficient hardware sending logs, improve upon data analysis, or give the ability to forensic investigators to use the algorithm in big data incident response cases after a data breach in large networked systems.

Figure 3.22 demonstrates the typical scenario for the architecture. The industrial system archives its logs through the syslog manager [77], which pushes the logs towards the GPU system. The rules are fed to the GPU system and can be edited according to the requirements and needs of the log management system. As shown, the algorithm can be used on a single GPU for smaller systems and be extended or outsourced to a GPU farm, if more processing power is required.

3.7.3 Evaluation

In order to test the algorithm, different log files have been analysed and generated. The synthetic log files have been generated using the Mersenne Twister (MT) uniform pseudo-random number generator [78] and have an exact size of 5 MB and 100 MB. The uniqueness of each file is ensured by computing its SHA256 hash. The performances of the algorithm are also tested against the previously described Knuth-Morris-Pratt and Boyer-Moore-Horspool algorithms, in order to demonstrate the potential of the parallelised multi-pattern matching GLoP algorithm against parallelised version of single-pattern matching algorithm, as well as a serial implementations of the GLoP algorithm.

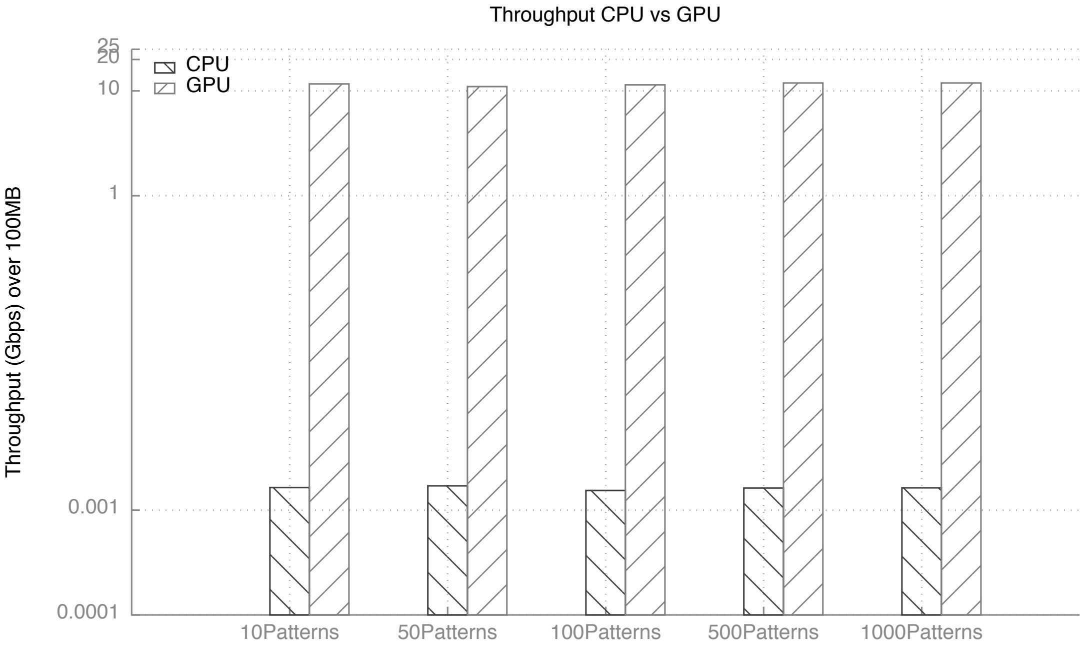


Figure 3.23
Comparison Between GLoP running on GPU and CPU

Figure 3.23 demonstrates the throughput achieved by the GLoP algorithm running on GPU with the patterns and the text string stored in global memory (Note the logarithmic scale). Figure 3.23 also highlight the weaknesses of the serial version of the GLoP algorithm running on the CPU. The GLoP algorithm also demonstrates that the failure-less trie approach maintains a constant throughput independent of the number of patterns (as expected from a multi-pattern matching algorithm [79]) as the maximum number of matched nodes for each thread corresponds to the trie length, defined by the trie reduction scheme, limiting the total depth of the trie. As shown in Figure 3.23 the GLoP algorithm achieves up to 11 Gbps throughput on GPU while the corresponding CPU version achieves throughput of only 1.7 Mbps. The throughput is measured using the CUDA event elapsed time function and quantified with the following Equation 3.3;

$$Throughput = \frac{8 * N}{Time_{gpu}} \tag{3.3}$$

Where $Time_{gpu}$ is the time elapsed during which the algorithm is running on the device, and $8 * N$ is the input length of the text in which the patterns are searched (bytes).

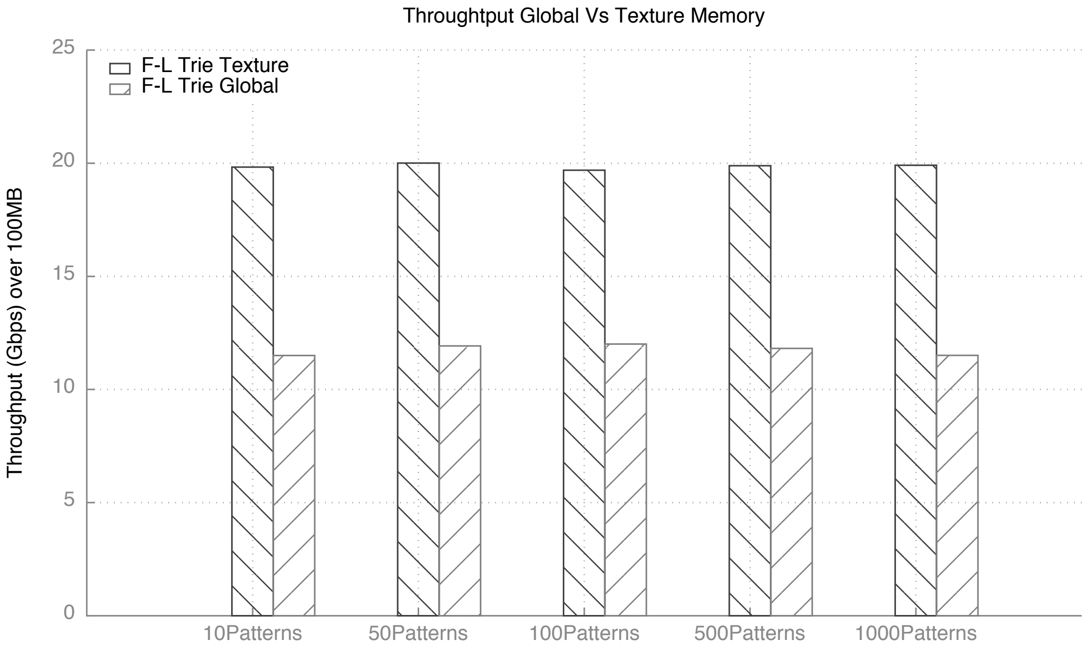


Figure 3.24
Comparison Between Data Processing from Global and Texture Memory

The implementation of the GLoP algorithm is further improved by using texture memory. The operations performed by the algorithm are identical with the only different being that the pattern table is being stored in texture memory.

Figure 3.24 highlights the throughput difference between patterns stored in global memory and texture memory. Furthermore, Figure 3.24 emphasises the importance and the throughput gained by cached data in texture memory, allowing threads to read memory in an non-coalesced fashion [80] and therefore demonstrates an overall throughput of 20 Gbps compared to global memory which demonstrates an overall throughput of 11 Gbps regarding the number of patterns used over synthetic data log files of 100 MB.

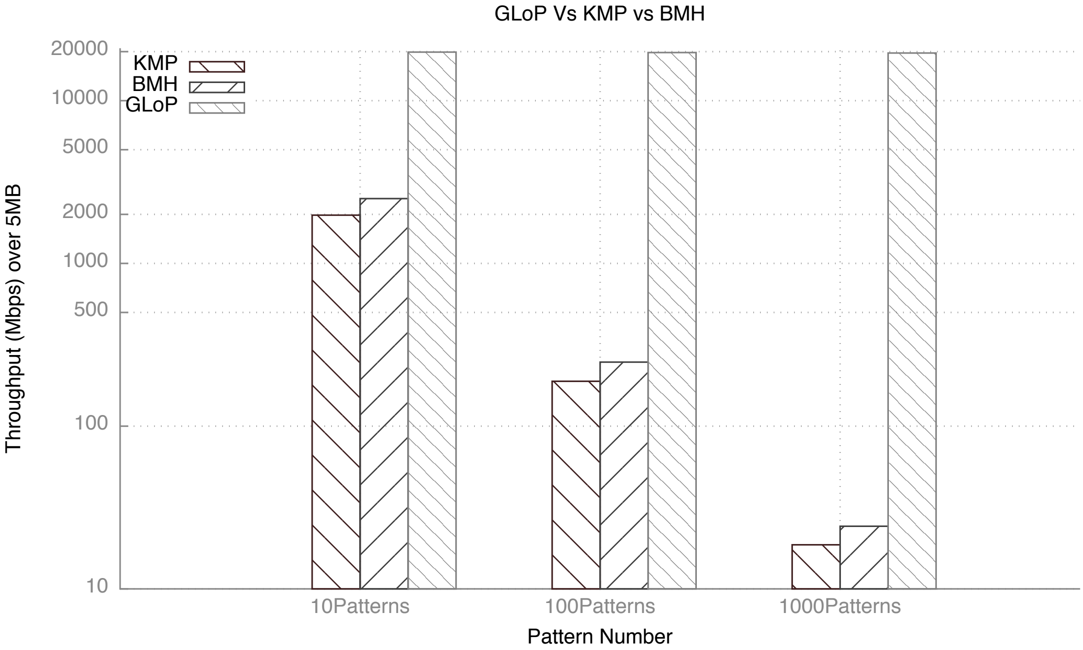


Figure 3.25
Comparison Between Single and Multi-Pattern Matching

Figure 3.25 demonstrates the efficiency of the multi-pattern GLoP algorithm against its counterpart single-pattern matching algorithm. As described earlier the throughput of the GLoP library on texture memory is constant, whereas the KMP and BMH algorithms are greatly affected by the number of patterns being matched. The GLoP algorithm using texture memory is able to demonstrate over 10 fold the performances of the KMP and over 7 folds the performance of the BMH algorithm when matching 10 patterns concurrently.

3.8 Summary

Graphics Processing Units are first and foremost off-the-shelf massively parallel devices dedicated to display graphics, however, it has been demonstrated that by using the CUDA framework, GPUs offer significant performance improvement by using their parallel capabilities in order to increase the overall processing throughput of computers.

Previous studies have shown that the performance obtained by running pattern matching on GPUs offered significant improvement over the counterpart serial implementation, however, the majority of those studies have focused on the design of pattern matching algorithms distributed over multiple GPUs in order to increase the throughput, instead of focusing on the quality of the code and optimisations offered by the CUDA framework and hardware available.

This work demonstrated the use of different techniques in order to increase the throughput of single and multi-pattern matching on GPU. The Knuth-Morris-Pratt and Boyer-Moore-Horspool algorithms were designed with a focus on efficiency, exploiting the different types of memory and loop unrolling techniques in order to increase the throughput of the algorithm and exploit the features from the CUDA framework and hardware. The KMP algorithm demonstrated over 19.8 Gbps throughput while matching a single pattern, while the BMH algorithm demonstrated over 24 Gbps throughput while matching DNA.

Both single pattern matching algorithms present however some setbacks, such as the need to skim through the text to identify each pattern. This phenomenon decreases the overall throughput of the single pattern matching algorithm, although the performance of the GPU still demonstrates 29 fold throughput over its CPU counterpart when matching a thousand patterns, demonstrating the potential of these experiments and the applicability of GPUs in pattern matching.

In order to solve the problems uncovered in the implementation of single pattern matching algorithms, a variant of the Aho-Corasick multi-pattern algorithm was investigated, allowing each thread to be discarded at mismatch. This technique was used to create the GLoP matching library. The efficacy of the library was demonstrated in Section 3.7.3 with two different memory schemes. The first memory scheme stores the patterns in global memory, demonstrating over 10 Gbps throughput, while the second memory scheme storing the patterns in texture memory, demonstrated over 20 Gbps throughput when matching over a thousand patterns simultaneously.

Previous implementations made of state tables, but did not handle the state explosion and increasing number of null cells required so store large patterns. In contrast our implementation limited the depth of the trie, reducing the overall size requirements and combined the bad character table technique from the BMH algorithm, which only stored the alphabet Σ of the patterns, allowing to reduce the sizes even further.

While this memory improvement is welcome, GPUs have a limited amount of memory, that will quickly saturate when used with a large number of patterns such as intrusion detection system, anti-virus, or GPU Log Processing rules. This matter is further investigated in the following Section.

Chapter 4

Highly Efficient Memory-Compression Scheme

The amount of data requiring analytics has increased dramatically in the last decade, and analysing large datasets has become key in underpinning innovation. This also represents a new era in data exploration, however it uncovers a number of challenges known as the 3Vs, Volume, Variety and Velocity [81].

Some of these challenges can benefit from massively parallel hardware such as GPUs [82], dramatically increasing the processing throughput of parallelised algorithms. However, off-the-shelf GPUs possess a limited amount of memory, driving the requirements for compressed data structures [83] [84].

Taking advantage of compressed data structures also enables faster data transfer between the host and the device, as well as enabling users to offload intensive computations to the older GPUs with restricted amount of memory while benefiting from their massively parallel capabilities [85] [86].

This chapter outlines a highly efficient memory compression scheme for pattern matching on GPU, modifying the core of the PFAC algorithm creating a novel Highly

Efficient Parallel Failure-less Aho-Corasick (HEPFAC) algorithm. Furthermore its efficiency is demonstrated and compared against the state-of-the-art.

4.1 Problem Statement

Graphics Processing Units are used in a plethora of different research fields, enhancing computational performance of personal desktops to GPU farms. The numerous features of modern GPUs and the CUDA libraries available allow for a large variety of algorithms to be modified to run on GPUs and benefit from their massively parallel capabilities.

To take advantage of these features, the computations need to be executed on the GPU and the data hosted in RAM (Host Memory) need to be transferred onto the GPU memory before any computation can occur as described earlier in Section 3.2. These are executed in six different steps, from the initialisation of the GPU to the GPU memory release.

The GPU computation improves the throughput of the algorithm, but the data transfer from the CPU and the GPU is often designated as the bottleneck of the architecture, increasing the overall processing time [87] [88]. The memory management process of the GPU also often allocates pointers sparsely depending on the free memory available, causing problems when working with a large number of data structures and pointers [89] [90].

Furthermore the domains to which pattern matching algorithms apply are in constant expansion and are increasingly demanding. The number of patterns to be matched is increasing exponentially [91] with over 82,000 new malware patterns reported everyday in 2013 and over 200,000 new malware patterns reported per day in 2014 [92]. This trend is also valid in other domains such as data mining [93] and DNA analysis [94], demonstrating high requirements for efficient pattern storage when performing pattern matching on GPUs.

This chapter firstly elaborates on the current state-of-the-art and the background, then moves onto the proposed memory compression scheme. A description of the storage model, the node reduction technique and the trie compression are given. The performances of the proposed algorithms are evaluated against different alphabet sizes, memory requirements and throughput. A second version of the algorithm is then proposed allowing for a further increase in the throughput by using a different memory representation allowing to make use of the different memory hierarchies included in GPUs. The throughput of algorithm is then evaluated and a performance analysis on the Amazon EC2 cloud is outlined.

4.2 Background

A GPU-based anti-virus search engine was proposed by Vasiliadis *et al.*[76]. The search engine presented by the authors, was massively parallelised using a modified version of the Aho-Corasick trie. The GrAVity engine presented by the authors uses a storage mechanism called prefix matching, where only the prefix of the signatures are stored, while the leaf nodes store the remainder of the patterns, also known as suffix. The authors reduce the overall number of trie levels to eight resulting in less than 0.0001% of false positives. A false positive occurs when the text string contains at least eight characters of the pattern but the remainder of the pattern has a different suffix. The authors demonstrated over 100 times the performances of ClamAV on an Intel Xeon E5520 CPU versus an NVIDIA GeForce GTX295, which is composed of 2 Printed Circuit Board (PCB)s. The GrAVity implementation however presents a major memory problem as described by the authors, with each node of the trie requiring 1 Kb of memory.

Seamans *et al.* [95] present one the first parallel hybrid (CPU/GPU) engine for virus matching in *Fast Virus Signature Matching on the GPU*. The authors parallelised the matching engine by searching for 2 bytes of the virus signatures in each files. The two first bytes of the signature are then mapped to a 64,000-entry array, leading to one or more signatures. When a file matches the two first bytes, the entry array is consulted and the four following bytes are compared. When a match is detected the host is notified and the file is compared against the full signature. The authors demonstrated 27 fold improvement by using an NVIDIA GeForce 7800 GTX over their CPU implementation on a 3 GHz Intel Pentium 4. The authors however pointed out a lack of flexibility of their approach and the limitations of the 64,000 entry arrays. Another downside of the six-bytes approach is the six-bytes pre-matching technique, which requires each signatures to have a unique starting sequence.

Trapnell *et al.* [96] presented MUMmerGPU in *Optimizing Data Intensive GPGPU Computations for DNA Sequence Alignment*. The MUMmerGPU library allows to accelerate data-intensive computation for next generation DNA sequences. This is realised by creating a stackless depth-first-search kernel, also known as threaded tree, running on an nVidia GeForce 8800 GTX and demonstrating 13 fold performances compared to the CPU version of the code running on a 3.0 GHz dual-core Intel Xeon 5160. The authors presented two versions of the code, MUMmerGPU 1.0 and MUMmerGPU 2.0, with the second version performing 4 times as fast as the first. The second version of the algorithm stores the pattern in two-dimensional layout in texture memory. The authors uses the Ukkonen's algorithm in order to construct the suffix trie [97]. The nodes of the trie are divided in 16 byte structs called the nodes and children structs. The node structures contains the depth of the node and the address of the parent and suffix nodes, whereas the children struct contains the address of the five possible children (A,C,G,T and \$) requiring 32 bytes of data to store each node, with an alphabet of $\Sigma = 4$.

Burtscher *et al.* [98] describes a tree-based implementation of the barnes-hut algorithm in *An Efficient CUDA Implementation of the Tree-Based Barnes Hut n-Body Algorithm*. The algorithm describes the implementation of an *n-body* algorithm in a tree like fashion. The tree is stored in 1D stack, representing elements in an array instead of a heap object. This allows the authors to avoid the use of pointers and translating the pointers between the host memory and the device memory. The authors demonstrate 72 times speedup over the CPU serial implementation using a nVidia Quadro FX 5800 GPU, when simulating over 5,000,000 bodies. An efficient use of stacks (array based) tree levels is demonstrated, in order to store the tree, however the trie implementation demonstrates significant un-coalesced accesses to memory, reducing the overall throughput.

Vasiliadis *et al.* [99] present a multi-parallel IDS in *MIDeA: a multi-parallel intrusion detection architecture*. The authors implement two versions of the Aho-Corasick algorithm: the AC-Compact and the AC-Full. The compact version uses compact state tables, whereas the full version describes the standard Aho-Corasick algorithm. The AC-Compact version demonstrates 77% throughput of the AC-Full version, however a 37 fold compression improvement. The state table however, demonstrates poor flexibility since in the worst case scenario, the AC-Compact and the AC-Full table require the same amount of memory. The algorithm achieves 5.2 Gbps throughput with zero packet loss running on two nVidia GeForce GTX480 graphics cards.

Lee *et al.* [100] demonstrate how the space required by the Aho-Corasick state tables can be reduced. The authors describe a method to compress the state tables of the finite automaton generated of the AC algorithm by removing the unnecessary failure links generated by the AC algorithm. To limit the depth of the trie they adopt a novel banded-row format and compress the state transition table, using linked lists in order to reduce the overall memory requirements. The authors demonstrate their memory scheme achieving 39.7 percents reduction in memory requirements when selecting 5,000 patterns randomly in comparison with the original version of the AC algorithm and improve by 83.9% the original banded-row format used in the Clam AntiVirus.

Pungila *et al.* [101] propose a compressed Aho-Corasick and Commetz-Walter automaton for GPU-accelerated pattern matching. The automaton constructed by the authors is stored as a one dimensional array, such as described earlier in [96] and [98] . The automaton is created on the host computer and flattened in a depth-first-search fashion. The authors uses array indexes on the host and restore the pointers once the automaton is stored in GPU memory by using a hash table. Their experiments are performed using an Nvidia GTX 560Ti and achieves a throughput of 1420 Mbps, or 33 fold improvement over their counterpart CPU version of the AC-CW algorithm running on an Intel Core i7.

4.3 Highly Efficient-Memory Compression Scheme

The algorithm presented in this thesis is based on the PFAC algorithm, described in Section 3.7. As a brief reminder, the PFAC algorithm solves the boundary problem introduced by allocating chunks to each thread. It also decreases the divergence introduced by the failure links.

The Aho-Corasick is known as being the multi-pattern matching evolution of the Knuth-Morris-Pratt algorithm [20] and has demonstrated superior performance in Section 3.7 when modified and implemented in a 2D state table. The investigation and implementation of this algorithm concluded that it is suitable as the algorithm is sufficiently modular to be improved upon and significantly decreases its memory footprint in order to create a highly efficient memory compression scheme.

The algorithm is implemented as a library with the objective of significantly reducing the memory footprint of the trie, while achieving high throughput. To accomplish this, the core of the PFAC algorithm is modified in order to create a Highly Efficient Parallel Failure-less Aho-Corasick (HEPFAC) algorithm along with a reduced data structure, a highly-efficient storage model, a trie compression algorithm and prefix matching.

4.3.1 Storage Model

The primary component of the highly efficient memory compression scheme is the storage model of the HEPFAC. The abstract memory representation of the HEPFAC trie shown in Figure 4.1 helps understand possible improvements the algorithm can sustain and the way these can be achieved.

In order to achieve an efficient storage model, the patterns added to the trie are sorted alphabetically. As the patterns are arranged in alphabetical order, common prefixes can be identified and merged together. This first step is executed on the CPU. Each of the patterns are then added to the trie in a breadth-first construction model. This model enables the construction of the trie level by level, requiring each letter of the same level to be added first.

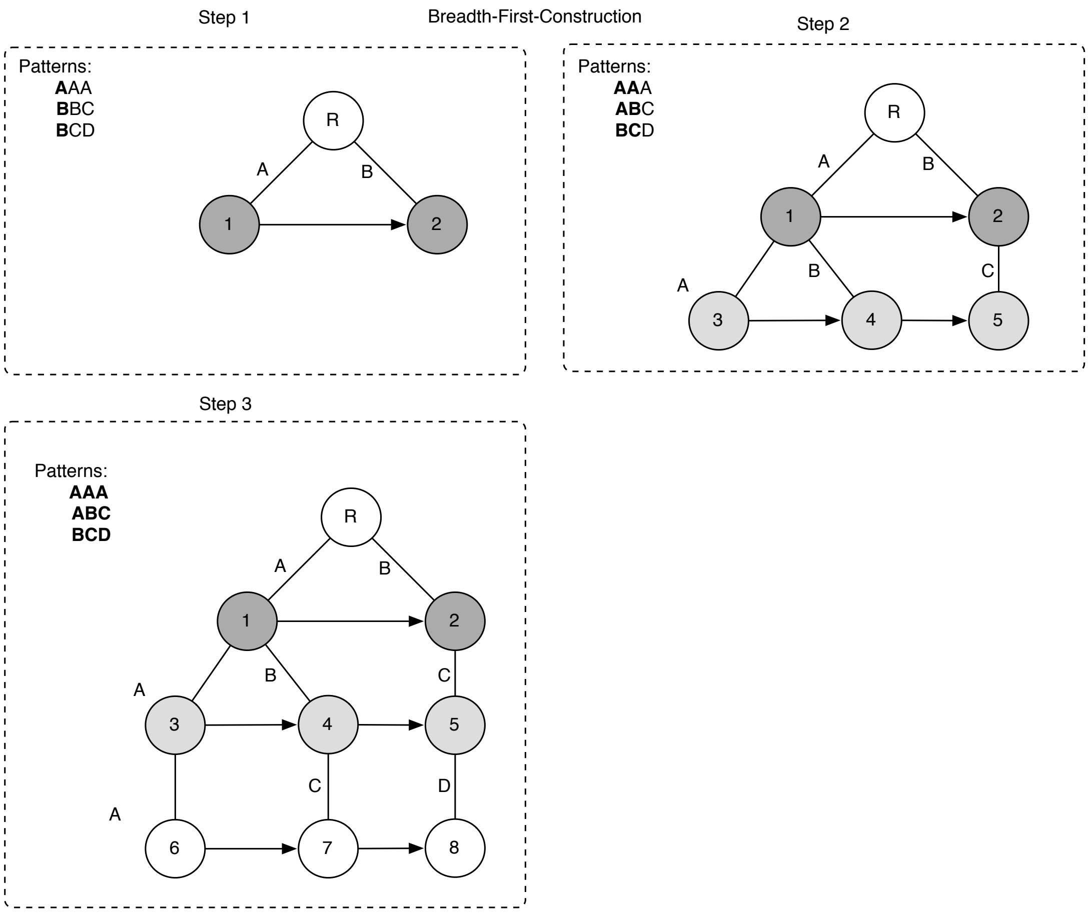


Figure 4.1
Breadth-First-Construction Trie

Figure 4.1 demonstrates the required steps in order to create the trie in a breadth-first fashion. Figure 4.1 (Step I) shows that the first letter of each lexicographically ordered pattern is added to the first level of the trie (Note that *R* denotes the root of the trie), in Figure 4.1 (Step II). The second level of the trie is added by adding sequentially each second letter of each pattern. At this stage the number of levels is based on the longest alphabetically ordered pattern.

The breadth-first created trie is an efficient construction model: it allows to control the number of trie levels required, consequently reducing the overall size of the trie by limiting its depth and by only adding pattern prefixes.

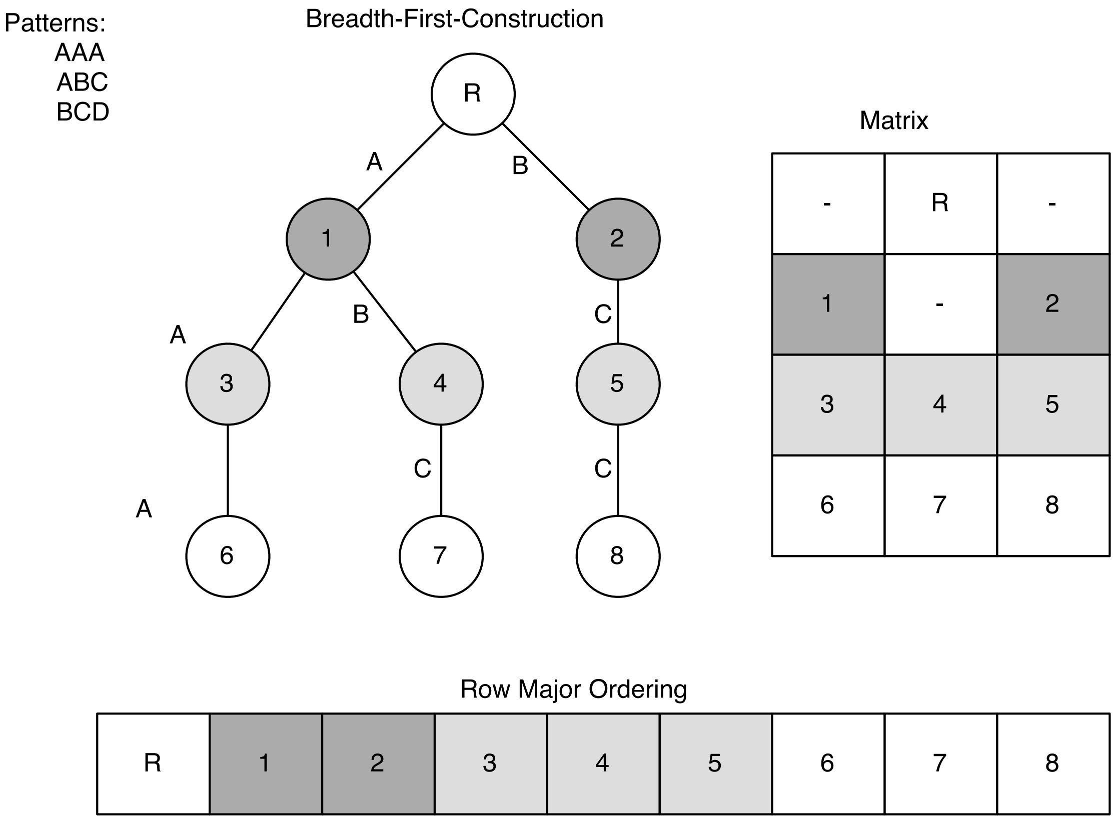


Figure 4.2
Row-Major-Ordering Matrix

Once the trie is constructed, it is represented in memory as a row-major-ordered array [102] [103]. This array aims at removing the need of memory pointers and store the trie in a contiguous block of memory. The removal of pointers is advantageous when working with different types of memory layouts and in order to avoid pointer translation between the GPU and CPU memory [96] [104].

Figure 4.2 shows how the process is achieved. The breadth-first constructed trie can be represented as a matrix; levels are then added sequentially in a row-major ordered array. Note that for simplicity the content of the cell (for the trie transversal) is not represented.

Transposing the trie from the tree representation to a matrix is optional. In order to optimise the process, this step was removed and the trie was constructed directly from the ordered patterns into a 1D breadth-first / row-major ordered array, requiring no pointer translation into indexes and fewer node copies, decreasing the code complexity and the trie construction process time.

During the construction of the trie, each node is assigned with an offset representing his first child. As the trie is ordered in a breadth-first / row major ordering array, the children of each node are consecutively ordered in the array.

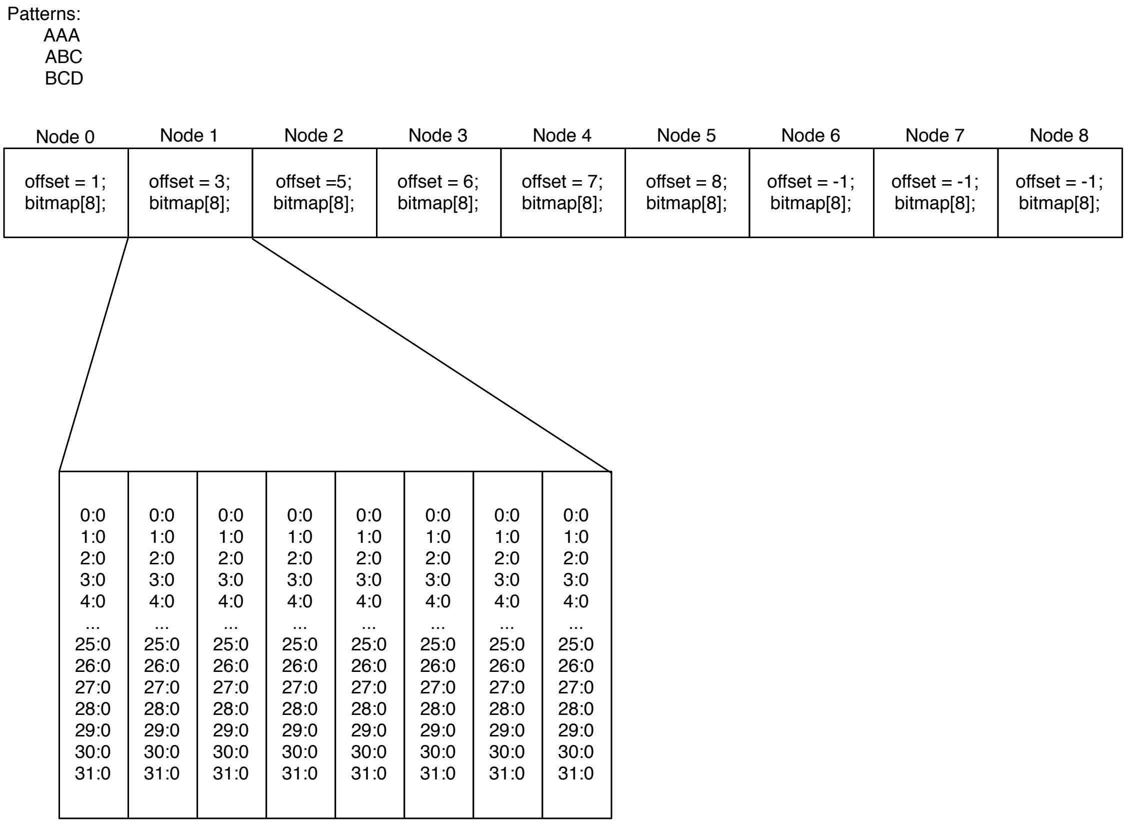


Figure 4.3
Bitmapped Array Trie

Bitmap: Each node of the trie contains a bitmap. A bitmap is a succinct representation of an alphabet using one bit to identify 1 character of the alphabet. For example, DNA only requires an alphabet $\Sigma = 4$, hence requiring 4 bits to map all the characters ('A', 'C', 'G, 'T') of the alphabet.

If a bit is set to 1, it is assumed that the corresponding character is present in the bitmap. If set to 0, it is assumed that the corresponding character is not present in the bitmap. This technique is used to provide compact data storage, allowing succinct data structures, hence reducing memory requirements.

Figure 4.3 shows the content of each node. Each of the trie nodes is composed of an offset pointing towards their first child and a bitmap. An accepting state is represented by the offset being equal to -1. Figure 4.3 shows a bitmap of $8*32$ bits representing the 256 letters of the ASCII alphabet [105]. It is important to note the size of the bitmap can vary with the requirements, (e.g. DNA requires an alphabet Σ =4, hence, reducing the size of the bitmap from 8 to 1).

This technique allows the reduction of the memory footprint of the algorithm. Common algorithms often use an integer array representation of the 256 letters alphabet, where each cell is an integer of 4 bytes, requiring 1024 bytes per cell for the ASCII alphabet. The technique presented in this thesis allows to use 32 bytes to represent the ASCII, demonstrating a memory reduction of 32 fold per nodes [106].

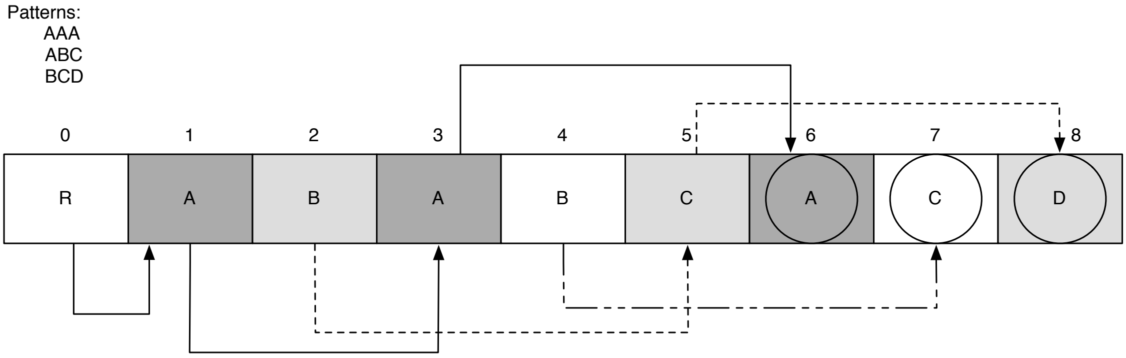


Figure 4.4
First Child Pointer

The offset field in every node represents their first child. In order to find the children of a node, a simple formula based on a population count of the bitmap array is used, as shown in Equation 4.1. The formula operates as follows; let *ChildNode* be the index of the child searched for, *curNode* the current node, *Offset* the index of the first child of the current node and i the population count up to the ASCII value of the child required.

$$ChildNode = curNode \rightarrow offset + i \tag{4.1}$$

In order to find the second child of the root node, or find if the the root node has a child “B”, a population count of the bitmap array is made, counting the number of bits set to 1 before the ASCII value of the letter “B” (ASCII=66).

Figure 4.4 shows that the root node has two children, “A” (Array Position 1) and “B” (Array Position 2). Figure 4.3 shows that “A” is represented as a bit set to 1 at position 65 and “B” is represented as a bit set to 1 at position 66.

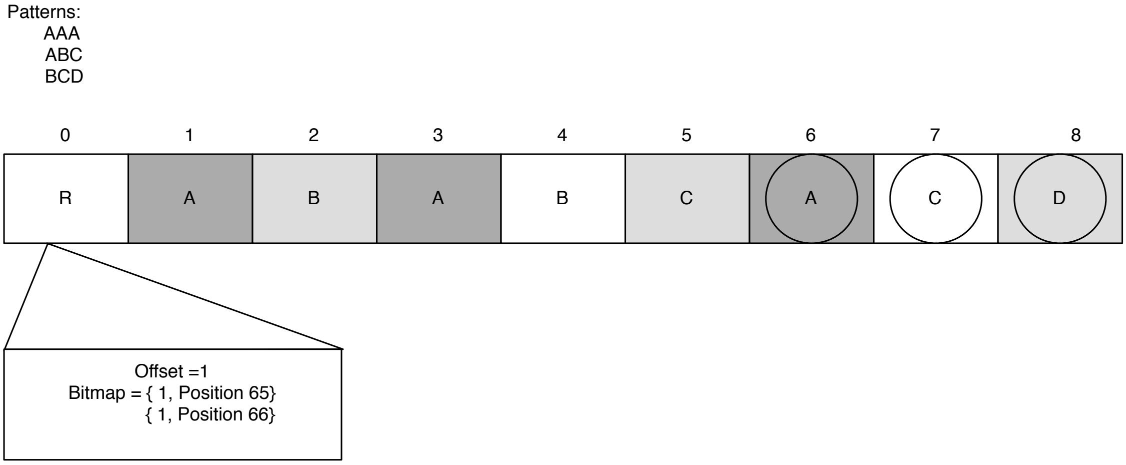


Figure 4.5
First Child of the Root Node

In order to match the pattern $P = \{BCD\}$ the following steps are required:

Step 1 A population check is made on the bitmap of the root node; As the letter “B” is a child of the root node, the ASCII position 66 of the bitmap is set to 1. If the bitmap returned 0 at ASCII position 66 the pattern matching process would have stopped, as the letter “B” would not have been present as a child.

Step 2 To find the position of the node "B" in the array, a population count of the bitmap of the root node is made. The population count is counting the number of bits set to 1 before the ASCII position of the first letter of the pattern "B" (ASCII position = 66) as shown in Figure 4.5. As the root node only possesses two children, the population count returns the value 1, as the bit corresponding to "A" (ASCII position = 65) is set to 1 and is the only bit set before the 66 th bit representing the letter "B".

Step 3 In order to calculate the position of the letter "B" in the array, the result of population count from **step 2** is added to the offset as shown in Equation 4.1. The value returned by the population count is $i = 1$, and the *offset* of the current (root) node is equal to the position of its first child, in this case *offset=1*. Hence the child node ("B") can be found at position 2 in the breadth-first / row major ordered array as shown in Figure 4.5.

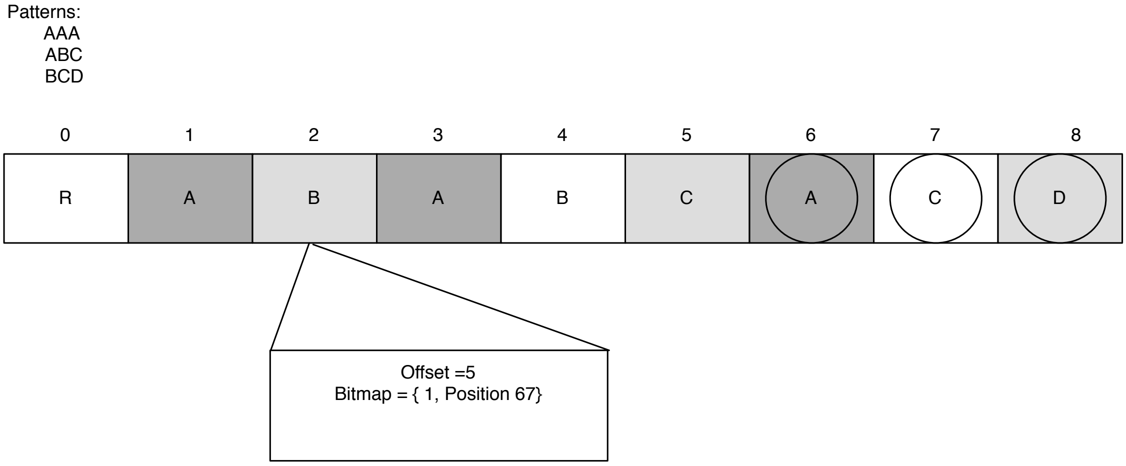


Figure 4.6
First Child of the Letter "B"

Step 4 A population check is made on the bitmap of the new parent node 'B' (Node 2); As the letter "C" is a child of the node 2, the ASCII position 67 of the bitmap is set to 1. If the bitmap returned 0 at ASCII position 67 (Corresponding to the letter "C") the pattern matching process would have stopped, as the letter "C" would not have been present as a child as shown in Figure 4.6.

Step 5 In order to find the second letter ("C") in the pattern "BCD", a population count of the parent bitmap node (letter "B") is made . The population count is counting the number of bits set to 1 before the ASCII position of the second letter of the pattern ("C" $ASCII = 67$). As "B" has no other children, the population count returns 0.

Step 6 In order to calculate the position of the letter "C" in the array, Equation 4.1 is used. The value returned by the population count is $i = 0$, and the *offset* of the current ("B") node is equal to the position of its first child, in this case *offset=5* as shown in Figure 4.6 . Hence the child node ("C") can be found at position 5 in the breadth-first / row major ordered array as shown in Figure 4.4.

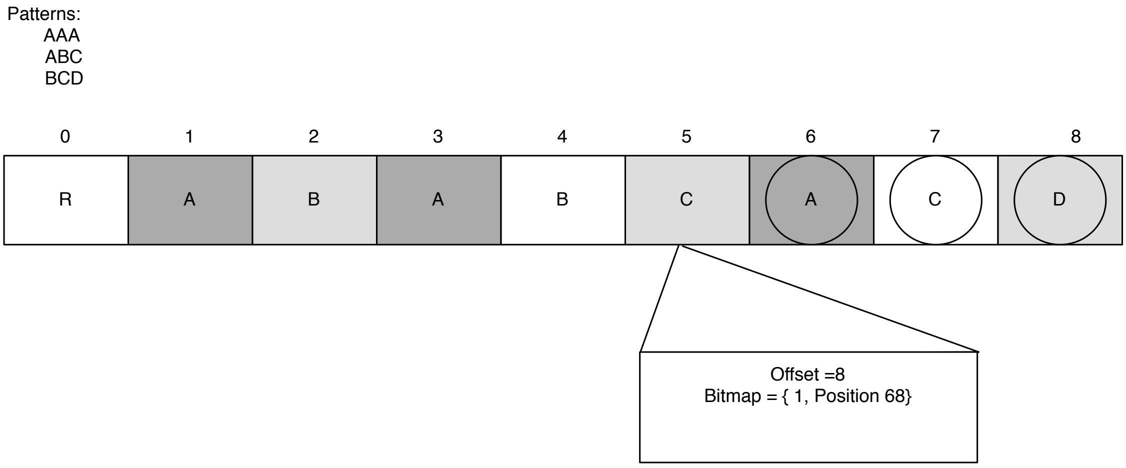


Figure 4.7
First Child of the Letter "C"

Step 7 In order to find the third letter ("D") of the pattern "BCD", a population count of the bitmap node corresponding to the letter "C" is made. The population count is counting the number of bits set to 1 before the ASCII position of the third letter of the pattern ("D" $ASCII = 68$) as shown in Figure 4.7. As "C" has no other children, the population count returns 0.

Step 8 In order to calculate the position of the letter "D" in the array, Equation 4.1 is used. The value returned by the population count is $i = 0$, and the *offset* of the current ("C") node is equal to the position of its first child, in this case *offset=8*. Hence the child node ("D") can be found at position 8 in the breadth-first / row major ordered array as shown in Figure 4.7.

These steps demonstrate how patterns are identified in the trie and are undertaken by every single thread. When the child of a given node is not present at the corresponding ASCII position (e.g. "A" $ASCII = 65$) in the bitmap, the population count will be equal to 0. The current thread will then be terminated whilst other threads will continue their matching process independently.

This memory representation of the trie can be further optimised to reduce the memory footprint by truncating the trie to a depth of 8th level, and by merging common pattern suffixes together. These techniques are discussed further in Section 4.3.2.

4.3.2 Trie Compression

The trie compression is the key part of the memory footprint reduction. It helps by reduce the number of nodes and eliminating redundant pattern suffixes. With the high number of patterns and the exponential increase of data, the number of nodes in the trie can become substantial, increasing drastically the memory footprint of the trie and making trie transfer prohibitive.

The trie reduction is alphabet dependent and demonstrates a strong correlation between similar suffixes and large alphabets [107] [108]. The proposed reduction technique allows reduction of the number of nodes on the last three levels of the trie, making use of unique branches and removing redundant paths. This process happens in four steps. Note that for convenience this example uses a visual representation of the trie, hence is demonstrated in a tree like structure rather than an array, in order to demonstrate the steps undertaken by the compression process and the effects on the trie branches, but it is equally applicable to a 1D memory representation of the trie.

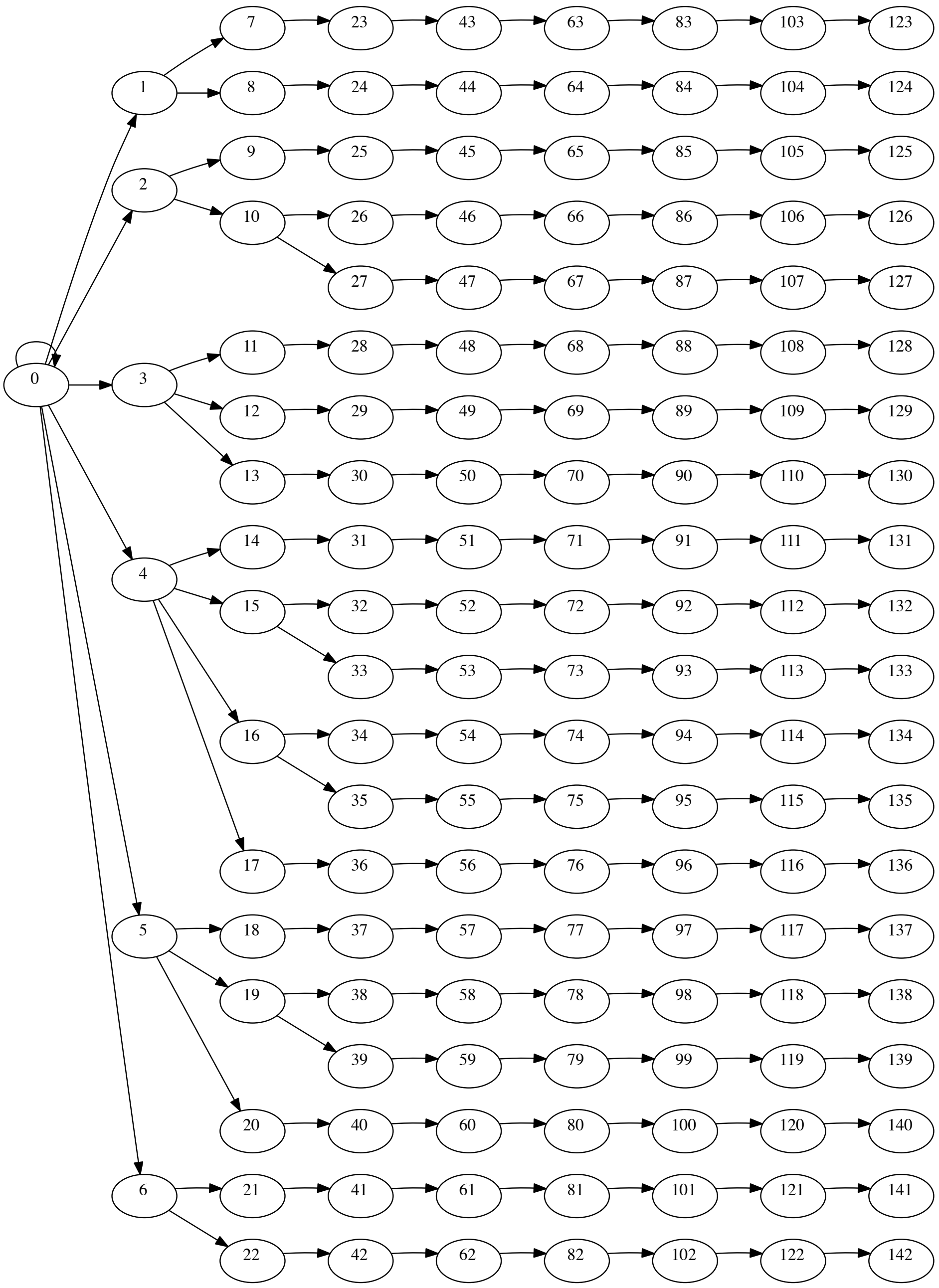


Figure 4.8
Full Truncated Trie

Step 1 The first step consists of truncating the trie and reducing the total length of the patterns. Figure 4.8 shows a trie of 20 Patterns truncated to a length of 8 after the root node. The constrained length allows each thread to perform the same amount of work when a pattern is fully matched, reducing thread divergence. A similar technique has previously been implemented by Vasiliadis *et al.* [76], on a different multi-pattern matching algorithm, achieving a false positive rate below 0.0001% with a trie depth of 8. This work verifies the technique independently for the HEPFAC trie presented in this thesis in Section 4.5 and evaluates it against different alphabet sizes in order to allow better performances against different types of alphabets, allowing applicability in the medical, security, computer science and other domains using custom alphabets sizes.

Step 2 The second step consists of eliminating every last node of every pattern and merging them into one last node. This reduction can be calculated as follows; let Q_{final} be the total number of nodes after merging the last nodes together, Q be the total number of nodes before merging and P be the total number of patterns,

$$Q - (\mid P \mid -1) = Q_{final} \tag{4.2}$$

Equation 4.2 demonstrates the expression that can be used to calculate the number of nodes after the second step of the compression. As the parents are able to identify their children, the last node of each pattern is unnecessary and therefore discarded and merged into one shared final node.

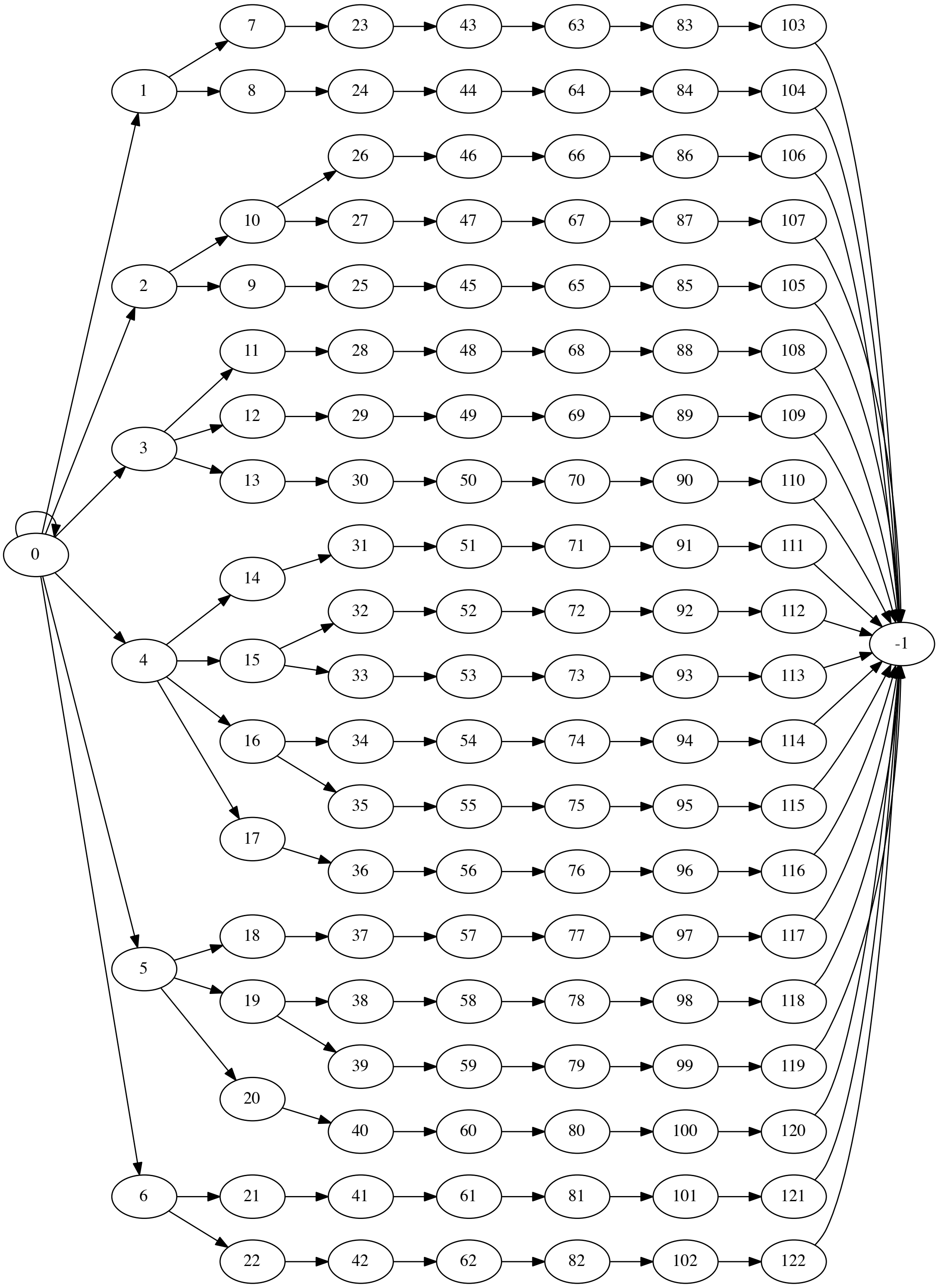


Figure 4.9
Last Node Reduction Step

Figure 4.9 demonstrates that last node compression does not affect the branches of the trie as the last node is only created to acknowledge a full match. In fact the offset of the last node can be set to -1, allowing the threads to register the full match while avoiding them keeping track of the trie levels.

Step 3 The third step is the most important regarding the trie compression, as it reduces the number of branches in the trie. The compression is applied on the three last levels of the trie, merging similar suffixes. This step is achieved by comparing each of the branches against their neighbouring tree-node suffixes and merging similar suffixes.

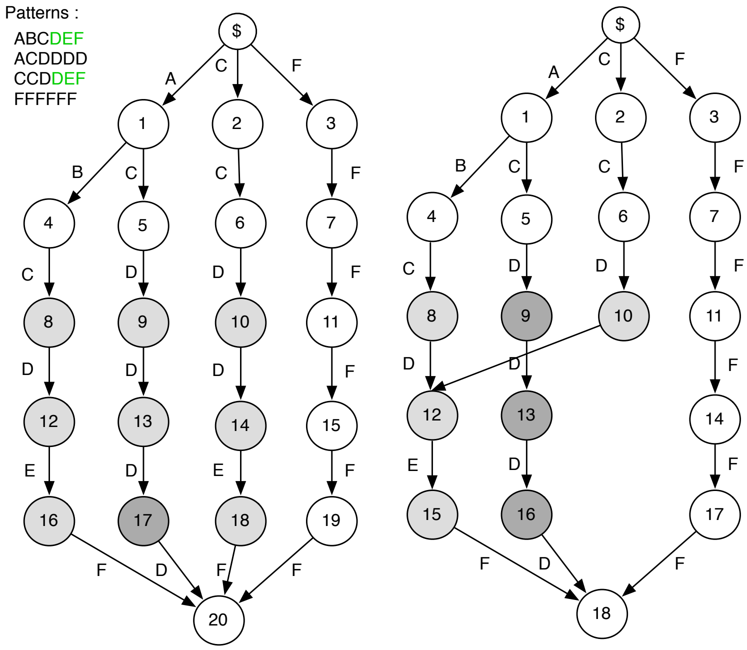


Figure 4.10
Branches with Similar Suffixes are Merged Together

Figure 4.10 provides an example of branch reduction for the set of pattern $X = \{ABCDEF\}, \{ACDDDD\}, \{CCDDEF\}, \{FFFFFF\}$. The patterns suffixes ending by DEF are merged together reducing the overall number of nodes required to store the trie in memory.

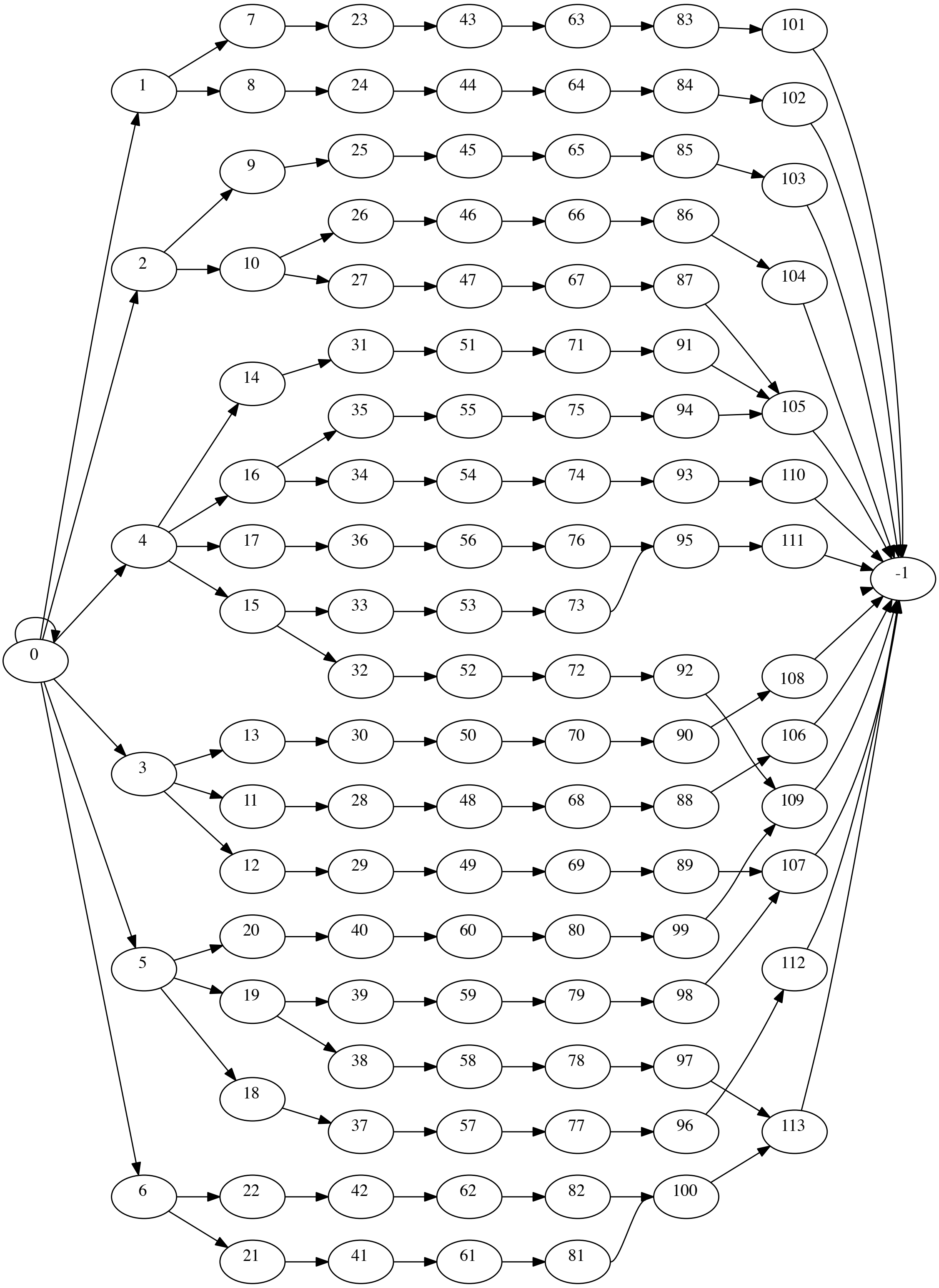


Figure 4.11
Reduced Trie

Figure 4.11 shows the reduced trie. Different branches have been merged at different levels drastically reducing the number of nodes required to store the initial trie containing 142 nodes, against 114 nodes for the reduced trie, demonstrating a total reduction of 20%.

These reduction and compression steps are achieved on the host, before sending the reduced trie to the GPU along with the text strings. The process of sending and matching patterns on the GPU is described subsequently.

4.4 Processing-Engine

The text strings are forwarded to the processing engine for analysis. The purpose of the processing engine is to match the text sequences against the patterns stored in global memory and transfer the results back to the GPU.

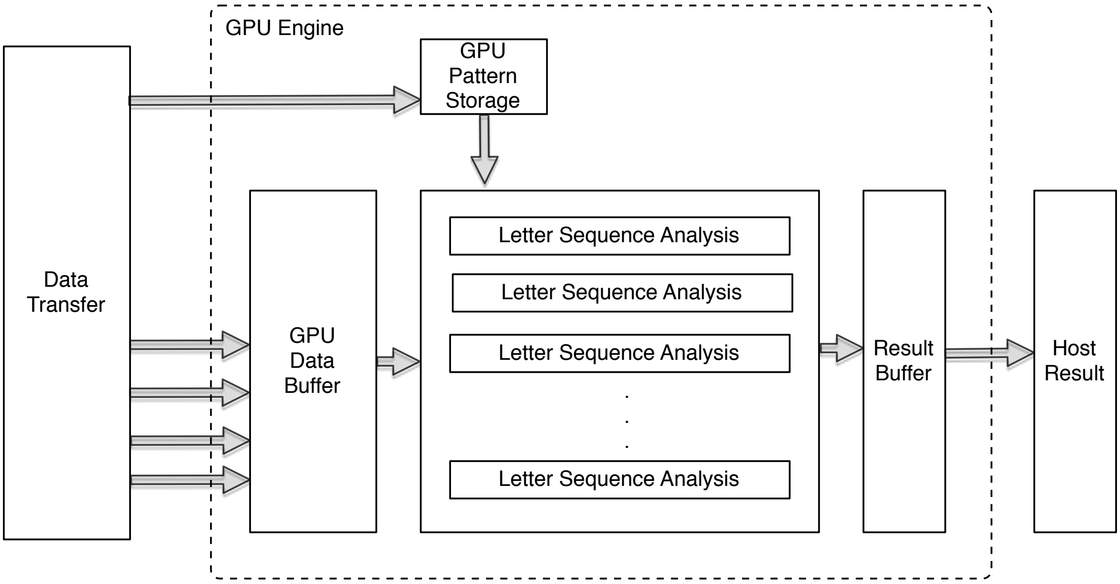


Figure 4.12
Overview of the GPU Processing Engine.

Figure 4.12 demonstrates the steps that occur between the data transfer from the host to the GPU and from the GPU to the host. The GPU engine also illustrates the matching technique used, where the text string is stored in global memory in a data buffer. Each letters of the text string are subsequently matched against the trie as shown previously in Figure 3.20.

Small data files are aggregated together in a single buffer for analysis, as the PCIe throughput degrades with small data transfers from the host to the GPU due to the overhead added by the buffering and transfer; this has been observed by different researchers [79] [109].

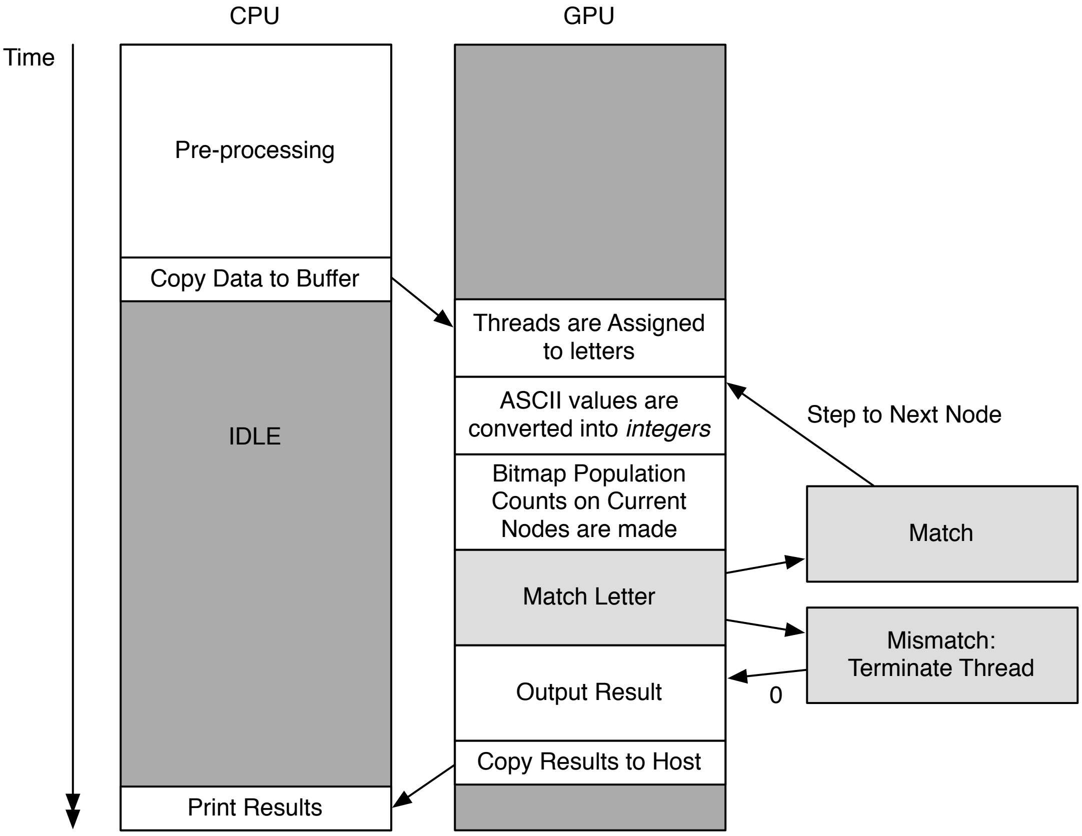

CPU
GPU
Time
Pre-processing
Copy Data to Buffer
IDLE
Print Results
Threads are Assigned to letters
ASCII values are converted into *integers*
Bitmap Population Counts on Current Nodes are made
Match Letter
Output Result
Copy Results to Host
Step to Next Node
Match
Mismatch: Terminate Thread
0

Figure 4.13
Overview of the Execution Flow.

Figure 4.13 shows the overall execution flow adopted by the HEPFAC algorithm over time. This flow highlights the idle state of the CPU during the matching process, which inherently allows the CPU to concentrate on other tasks while the GPU matches the text string against the different patterns. The matching process operated on the GPU follows the steps described in Section 4.3.1.

4.4.1 Performance Optimisation

Having detailed the overall architecture of the HEPFAC algorithm, a number of optimisations improving the overall computations are described. The core idea behind the optimisation techniques introduced is to increase the processing throughput of the algorithm.

The fist optimisation applied to the HEPFAC algorithm is the implementation of grid level concurrency. This is achieved by creating multiple execution streams. CUDA streams refers to a sequence of operations that can be conducted serially or in a concurrent fashion. This technique allows to overlap different types of operation in an asynchronous fashion.

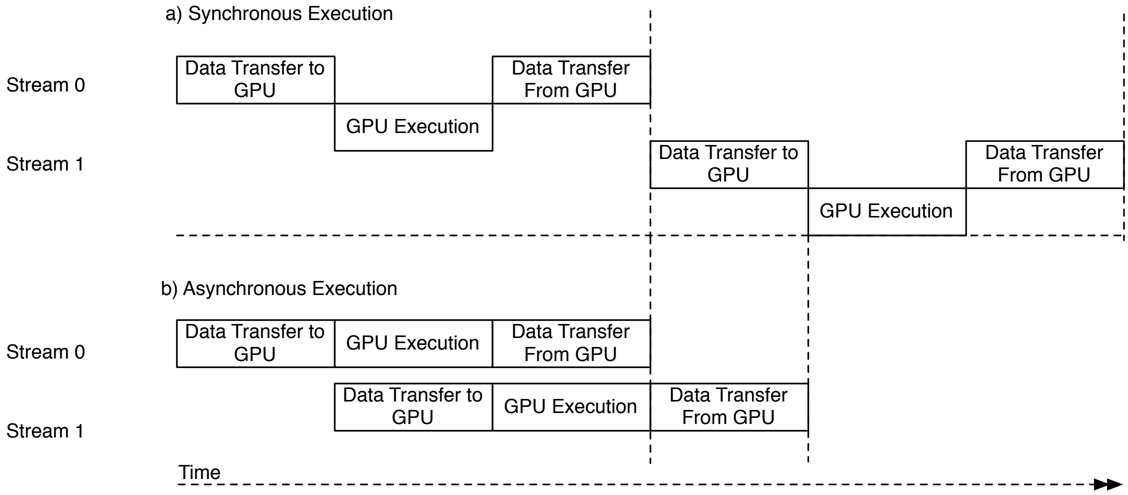


Figure 4.14
Data Transfer Using Asynchronous Execution on the GPU in Order to Overlap Execution Streams.

Figure 4.14 (a) (Synchronous Execution) demonstrates a simple timeline of the different operations performed by the algorithm serially, while Figure 4.14 (b) Asynchronous Execution) demonstrates kernel computations evenly distributed amongst the two concurrent streams. Due to common hardware and the shared resources the execution of the two different streams are serialised and the data transfer can only be overlapped when used with a duplex PCIe bus. Note that the number of concurrent streams is limited by the available resources on the GPU, such as the register and shared memory [110].

In order to increase the performance of the processing engine and reduce thread divergence, the matching process uses a loop unrolling technique [111]. This technique allows to increase the work per thread, reduces the instruction counts and increases the branch prediction resulting in higher throughput [112] [113]. This process increases total code space, however the trade-off between code space and speed is significant in this case, this technique also increases the register usage, but once again the trade-off is significant and the overall matching throughput increases significantly [114].

4.5 Performance Evaluation

The performance evaluation of the proposed HEPFAC algorithm is made as follows: Firstly the evaluation of the prefix matching analysis is realised. Secondly the memory requirements for the tries are realised, comparing with state of the art research. Thirdly the overall throughput of the engine architecture is analysed including the data transfer and trie creation. Finally the performance measurements of the matching engine are presented.

The performances of the algorithm are assessed by using synthetic files generated by the Mersene Twister (MT) uniform pseudo-random number generator [78] as it is the most widely used. Each evaluation is performed 100 times against 5 different synthetic files representing different alphabet sizes, demonstrating the potential of the algorithm in different scientific fields and highlighting particular characteristics such as throughput, prefix matching sizes and compression.

Furthermore the use of synthetic files allows the creation of a control environment, in order to perform advanced testing. It is acknowledged that real-world traces allow a realistic representation of the sustained throughput, however, synthetic traces have the advantage of highlighting and isolating specific behaviours which are often not well represented in real-world traces [115] [116]. The different characteristics of the traces have been varied as much as possible in order to cover the different aspects required by the evaluation and to demonstrate the performances sustained in different types of environment. Such testing is often used by researchers and is a common and recommended practice throughout academia and industry [117] [118] [119].

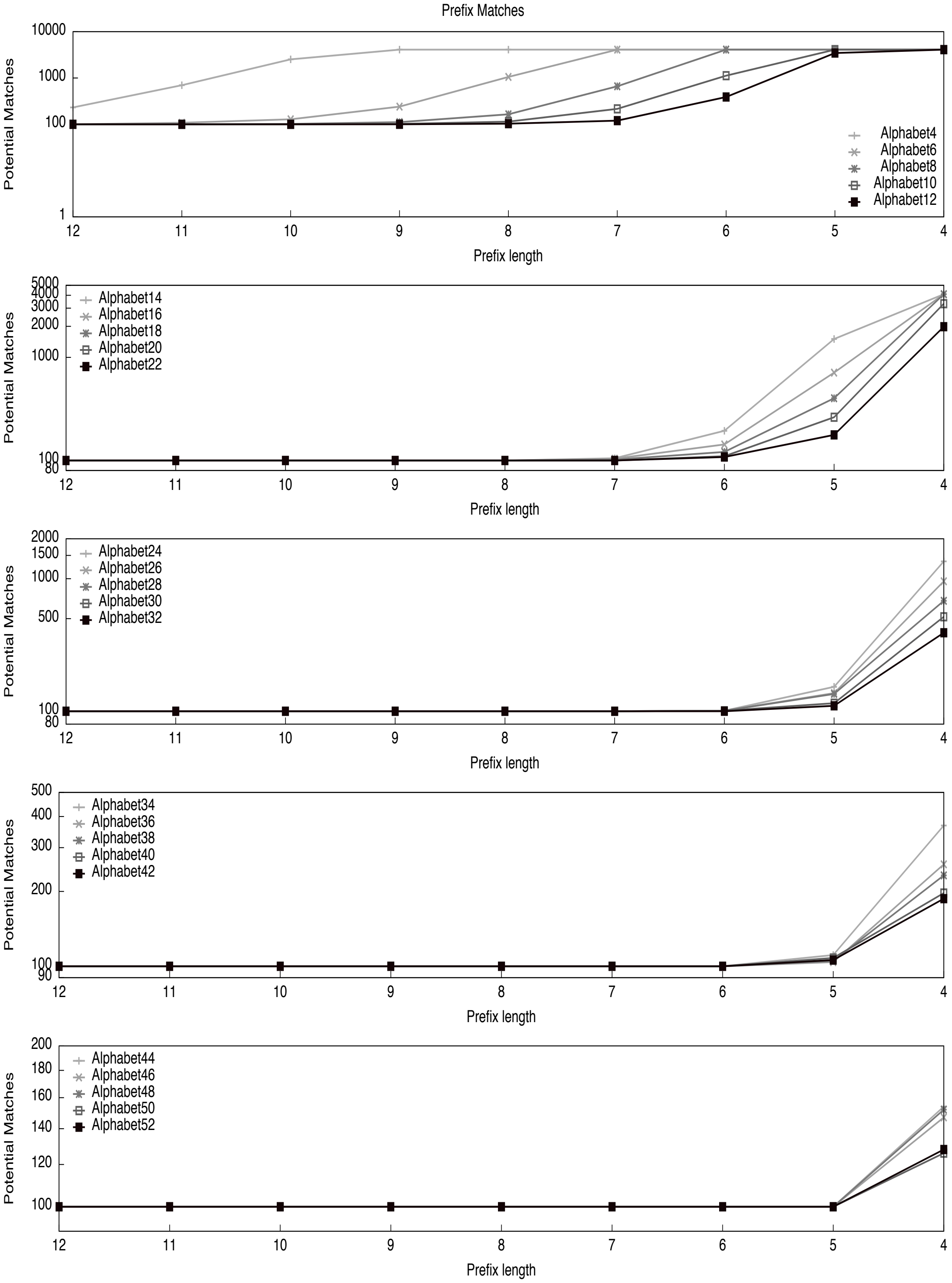


Figure 4.15
Comparison of Full Pattern Matches Based on their Prefix Sizes and Alphabet Used

4.5.1 Prefix Size Matches

The technique of prefix matching is alphabet dependent [120]. The probability of a prefix being identified as unique depends on the size of the alphabet. For example small alphabets decrease the total number of combinations of letters and unique patterns require more effort to be identified. Conversely larger alphabets require more combinations, requiring fewer computations to uniquely identify a pattern [121]. This is demonstrated in Figure 4.15. The top figure shows that, a trie depth of over 12 characters is required to uniquely identify a pattern of $\Sigma = 4$, whereas patterns of alphabets $\Sigma = 8$ and $\Sigma = 10$ require at least a depth of 9 to be uniquely identified.

The trend is verified in Figure 4.15 (2, 3 and 4). Figure 4.15 (5) shows that alphabets over $\Sigma = 44$ only require 5 characters to be uniquely identified. Applying this technique to the HEPFAC allows the reduction of the overall trie size while eliminating numbers of false positive matches. It is assumed that further manual or automatic matching is implemented on the host in order to confirm the full match, while offloading the heavy pattern matching to the GPU.

4.5.2 Memory Requirements

In Sections 3.5, 3.6 and 3.7, different algorithms were presented and their memory requirement were highlighted, demonstrating high memory footprints. However in Section 4.3.2 a trie compression algorithm was presented on a reduced bitmapped trie leading towards reducing memory footprint to a minimum. In this section, the reduced bitmapped trie is analysed against different trie implementations and compared against state of the art implementations by other researchers.

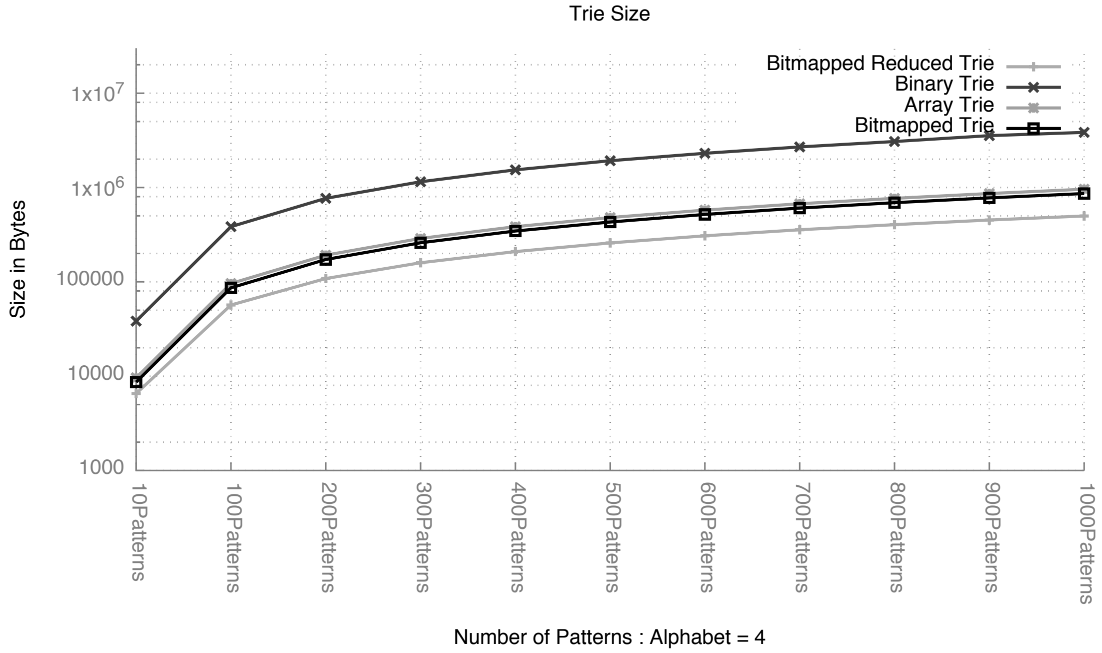


Figure 4.16
Trie Size Comparisons Without Node Reduction for Small Alphabet

Figure 4.16 shows the performances of the bitmapped reduce trie over an alphabet of $\Sigma = 4$. It shows how the HEPFAC algorithm scales over different number of patterns. The truncated patterns used for this experiment were 20 characters long. The reduction achieved by the bitmapped reduced trie demonstrates an average size reduction improvement of 38% compared to the simple bitmapped trie. (Note the logarithmic scale of the Y axis).

As shown, the binary trie demonstrates a higher memory footprint than the bitmapped array trie, even though the data structure of the binary trie is smaller. This is due to the explosion of the number of nodes required in order to store every pattern leading to high memory requirements. The array trie demonstrates similar properties as each characters of the ASCII alphabet is represented by 4 bytes.

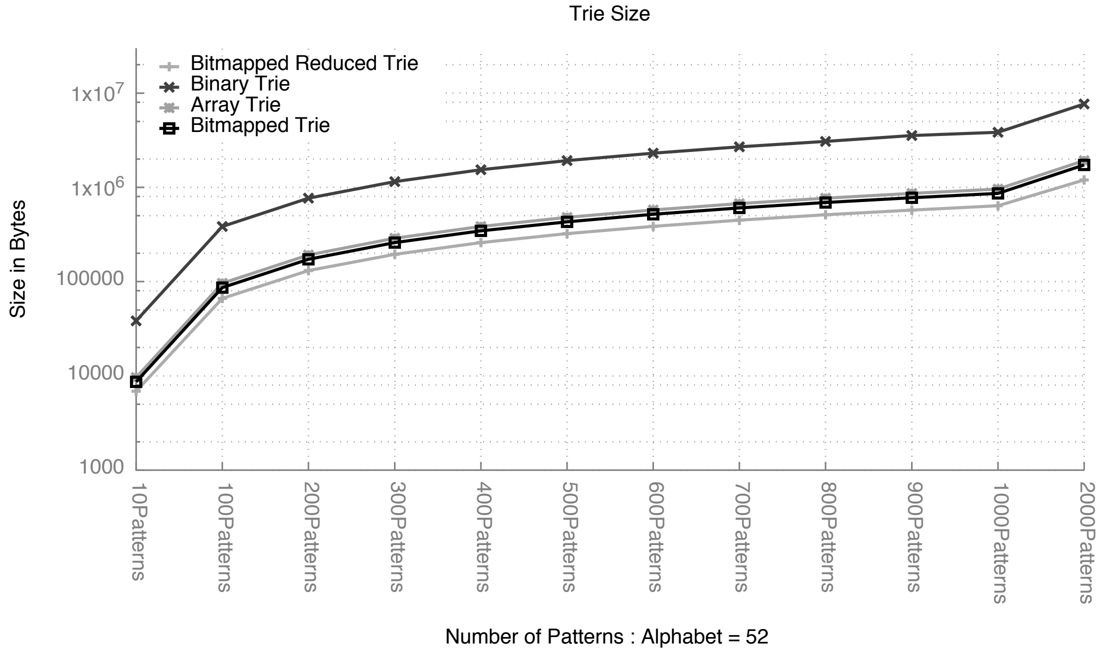


Figure 4.17
Trie Size Comparisons Without Node Reduction for Large Alphabet

Figure 4.17 depicts similar properties to Figure 4.16. However, the trie demonstrates a smaller gap between the bitmapped trie and the bitmapped reduced tries. This phenomenon is due to lower suffix reduction rates.

As the alphabet size has increased to $\Sigma = 52$ the number of similar branches decreases, leading to a smaller number of branches being merged, hence reducing the gap between both implementations. Despite the increasing number of unique suffixes the trie still demonstrates a memory foot print reduction of 16% against its bitmapped trie counterpart.

Note that the array trie and the binary trie displayed in Figure 4.17 are not affected by the increase of the alphabet size as they are not alphabet dependent and therefore the average memory footprint demonstrated by these implementations is similar to the one presented in Figure 4.16.

Table 4.1
Memory Comparison between different state of the art implementations.

Alphabet Sizes	Nodes	HEPFAC	PFAC [57]	Pungila *et al.* [101]	GrAVity [76]
$\Sigma = 4$	1000,000	5 MB	12 MB	7 MB	1000 MB
$\Sigma = 32$	1,703,023	12.9 MB	24.18 MB	15.02 MB	1703 MB
$\Sigma = 256$	352,921	12.1 MB	15.18 MB	13.41 MB	345 MB

The bitmap can also be adapted to the size of the alphabet and occupy Σ bits. Table 4.1 shows that given alphabets of Σ = 4, 32 and 256 and a corresponding bitmap size, the HEPFAC described in this thesis demonstrates the best compression. It is of importance to note that the results depicted in Table 4.1 do not include the HEPFAC trie compression step, as this step is data dependent. Instead the comparison is solely made against the node sizes of the state of the art approaches.

Table 4.1 shows that the PFAC implementation requires a node data structure of 15 bytes, requiring a total of 24.18 MB over 1,703,023 nodes. It also demonstrates that the approach described by Pungila et al. [101] using Aho-Corasick and the Commentz-Walter (AC-CW) algorithm requires 10 bytes per node for a total of 15.02 MB over the same number of trie nodes. The HEPFAC approach described in Section requires 1.87 times less memory than PFAC and 1.16 times less memory than AC-CW approach. Furthermore, the approach proposed by Vasiliadis *et al.* [76] in GrAVity requires 345 MB for an alphabet of Σ = 256, for total of 352,921 nodes whereas our approach without the trie compression only requires 12.1 MB, demonstrating 28.5 fold memory savings.

It is also possible to approximately infer the memory requirements of the different algorithms proposed in [57], [101] and [76] for different alphabets as described [101] in order to draw comparisons and demonstrate that the HEPFAC implementation presented in this thesis is the most memory efficient algorithm when used with the same number of nodes.

4.5.3 Sustained Throughput

In this section the throughput of the matching algorithm is evaluated against different types of alphabet sizes and against different number of patterns.

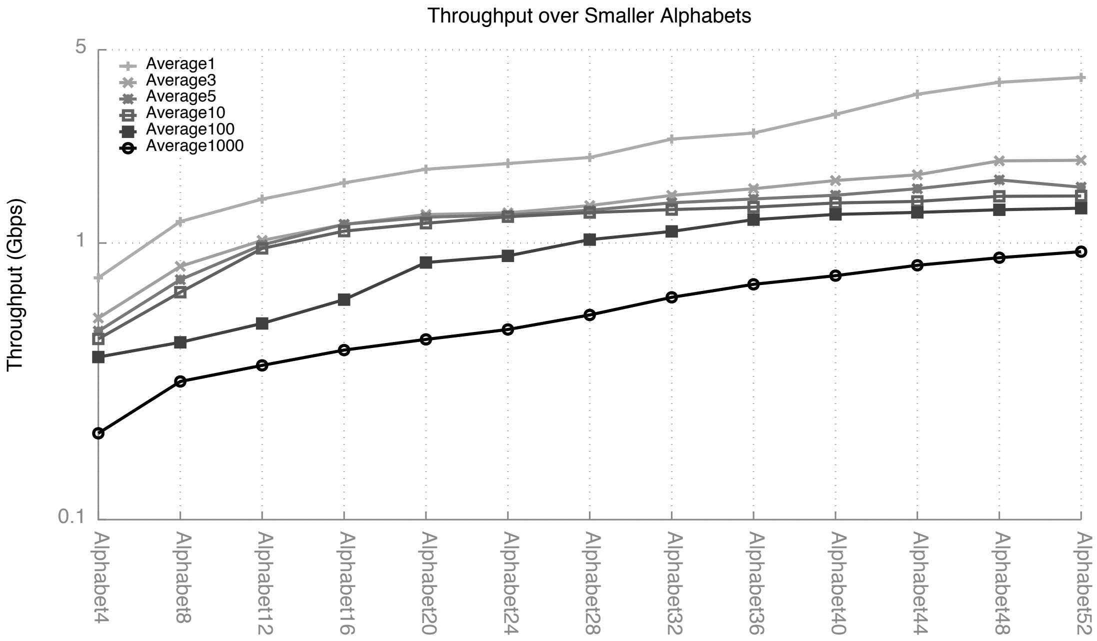


Figure 4.18
Sustained Throughput for Small Alphabet Sizes Versus the Number of Patterns

Figure 4.18 shows the average speed of the HEPFAC algorithm for small alphabet sizes. It is demonstrated that larger alphabet sizes perform better than smaller alphabet sizes. The throughput is also affected by the number of patterns being matched by the engine. When matching only 1 pattern with an alphabet of $\Sigma = 8$ the algorithm demonstrates over 1 Gbps of throughput, however, when matching only one pattern with an alphabet of $\Sigma = 52$ the algorithm demonstrates a throughput of 4.5 Gbps, whereas when matching 1000 patterns with an alphabet of $\Sigma = 52$ the algorithm demonstrates an average of 1 Gbps throughput. The throughput increase is due to the amount of threads facing early termination during a mismatch as the alphabet sizes grows. To conclude, the throughput increases with the alphabet size.

Figure 4.18 also displays interesting properties regarding the throughput over small alphabets when matching a small amount of patterns. For 3, 5, 10, and 100 patterns the throughput sustained by the algorithm follows the same trend and displays similar properties. This is due to the small number of patterns and the reduced divergence involved during the matching.

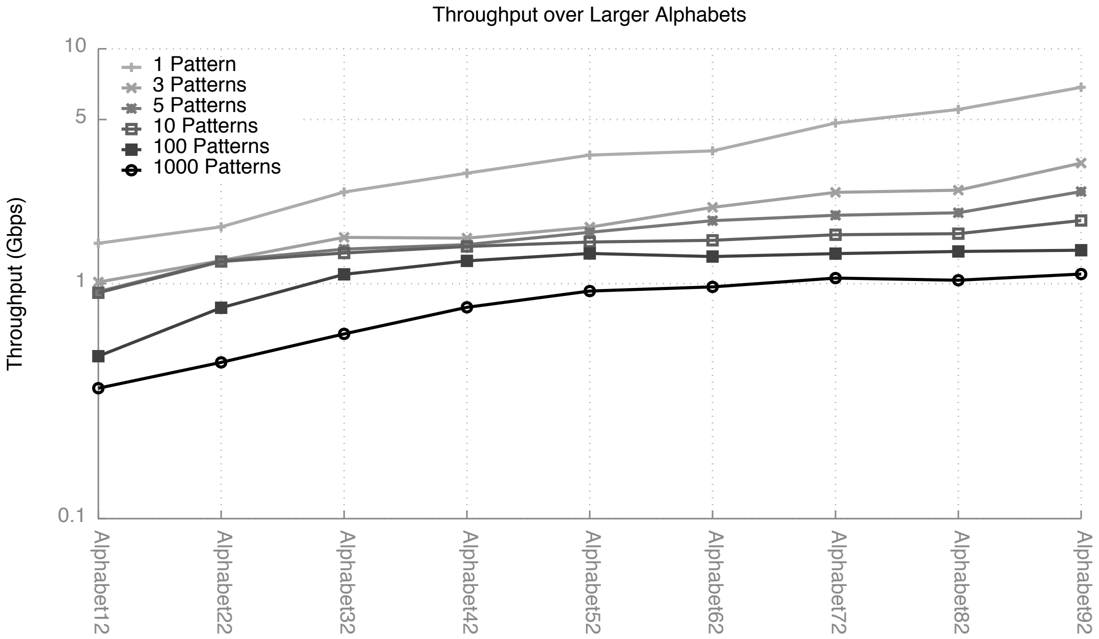


Figure 4.19
Sustained Throughput for Large Alphabet Sizes Versus the Number of Patterns

Figure 4.19 shows the throughput attained by the matching engine over larger alphabets. When using an alphabet of $\Sigma = 92$ matching one single pattern the throughput reaches 8.5 Gbps, however when matching 1000 patterns the throughput reaches 1 Gbps. A similar trend was expressed in Figure 4.18.

Figure 4.19 displays another property of the HEPFAC algorithm. When using large alphabets and small number of patterns (over $\Sigma = 52$), the throughput increases with the smaller number of patterns, whereas Figure 4.18 demonstrated similar properties. This effect is due to the increasing number of character combinations, decreasing the thread divergence and the depth a single thread has to travel before being discarded.

4.5.4 Overall Timing Breakdown

The overall throughput of the HEPFAC algorithm is quantified by aggregating different measurements together. These time measurements correspond to different aspects of the algorithm, such as the array construction, the trie reduction, the transfer from the host to the device and the searching phase.

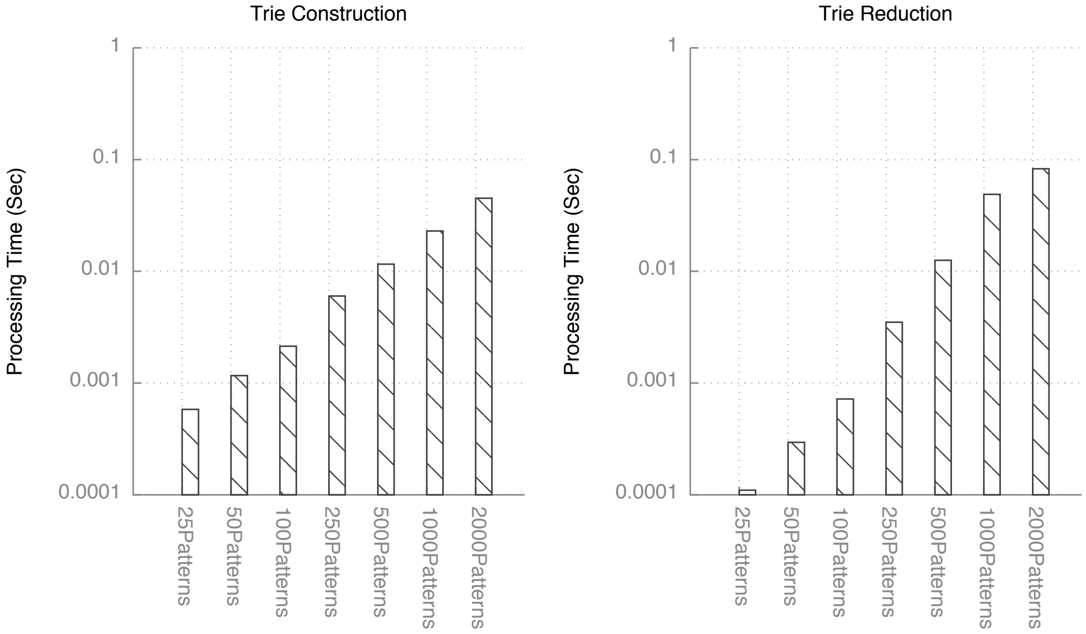


Figure 4.20
CPU Processing Time During the Construction and Reduction Phase

Figure 4.20 (Left) shows the processing time taken in seconds by the construction of the trie, over different numbers of patterns, with a trie depth of 8 levels. Figure 4.20 (Right) highlights the processing time of the trie reduction, over the same amount of patterns, with a similar trie depth. The trie construction and reduction time increases when the number of patterns increases. This is an expected behaviour, as more processing is required to process the patterns during the construction and the reduction of the trie, with the increasing number of patterns.

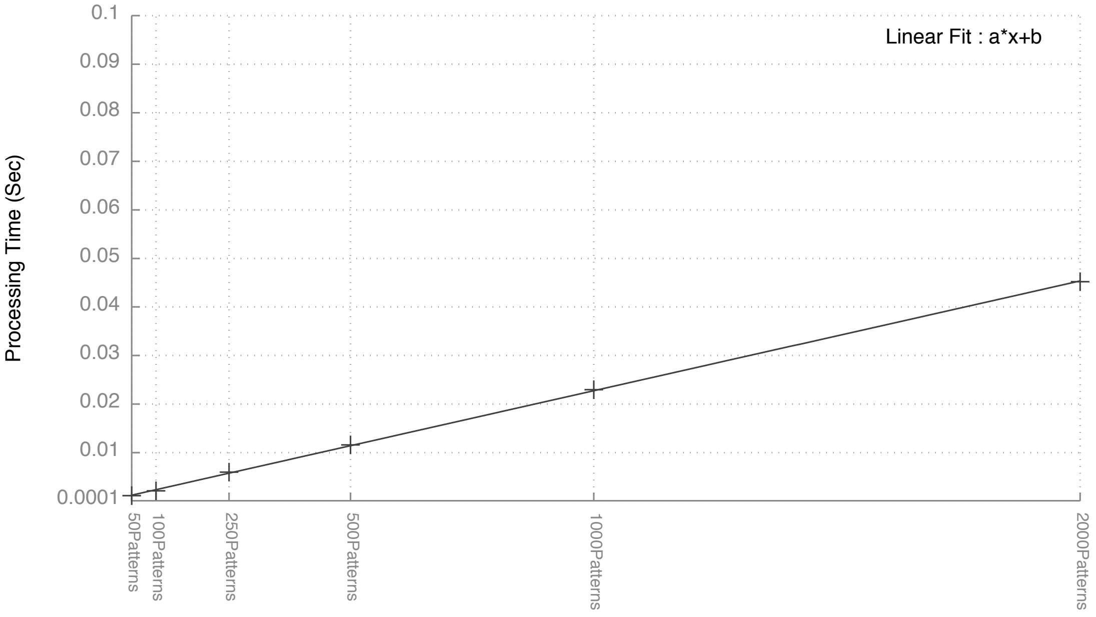


Figure 4.21
Linear Fit for the Trie Construction

The construction process is demonstrated to be linear as shown in Figure 4.21. That is, the processing time increases linearly as the number of patterns increases. The best fit can be calculated as follows. Let a and b be the constant parameters and x the number of patterns. Then, the relationship between the number of patterns and the processing time can be described according to Equation 4.3.

$$f(x) = a \cdot x + b \tag{4.3}$$

In order to fit the data points from the trie reduction algorithm the constant parameters of the linear fit demonstrated in Equation 4.3 can be set as follows, $a =$ 0.0000226253 and $b =$ 0.000122627, where a represents the processing time per pattern, and b the fixed overhead time. The coefficient of determination is $R^2 =$ 0.999854 and demonstrate that the linear regression model has an accuracy over 99%.

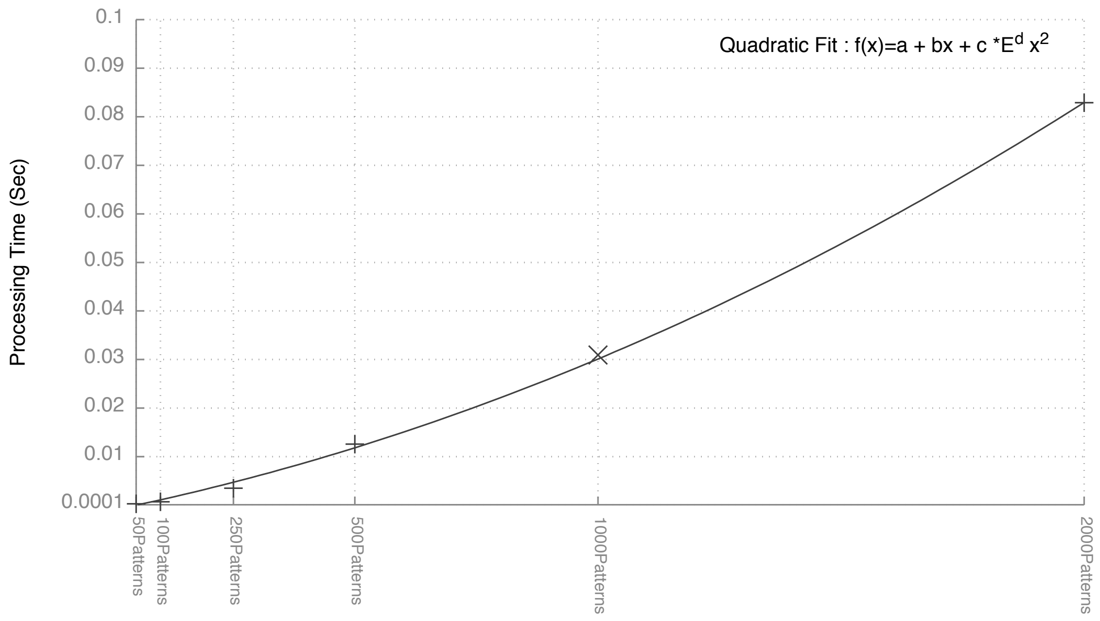


Figure 4.22
Quadratic Fit for the Trie Reduction

The reduction process follows a quadratic polynomial regression, as demonstrated in Figure 4.22. The quadratic model demonstrates one of the limitations of the reduction algorithm. That is, the time increases in a quadratic fashion, with the increasing number of patterns. However, as demonstrated in Figure 4.20 (Right) and Figure 4.22, the time requirement for the trie reduction is low and requires less than a tenth of a second to reduce over 2000 patterns.The best fit can be calculated as follows. Let a, b, c and d be the constant parameters and x the number of patterns. The relationship between the number of patterns and the reduction time can be described according to Equation 4.4.

$$f(x) = a + b \cdot x + c \cdot exp(d) \cdot x^2 \tag{4.4}$$

Equation 4.4 demonstrates the quadratic fit, with the following constant parameters $a = 0.00102858$, $b = 0.0000203226$, $c = 1.0828241444881265$ and $d = -8$. The coefficient of determination is $R^2 = 0.999507$ and demonstrates an accuracy of the quadratic fit over 99%.

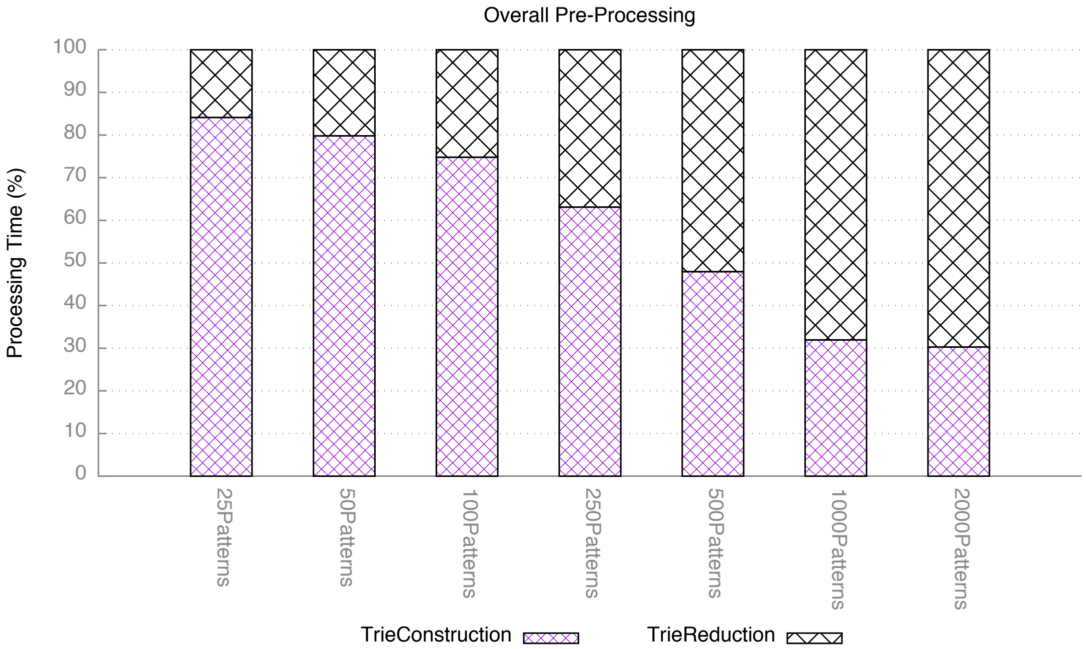


Figure 4.23
Overall CPU Preprocessing Breakdown

The overall pre-processing time is combined and analysed in Figure 4.23. With the increasing number of patterns, the time requirements for the trie reduction overtake that of the trie construction. This behaviour is expected, as previously described in Figure 4.21 and Figure 4.22.

As the reduction phase of the algorithm is required to analyse and merge similar branches together, the time requirement increases. This behaviour is due to the number of iterations required at each level during the branch merging process.This particular behaviour of the algorithm is explicitly highlighted in Figure 4.23.

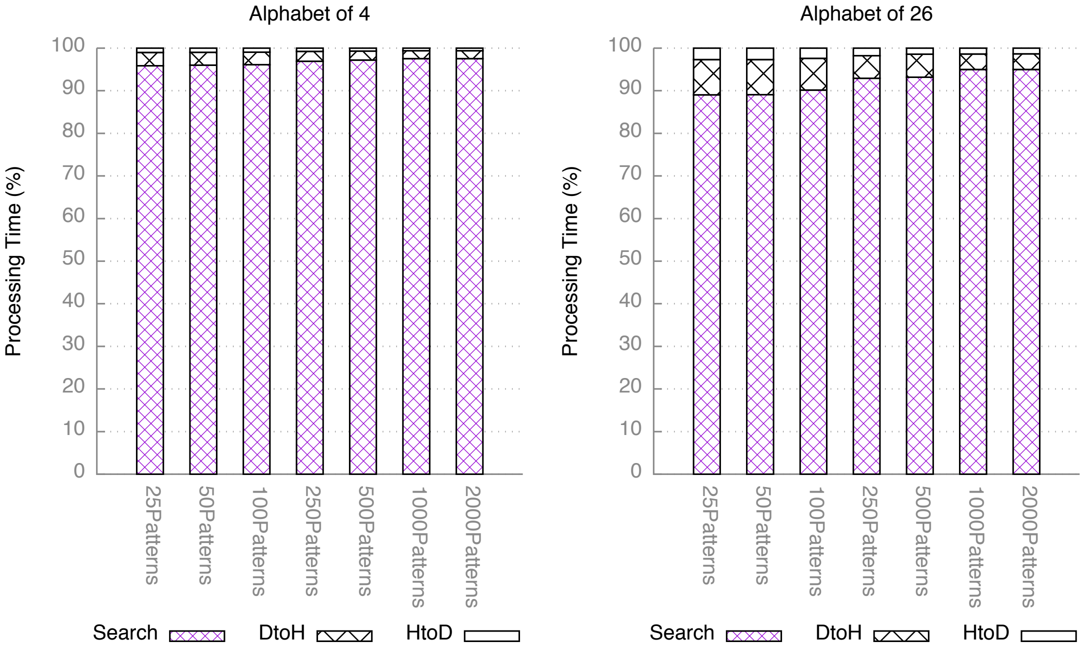


Figure 4.24
Overall GPU Preprocessing Breakdown Over Different Alphabets

Figure 4.24 shows the overall GPU processing time breakdown including the searching phase, the Host to Device (HtoD) transfer and the Device to Host (DtoH) transfer. Individual execution times for different pattern numbers and various alphabets are shown. Figure 4.24 does not include the pre-processing time, as the trie is only constructed once and the asynchronous execution of the data transfers have been aggregated together.

Figure 4.24 (Left) demonstrates a trend where the DtoH transfer requirements decrease with the increasing number of patterns; this trend can also be seen in Figure 4.24 (Right). This behaviour is due to two factors. The first factor speculates that, as the number of patterns increases, more branches can be merged together, hence reducing the trie size and thus the transfer time. While the second factor and most likely the one with the most impact speculates that as the number of patterns increases, the divergence during the searching phases increases as well, requiring more time during the searching phase. Hence the transfer time decreases proportionally.

The first factor is expected to have an impact on the overall processing time, as demonstrated in Figure 4.24 (Right), with an increasing alphabet size less branches are merged together and more time is required for the DtoH transfer. However, as the number of pattern increases the overall time required for the DtoH transfer the time required by the searching phase increases, hence requiring over 90% of the overall processing time.

As displayed by the results, the throughput of the HEPFAC algorithm demonstrates 8.5 Gbps, however, the searching phase still accounts for over 90% of the overall processing time. In order to decrease the time required by the searching phase and increase the throughput of the algorithm, multiple modifications to the core of the algorithm are required, such as the memory representation of the bitmap and the type of memory storage used by the algorithm. These optimisations and core modifications are investigated in the subsequent section.

4.6 Node Expansion Allegory

In order to increase the processing throughput of the HEPFAC algorithm, different modifications of the storage scheme and the searching phase are undertaken. This process has been called "node expansion allegory", as the expansion of the nodes is a figure of speech.

The node expansion allegory uses a similar construction technique as the one described in Section 4.3. However the ability to store the nodes in a different memory scheme allows the algorithm to increases its processing throughput by a factor of 12 compared to its original version while demonstrating same compression. This is possible by taking advantage of GPU architecture (memory, 2D locality) and the core modification of the search engine of the HEPFAC.

4.6.1 Storage Model

The primary component of the memory compression scheme lies in the way data is stored in memory as previously described in Section 4.3.1, however, in order to increase the throughput of the algorithm a second version of the algorithm was created. This version undergoes the same three steps during the construction process, namely:

Step 1 Patterns are added to the trie in a breadth-first construction row-major ordering array.

Step 2 The trie is constructed using bitmaps and offsets only identifying the first child of the current node.

Step 3 The trie undergoes the reduction process, merging all last nodes together, as well as merging similar branches based on the last three levels.

The core changes to the storage scheme occur during step 2 and 3, the data is reordered in a matrix fashion, after the breadth-first has occurred. This modification helps the nodes to be stored in texture memory and take advantage of the cache. Texture memory also increases throughput as no coalesced accesses are required.

In order to accommodate the changes to the storage scheme, the search algorithm is modified. The modification helps finding the first child of the current node while using a two dimensional matrix instead of the row major ordered array used in the previous implementation.

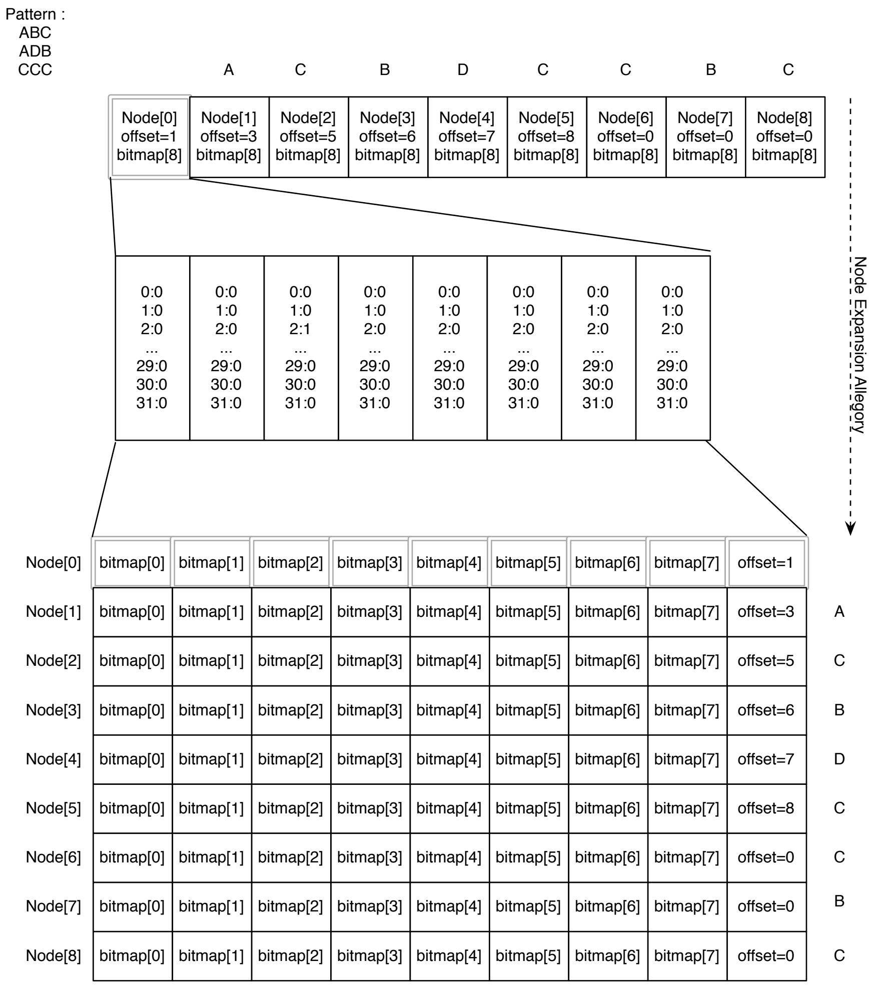

Pattern :
ABC
ADB
CCC
A C B D C C B C
Node[0] offset=1 bitmap[8]
Node[1] offset=3 bitmap[8]
Node[2] offset=5 bitmap[8]
Node[3] offset=6 bitmap[8]
Node[4] offset=7 bitmap[8]
Node[5] offset=8 bitmap[8]
Node[6] offset=0 bitmap[8]
Node[7] offset=0 bitmap[8]
Node[8] offset=0 bitmap[8]
0:0 1:0 2:0 ... 29:0 30:0 31:0
0:0 1:0 2:0 ... 29:0 30:0 31:0
0:0 1:0 2:1 ... 29:0 30:0 31:0
0:0 1:0 2:0 ... 29:0 30:0 31:0
0:0 1:0 2:0 ... 29:0 30:0 31:0
0:0 1:0 2:0 ... 29:0 30:0 31:0
0:0 1:0 2:0 ... 29:0 30:0 31:0
0:0 1:0 2:0 ... 29:0 30:0 31:0
Node Expansion Allegory
Node[0] bitmap[0] bitmap[1] bitmap[2] bitmap[3] bitmap[4] bitmap[5] bitmap[6] bitmap[7] offset=1
Node[1] bitmap[0] bitmap[1] bitmap[2] bitmap[3] bitmap[4] bitmap[5] bitmap[6] bitmap[7] offset=3 A
Node[2] bitmap[0] bitmap[1] bitmap[2] bitmap[3] bitmap[4] bitmap[5] bitmap[6] bitmap[7] offset=5 C
Node[3] bitmap[0] bitmap[1] bitmap[2] bitmap[3] bitmap[4] bitmap[5] bitmap[6] bitmap[7] offset=6 B
Node[4] bitmap[0] bitmap[1] bitmap[2] bitmap[3] bitmap[4] bitmap[5] bitmap[6] bitmap[7] offset=7 D
Node[5] bitmap[0] bitmap[1] bitmap[2] bitmap[3] bitmap[4] bitmap[5] bitmap[6] bitmap[7] offset=8 C
Node[6] bitmap[0] bitmap[1] bitmap[2] bitmap[3] bitmap[4] bitmap[5] bitmap[6] bitmap[7] offset=0 C
Node[7] bitmap[0] bitmap[1] bitmap[2] bitmap[3] bitmap[4] bitmap[5] bitmap[6] bitmap[7] offset=0 B
Node[8] bitmap[0] bitmap[1] bitmap[2] bitmap[3] bitmap[4] bitmap[5] bitmap[6] bitmap[7] offset=0 C

Figure 4.25
Node Expansion Allegory: Memory Scheme Transformation

The main difference in the storage scheme resides in the internal organisation of the data nodes. Previously each node stored an offset and a bitmap such as demonstrated in Figure 4.25 (Top). Each of the nodes was self contained in a data structure, whereas the improvement allowed the nodes to co-exist in a two dimensional matrix as demonstrated in Figure 4.25 (Bottom). Each row represents one node, containing nine cells. The first 8 cells contain 32 bits each for a bitmap of $\Sigma = 256$, while the ninth cell contains the offset identifying the first child of the current node.

As the storage scheme is exclusively represented by integers in a two dimensional array, the data can be stored in texture memory. The characteristics of the texture memory as described in Section 3.3.1 is the optimised memory layout for two dimensional spatial locality. Another requirement of texture memory is that it can only store primitive data types such as *integer*, *float* and *double*, making the storage of the first implantation of the HEPFAC algorithm impossible as each node is represented by a structure containing an array of integers representing the bitmap, and an integer value representing the offset.

4.6.2 Processing Engine

The patterns are forwarded independently of the text strings to the processing engine for analysis. The role of the processing engine is to retrieve the patterns stored in texture memory and compare them against the text string stored in global memory, in order to identify the patterns searched for.

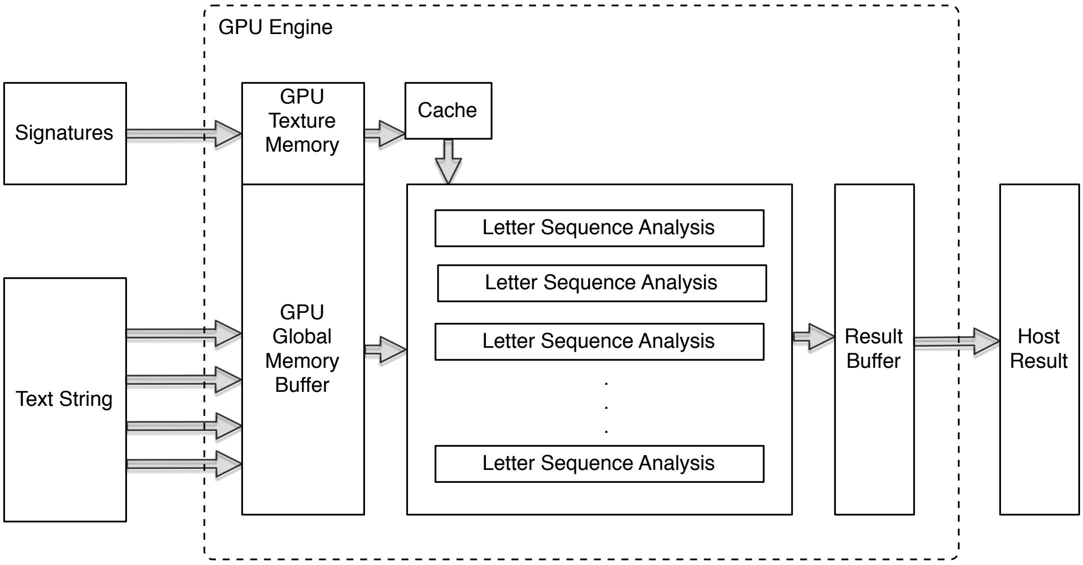


Figure 4.26
Node Expansion Allegory Texture Memory Processing Engine

Figure 4.26 shows the layout of the processing engine. The GPU accesses the patterns via the texture memory cache and specialised hardware, the patterns are then matched against the text string residing in global memory. When the searching phase is completed, the results are sent back to the host.

4.7 Throughput and Performance Evaluation

The performances of the second version of HEPFAC algorithm are evaluated against different files generated using the Mersene Twister (MT) uniform pseudo-random number generator [78]. The throughput is represented by the average of 100 evaluations against 5 different synthetic files representing different alphabet sizes.

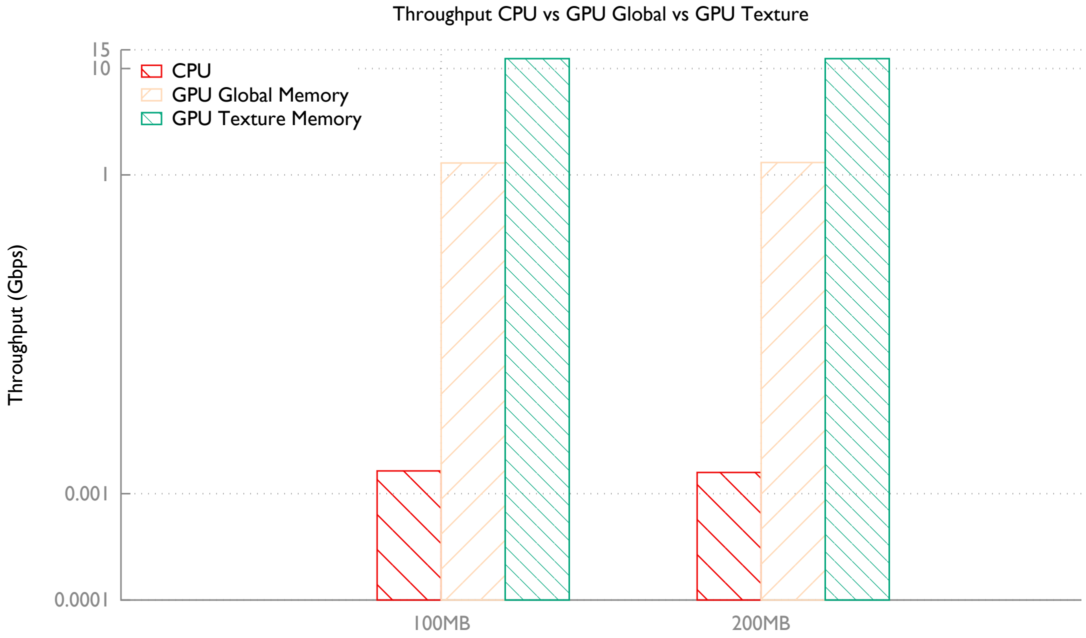


Figure 4.27
Comparison of the Different Versions of the HEPFAC Algorithm

Figure 4.27 demonstrates the performances of the different versions of the HEPFAC algorithm when analysed against different text strings. Both implementations demonstrate the performance of the algorithm when matching concurrently 1000 patterns randomly selected in the text string.

The algorithm demonstrates constant performance regarding the size of the text string as shown Figure 4.27 (logarithmic scale); it also demonstrates over 9.59 fold increase compared to its earlier version, demonstrating a throughput of 12.40 Gbps.

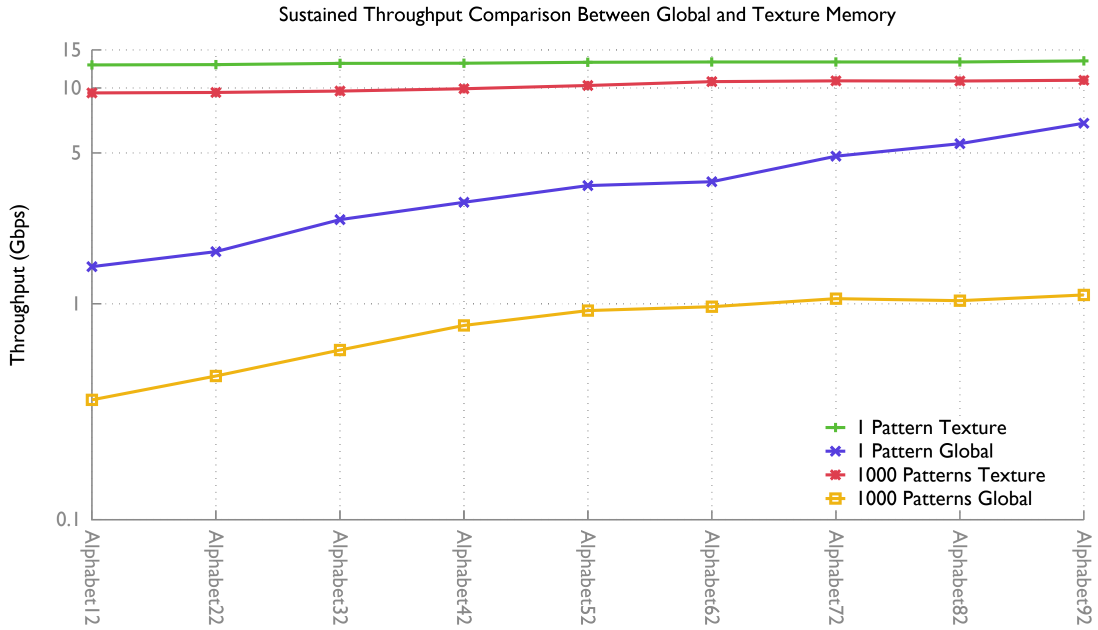


Figure 4.28
Performance Comparison Between the First and Second implementations of the HEPFAC algorithm

The node expansion provided by the second version of the HEPFAC algorithm demonstrates over 11 times improvement over its global memory counterpart implementation, when used with a large alphabet and matching a thousand patterns. The algorithm also demonstrates a 2.5 fold improvement when matching a single pattern over larger alphabets. The matching phase observes the same properties as the first version of the HEPFAC implementation, with the throughput increasing along with the alphabet size as shown in Figure 4.28.

The modified memory scheme confers on the algorithm the ability to store a large portion of the trie in the cache and eliminates the requirements for coalesced memory accesses, in contrast with global memory, which requires coalesced memory access, hence reducing the performance of the first version of the HEPFAC algorithm as shown in Figure 4.28 (note the logarithmic scale).

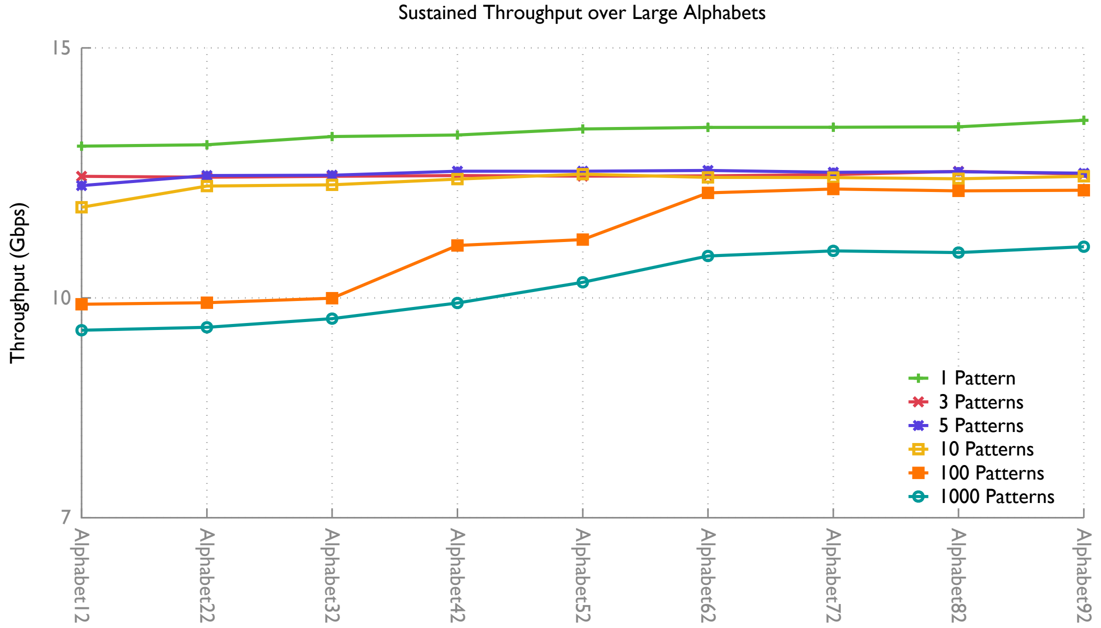


Figure 4.29
Throughput Analysis of the second version of the HEPFAC Algorithm with Different Pattern Number

Figure 4.29 demonstrates the performances of the second version of the HEPFAC algorithm when used over different number of alphabets and an increasing number of patterns. With the increasing number of patterns the throughput of the algorithm increases, the algorithm also demonstrates better performances when matching a lower number of patterns simultaneously, due to the reduced amount of divergence between the threads when matching.

The algorithm also demonstrates similar throughput for 1,3,5, and 10 patterns; this behaviour had been observed previously in Figure 4.19 in the first implementation of the HEPFAC algorithm, however the throughput was significantly different between smaller and larger alphabets, whereas in Figure 4.29 the throughput has negligible variance. This is speculated to be due to the cache; as the trie is small, a large amount of the trie can be cached, hence, lower variance between smaller and larger alphabets is observed.

4.8 Processing-Engine Performance Comparisons

The evaluation of different versions of the HEPFAC algorithm was made by using an off-the-shelf GPU. However, as the performance of the algorithm depends on the architecture of the GPU and the datasets, it is currently not possible to evaluate the throughput of the algorithm against current research in the field.

Researcher, do not release the sources of their algorithms or their implementations. Current research indicates the lack of available binary and datasets, for throughput comparison in the field of pattern matching. The sources of the code are often not released for intellectual property concerns, or are part of current and ongoing research, whereas the lack of binary is due to the increasing number of architectures for which the algorithm requires to be compiled.

In order to mitigate the hardware problem and allow other researchers in the field to compare the performances of their algorithm against the HEPFAC, the searching phase has been run on an Amazon EC2 Cloud instance. The *g2.2xlarge* AWS EC2 instance offers the possibility to run code on Nvidia GRID K520, containing 2 GK104 GPUs, featuring 8 GB GDDR5 (4 GB/GPU).

With the increasing data processing requirements Amazon AWS EC2 offers GPU computing at a reasonable rate of $0.65 per hour [122]. Cloud computing allows researchers to run their algorithms individually while allowing other researchers to compare the performance of their algorithm, harmonising the hardware dependence problem.

In order to solve the datasets problem, the synthetic datasets can be released via a repository on Github, or a dedicated website, allowing to compare the performance of the pattern matching engine against the same data.

4.8.1 Throughput and Performance Analysis on Amazon EC2

The throughput of the algorithm is evaluated on the Amazon EC2 Cloud with different synthetic datasets. The performances presented in this section are the results obtained solely by the execution of the matching kernel on the GPU.

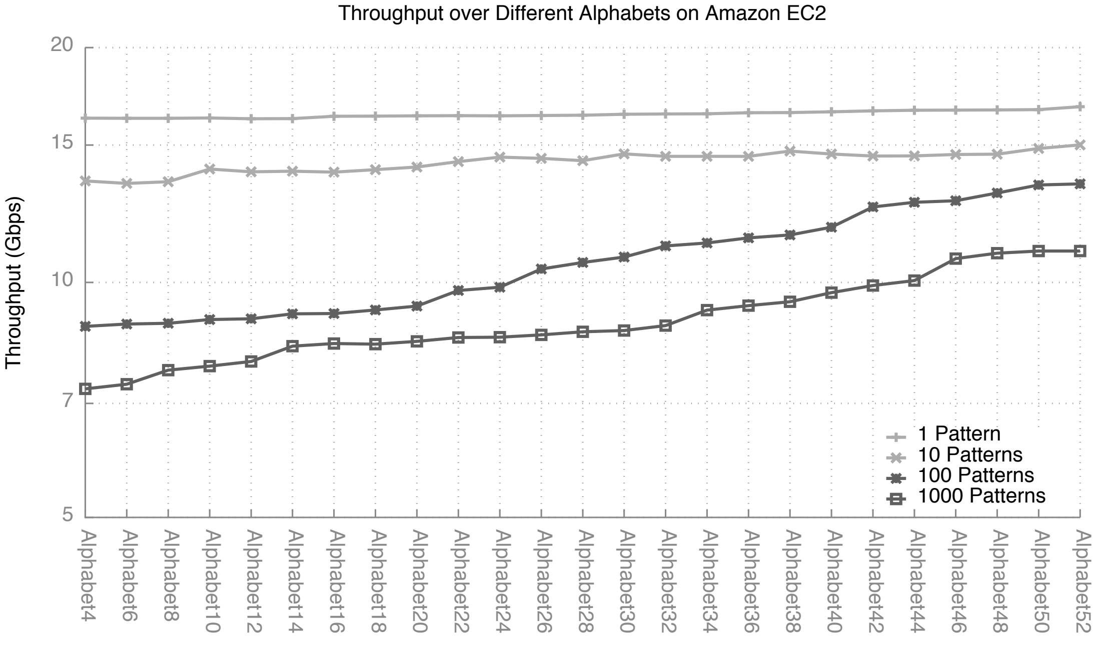


Figure 4.30
Throughput Analysis of the second version of the HEPFAC Algorithm with Different Pattern Number on the Amazon EC2 Cloud

The HEPFAC algorithm demonstrates over 15 Gbps throughput when matching a single pattern and over 10 Gbps throughput when matching a thousand patterns over an alphabet $\Sigma = 52$ on the Amazon EC2 Cloud as shown in Figure 4.30 (note the logarithmic scale of the Figure). The Amazon EC2 implementation observes the same trends as those obtained on the Nvidia K20 GPU. With the increase of alphabet size the throughput rises, and the increasing number of patterns are reducing the throughput of the algorithm, due to divergence as explained in Section 4.7. The algorithm also demonstrates a slow increase in throughput when matching a single pattern, as observed earlier.

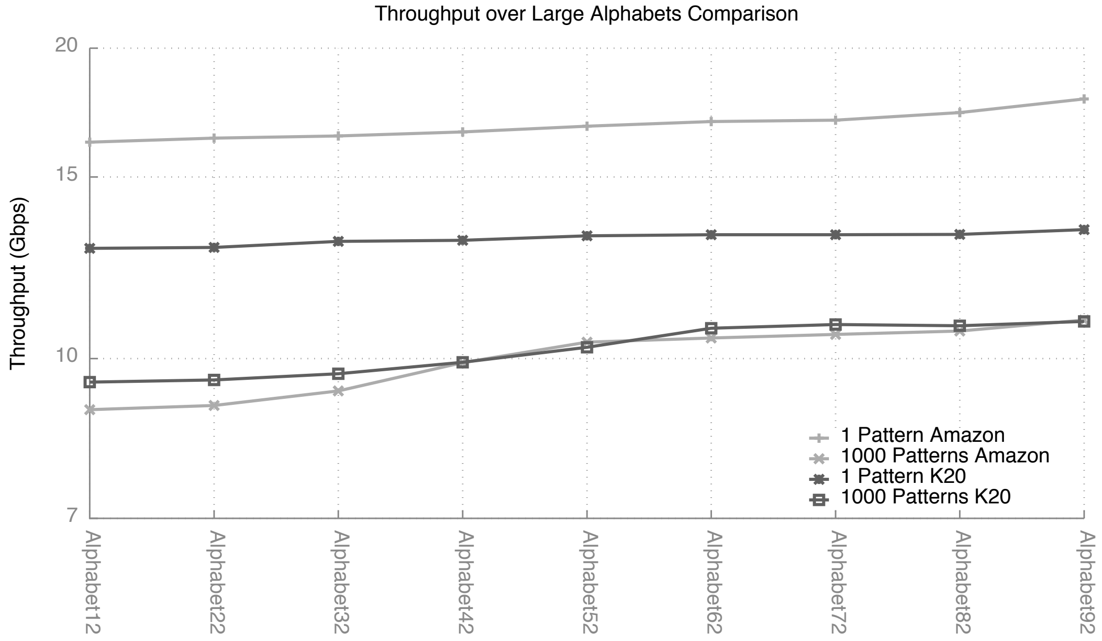


Figure 4.31
Throughput Analysis of the second version of the HEPFAC Algorithm with Different Pattern Number on the Amazon EC2 Cloud and the Nvidia K20

Figure 4.31 draws a comparison between the results obtained on the Amazon EC2 cloud and the results obtained on the Nvidia K20. The results demonstrate better performance on the Amazon cloud EC2 with the Nvidia GRID K520 when matching one single pattern over all the alphabets. However when matching a thousand patterns concurrently, the Amazon cloud performs very similarly to the Nvidia K20; it is speculated this is due to the similar architecture of both GPUs.

The technique of comparing throughput over similar hardware and with similar data in order to experiment and compare results is already in application in other domains such as biology [123]. Using the Amazon EC2 cloud is cost effective and allows researchers in the domains using pattern matching to evaluate their algorithms against each other, draw conclusions and further improve research.

4.9 Summary

Data compression is the art of reducing the number of bytes required to store the same amount of data. This chapter introduced a novel and highly efficient memory compression scheme.

Two versions of the algorithm are presented in this work, making use of an efficient storage model, augmented by a bitmap in order to reduce the size of the nodes and a trie reduction algorithm merging the similar trie branches together.

The first version of the HEPFAC algorithm was demonstrated in Section 4.3. The trie is constructed on the CPU in an 1D breadth-first / row-major ordered array fashion, allowing each cell to reside in contiguous memory. Each node contains a bitmap and an offset, the bitmap allows to reduce the number of integers required to store the alphabet. Once the trie is fully constructed, the algorithm truncates the trie in accordance with the alphabet in use. Finally similar suffixes are merged together in order to reduce the number of branches and number of nodes required to store the trie in memory.

The second version of the HEPFAC algorithm was demonstrated in Section 4.6. The trie is constructed on the CPU using a novel node expansion allegory algorithm, which allows the trie to be stored in a 2D memory scheme, while preserving the size of the trie. The trie is constructed using the same process as the first version of the HEPFAC.

The first version of the HEPFAC was solely stored in global memory and while it demonstrated over 8 Gbps throughput while matching a single pattern, it also presented significant drawbacks, such as throughput limitations while increasing the number of patterns being matched. This problem was subsequently solved in the second version of the HEPFAC where the trie was stored in texture memory. This allowed to increase the matching throughput, while preserving the size of the trie. Storing the nodes in a 2D array, in texture memory allowed the threads to access un-coalesced memory without penalty.

Moreover in this chapter the construction and the reduction algorithm were analysed against different pattern sizes and alphabets, demonstrating the performance of the algorithm such as the linear time construction and the quadratic time for the trie reduction.

It is strongly believed that a number of applications can benefit from the memory reduction proposed in this work, such as GPU firewalls, deep packet inspection, spam filters, RNA sequencing and genetic matching systems, as these applications are required to process increasing amounts of data and as GPUs memory is a scarce resource.

4.9.1 Price Vs Performance

In order to achieve the performance and the ability to use the latest features of the CUDA framework, the Nvidia Tesla K20x was selected, along with two Intel Xeon E5-2620 CPU, as well as the Amazon EC2.

Table 4.2
Hardware Price

Model	Quantity	Price Per Unit
Intel Xeon E5-2620	2	£ 263.65
Nvidia Tesla K20	1	£ 1186
Amazon EC2	1	£ 0.46/h

Table 4.2 demonstrates a total of £ 1714 as of June 2013 producing a throughput per pounds cost of 7.24 Mbps/£while preserving the limited size of the trie to a minimum.

4.9.2 Limitations

The first version of the HEPFAC was subject to different limitations trading off the throughput for the space requirement. Furthermore, the un-coalesced memory accesses during the matching increased the overall processing time. However the second version of the HEPFAC algorithm subsequently overcame these limitations by storing the patterns in texture memory.

The construction and reduction algorithm are also subject to limitations, with the increasing number of patterns; the trie construction time will grow linearly, while the trie reduction will grow in a quadratic fashion, requiring more pre-processing time. These limitation can currently be discarded as the time required for reducing the 60,000 ClamAV rules [76] is a 40 Seconds at first launch of the algorithm.

The existence of GPU cloud is a testament for their usefulness and capabilities. As demonstrated in Table 4.2, GPU cloud Computing provides an affordable alternative to the Nvidia K20. However cloud computing comes with limited data privacy, raising a number of issues which will be explored in the next chapter.

Chapter 5

Data Remanence and Privacy on GPU

5.1 Problem Statement

With the exponential growth of data analytics and data storage, GPUs have been used in various research as cost-effective off-the-shelf High Performance Computers (HPC). Different work has shown their efficacy in Intrusion Detection Systems [124], Deep Packet Inspection [125], pattern matching [126], as well as database processing [127]. These applications leverage the highly parallel processing capabilities offered by General-Purpose Graphics Processing Unit but do not ensure the confidentiality of the data processed. Moreover cloud providers often describe GPU-as-a-service as secure, however, while this is often true for the Operating System (OS) other component of the system such as GPUs can be impacted by data remanence and data leakage. This is exacerbated by the large onboard memory of the GPU.

GPUs are inherently used to offload computationally intensive tasks from the CPU; often such tasks require the processing of confidential information such as cryptographic keys [128], network traffic [129], or financial data [130]. Graphics Processing Units are often described as HPCs and marketed as (GPU-as-a-service) such as the Amazon AWS EC2.These infrastructures often share resources between multiple actors and can be used intentionally by malicious users for password cracking [131], browser information retrieval [132]or to conceal malware [133].

Even within single-user environments, using more than one application on the same GPU may pose risks; if a user runs an intrusion detection system or malware scanner on their GPU, subsequent GPU-accelerated software may have access to the memory used by the antivirus scanner, which scanned files with restricted access permissions. Therefore there is potential for GPU memory to contain protected information, since the GPU is regarded as a trusted processing device.

Furthermore there is a growing need for Reverse Engineering (RE) in the field of GPU programming. As the number of threats increases, RE can be used to gain an understanding of code being executed; for example in software auditing, the use of reverse engineering could ensure that software handling sensitive data is not abusing its privileges. Additionally, as GPU applications often represent a commercial investment of the originating company, reverse engineering can pose a direct threat through the theft of Intellectual Property (IP) [134].

This chapter investigates the feasibility and practicality of digital memory forensics on different GPU architectures with different versions of the CUDA framework. To this end the state of the art is outlined, followed by an investigation and evaluation of the possibilities of Reverse Engineering for CUDA applications, with a particular focus on security and privacy concerns for applications holding IP. Static and dynamic analysis techniques for RE in order to help digital forensic investigators are then highlighted. Finally a novel RE and DF methodology is presented.

5.2 Background

5.2.1 Security, Privacy and Confidentiality on GPUs

Maurice *et al.* [135] discuss the confidentiality issues related to GPUs in virtualised environments in *Confidentiality Issues on a GPU in a Virtualised Environment*. The authors demonstrate data leakage in a virtualised cloud computing environment, with privacy issues created by the different virtualisation techniques alongside the security implications. For their experimental setup Maurice *et al.* use KVM and XEN, running a Linux kernel. The research focuses on global memory data leakage and demonstrates the ability to recover data, including situations where a malicious user is able to recover data while launching a virtual machine after the original author has stopped his virtual machine, bypassing the security mechanism in place. The authors then describe cloud countermeasures.

Ladakis *et al.* [136] demonstrates the ability to run a keylogger on the GPU in *You Can Type, but You Can't Hide: A Stealthy GPU-Based Keylogger*. The authors demonstrate a novel approach, by stealthily hiding a keylogger on the graphics card, redirecting the keyboard buffer directly to the GPU from the DMA without requiring modifications to the linux kernel code. The malware is able to run on two different Nvidia graphics cards, respectively a GTX 480 and a GT 360. The authors demonstrate the possibility of recovering every keystroke on the keyboard and apply a regular expression algorithm on the recovered data in order to find credit card numbers. Finally the author discuss different countermeasures in order to prevent GPU malwares.

A *GPU-Assisted Malware* is presented by Vasiliadis *et al.* [133]. The authors demonstrate the abilities for malware to increase its robustness by hiding in GPUs. The malware implemented by Vasiliadis *et al.* is made using unpacking and run-time polymorphism techniques. The authors then improve their malware in *GPU-Assisted Malware* [137] and implement a brute-force unpacking technique, with a run-time polymorphism as well as running different parts of the malware on different GPU architectures. The author also describe potential attack vectors and future threats. The authors then describe possible defences against malware and discuss GPU malware detection systems.

Danisevskis *et al.* [138] demonstrate how mobile GPUs can be used as an effective payload delivery method for numerous new types of attacks. The authors demonstrate these attacks in the context of mobile devices, using the capabilities of the Direct Memory Access in order deliver malicious applications as well as for privilege escalation. The paper highlights mobile GPUs weaknesses and describes a successful attack. Furthermore, a proof of concept against a phone running Android and using an ARM Mali MP 400 GPU is described. The papers also emphasises the problems engendered by the use of different programming models on mobile GPUs.

A GPU vulnerability allowing malicious user to steal webpages is proposed by Lee *et al.* [139] in *Stealing Webpages Rendered on Your Browser by Exploiting GPU Vulnerabilities*. The paper demonstrates that new memory allocated by the host on the GPU is not properly initialised, on both Nvidia's and AMD graphics cards. By exploiting different vulnerabilities the authors are capable of dumping webpages with an RGB histogram accuracy of up to 95.4 %. The authors also demonstrate that the attack can be executed during the execution of the rendering or after the rendering is done and is tested against 5 different platforms including an Nvidia GeForce 210, an Nvidia Tesla C2050, an Nvidia GeForce GTX 780 and AMD Radeon HD 7850 and an AMD FirePro W9000.

5.2.2 Digital Forensic and Reverse Engineering

Byrne *et al.* [140] present the conclusions of an experiment of reverse engineering Fortran software into an Ada implementation, allowing the authors to upgrade the existing documentation and extract software design information directly from the fortran binary. The authors then used the gained information to understand the design constraints and the inner flow of the software. Byrne *et al.* follow a 7 step process including the collection of information, examination of information, extraction of the structure, the recording functionality, the recording of data-flow, the recording of the control-flow and the control of the final re-designed application in Ada. The experiment revealed major issues such as the problems occurring during re-engineering of the software and the erroneous information obtained. The authors then describe the lessons learned.

Kato *et al.* [141] demonstrate in their paper *Implementing Open-Source CUDA Runtime* how often GPUs are treated as blackboxes due to the proprietary strategies implemented by vendors. Kato *et al.* highlight in detail the GPUs architectures and the run-time mechanisms, helping research to tackle novel problems occurring in GPU architectures and dedicated software. In their work the authors propose an open-source implementation of the CUDA runtime based on linux and targeted towards Nvidia GPUs.

In the paper *The impact of GPU-assisted malware on memory forensics: A case study* [142] the authors analyse the impact of GPU-assisted malware and highlight the difficulties of GPU assisted malware and memory forensics. The authors primarily discuss numerous techniques malicious software could use in order to hide its presence from the user and the host antivirus. The authors then describe how the host's memory forensic could highlight the presence of such an application and different cases in which forensic investigators might require access to the GPU memory. The authors focus their research on the Intel Integrated Graphic Devices (IGDs), using the Linux Direct Rendering Infrastructure (DRI), allowing them to access the GPU using user space applications. They also highlight the possibilities of different scenarios on other GPU devices. The authors also developed a set of custom tools in order to parse a number information retrieved via the GPU memory and by the operating system.

Breß *et al.* [143] demonstrate how to retrieve information from a co-processing GPU for databases in *Forensics on GPU Co-processing in Databases – Research Challenges, First Ex-*

periments and Countermeasures. The key challenges for digital forensic are pointed out by the authors, while focusing on global memory, the authors demonstrate the ability to retrieve data. The authors clearly outline the requirements for a forensic methodology while pointing out possible solutions as well as anti-forensic methods in order to increase the privacy of databases systems using GPUs as co-processors.

Di Pietro *et al.* [144] describe security issues related to GPGPU and particularly Nvidia GPUs working with the CUDA framework. The work described in *CUDA Leaks: Information Leakage in GPU Architectures* highlights severe information leaks from the different memory, on the Fermi and Kepler architectures and described shared vulnerabilities between these architectures. The case study described by Di Pietro *et al.* also demonstrates vulnerabilities that can affect real world applications, such as AES encryption algorithm running on GPUs. The authors also highlight shared memory vulnerabilities and identify partial register spilling on both Tesla C2050 and the GeForce GT 640. Finally the authors propose different countermeasures for the shared and global memory, such as for the registers.

5.3 CUDA Programming Layer

5.3.1 CUDA Binaries

The CUDA framework allows developers to take advantage of the massively parallel processing capabilities of graphics processors, allowing GPUs to become widely available general purpose computing devices. The CUDA framework largely extends the instruction set of to the C99/C++ languages, allowing developers to take full advantage of the parallel nature of the hardware, through the use of a flexible abstraction model.

CUDA applications are compiled by `nvcc` (the Nvidia CUDA Compiler). The compiler takes a set of source files as input and generates a variety of compiled output formats, allowing backward and forward compatibility between devices. The standard output format from the nvcc compiler is an Executable and Linkable Format (ELF)-based host executable, which is executed like any standard program on the host CPU. This program then initialises the GPU via the CUDART library and loads the relevant CUDA code onto the GPU. The ELF binary is a container, holding a variety of different formats of program code.

The GPU code present in the CPU-executed ELF binary is known as a *kernel*. When invoked, the kernel first allocates the required memory on the GPU and loads the desired data onto the device for processing. After processing, the resulting output is transferred back to the host CPU memory and the GPU memory is released for use by other applications.

While a regular piece of software to be executed on the host CPU would typically be translated to machine code, there are a number of different output formats that CUDA code can take. All of these output formats can be bundled in the ELF binary. Some of these make the process of reverse engineering of the compiled code considerably easier than others.

The highest level of abstraction is the original CUDA C/C++ source code. It is highly unlikely, however, that source code for a given binary would be available to a forensic investigator attempting to ascertain what was being done by a piece of software on a GPU. Nonetheless, if open-source code was being used (and an investigator could determine this), it may be possible to acquire the source code from the original authors and inspect it to understand the operation of that code.

The CUDA source code of a given piece of CUDA software is not distributed as part of the executable binary. Like with regular programming languages for host CPUs, the source code undergoes compilation to produce an executable binary for the target instruction set. Due to the rapid development and improvements in GPU technology and their associated capabilities, CUDA GPUs do not all support the same baseline instruction set as is the case with an x86-based CPU, for example. Where almost all general purpose desktop CPUs support the 8086 instruction set with full backward compatibility, this is not the case with Nvidia CUDA GPUs [145] [146] [43].

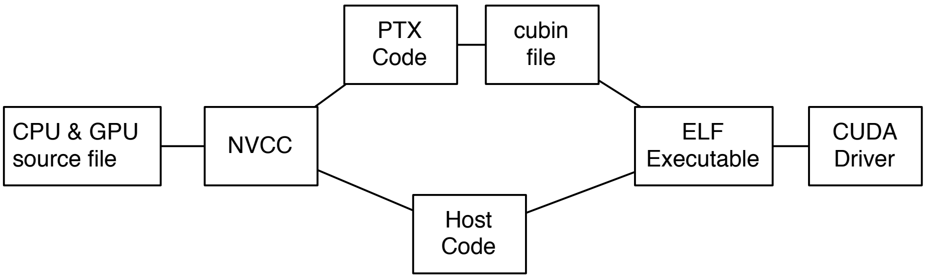


Figure 5.1
Compilation Process

Compiled CUDA binary code is often referred to as a Cubin, a contraction of CUDA and binary, which contains executable machine code, designed to run on a given target architecture. As a result of the heavily architecture-dependent nature of Cubin executables, the Nvidia CUDA `nvcc` compiler compiles CUDA source code into a platform-independent intermediary language, known as Parallel Thread Execution (PTX). PTX is laid out in an assembly-like structure using basic primitive instructions for all operations, while still remaining GPU-agnostic (allowing for the same code to be executed on different GPUs) [147]. It is produced by the `nvcc` compiler, which translates CUDA code into PTX rather than directly to assembly, as a conventional CPU compiler would. The CUDA graphics driver can process the resulting PTX, producing native binary code (a cubin), capable of being executed on the GPU [148] [110]. Figure 5.1 demonstrates the compilation process achieved by `nvcc`. The code executed on the host is compiled using the `GCC` compiler while the code executed on the GPU is translated into PTX code and Shader ASSeMbly (SASSM) instructions before being bundled with the host code in a single *cubin* file that will later on be executed by the CUDA driver.

If a host ELF binary contains PTX intermediary code, a suitable CUDA binary for any supported GPU architecture can be compiled at runtime (including, GPUs which were not released at the time of writing the code), since the PTX code is platform-agnostic and can be targeted at the detected GPU architecture based on the register mapping data contained within the CUDA driver. By including PTX code in the output binary, the compiled CUDA host ELF executable will be as generic as possible and will enjoy a degree of forward compatibility with future GPUs [43]. Given the nature of the PTX format, however, its presence will significantly aid the disassembly and reverse engineering processes, since it yields highly readable and generic intermediate-language code somewhat similar in nature to Android's intermediary Dalvik code [149], that can be understood with relative ease.

SASSM is used as a definition of the instruction set implemented by the GPU. Each major GPU architecture (Tesla, Fermi, Kelper) has its own instruction set [43]. This allows new GPU architectures to change their underlying technical design, to improve performance without need for consideration of backwards compatibility. By combining the appropriate target specific register definitions with platform-independent PTX code, the GPU driver can produce an executable SASSM re-bundled into a CUDA binary (Cubin), designed for execution on the correct GPU architecture.

As such, from the perspective of a forensic investigator wishing to carry out an investigation of code running on a GPU, they are likely to encounter one of two scenarios. The first and most likely scenario is that the code running on the GPU was executed using a host binary containing PTX code for the algorithm in question. Since the host binary contains the PTX code, it is relatively straightforward to extract the PTX section of the binary and use it to determine the operation of the CUDA code in question. The second and less likely scenario is that the code being executed on the GPU has the PTX section stripped, perhaps in order to hinder inspection or reverse engineering of the code. In this case, the platform-specific compiled Cubin code must therefore be disassembled, to yield a listing of the machine code being executed.

5.4 Experimental Environment

The different experiments have been realised on two different machines, assessing mobile, consumer and high end devices. The high end and consumer grade devices, are respectively an Nvidia K20m graphics card, composed of a single PCB, consisting of 192 CUDA cores and 13 multiprocessors and an Nvidia GTX 295 composed of 2 PCB, each of them containing 240 CUDA cores, distributed over 30 multiprocessors. The GTX 295 is constituted of a total of 2 GB of RAM. Both graphics cards have been used on a Supermicro server, including two Intel Xeon E5-2620 and running Ubuntu 14.04.3 LTS.

Table 5.1
Experimental Environment Specifications

GPUs	GTX 295	Tesla K20m	Tegra K1
Number of GPUs	2	1	1
CUDA Version	5.5, 6.0, 6.5	5.5, 6.0, 6.5	6.0
CUDA Capabilities	1.3	3.5	3.2
GPU Architecture	GT200B	Kepler GK110	Kepler GK20a
Warp Size	32	32	32
CUDA Cores	8	192	192
Multiprocessors	30	13	1
Global Memory	896 MB	5 GB	2 GB

The tests carried on the mobile device have been done on a Nvidia Jetson Tegra K1, corresponding to the first mobile processor released by Nvidia. The Tegra k1 is based on the Kepler architecture, featuring one streaming multi-processor of 192 CUDA cores. The CPU is an NVIDIA 4-Plus-1 Quad-Core ARM Cortex-A15 “r3” with 2 GB of shared RAM. The Jetson board is running Ubuntu 14.04.3 LTS.

The experiments have been realised using different versions of the CUDA framework for the consumer and high end graphics card, as demonstrated in Table 5.1; the mobile GPU however is only tested against the version 6.0 of CUDA, as this was the only version released for the Jetson TK1 at the time of the experiment.

5.5 Data Remanence on GPUs

General security flaws and data leakage on GPUs have been highlighted in different studies [135] [143] [144], however the majority of these studies concentrated on one type of GPU applied to a specific scenario, such as cloud computing, or a type or RAM, such as global memory. This section builds upon the former studies and demonstrates data remanence on global, shared and texture memory and is reproduced on consumer, mobile and professional grade GPUs. Different version of the CUDA framework are also used in order to assess the different improvement and security countermeasures present in the different updates. Furthermore a digital forensic methodology is also presented, allowing digital forensic investigators to understand the architecture of graphics processing units and take the different memory requirements of GPUs into consideration when required to retrieve data.

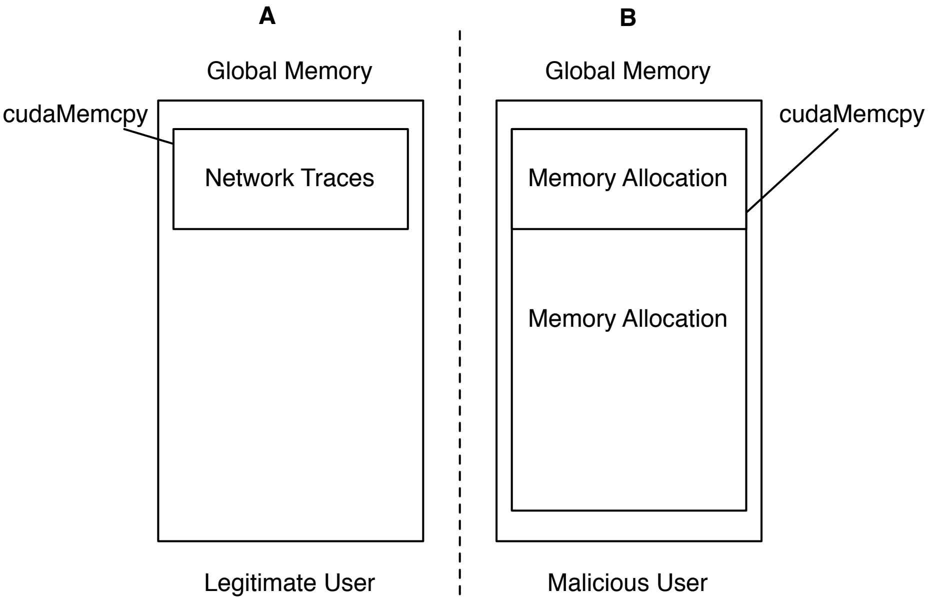


Figure 5.2
Global Memory Retrieval

5.5.1 Global Memory Scenario

Global memory represents the largest memory on the GPU. Global memory is used to store a large amount of data to process it. In order to test data leakage and data remanence, the following scenario is established:

Network traces containing confidential data are sent to the GPU and stored in global memory. A primary and legitimate user runs the network traces through an algorithm until receiving the required results. The traces are sent from the host to the device, following the standard procedure, using *cudaMalloc* and *cudaMemcpy* as shown in Figure 5.2 (A).

After the execution of the first process (by the legitimate user), a second independent user with malicious intention, runs a malicious application, requiring the allocation of the same, or larger amount of memory. The memory allocated by the malicious user will allow him to fully or partially recover the remanent data on the GPU. Figure 5.2 (B) shows the process of memory allocation by the malicious user. The malicious user is allocating the entire memory space available in global memory, then requests all data stored on the device to the host memory by using the *cudaMemcpyDeviceToHost* function.

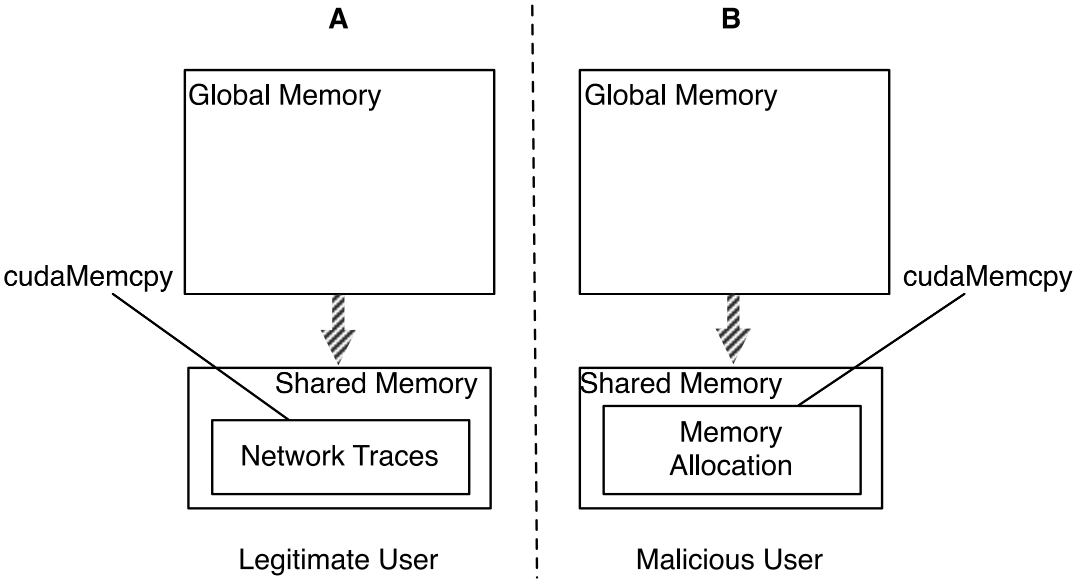


Figure 5.3
Shared Memory Retrieval

5.5.2 Shared Memory Scenario

Shared memory is on-chip memory, set to either 48 KB or 16 KB (defined by the user) as discussed in Section 3.3.1, allowing threads to exchange data between each other, as well as having fast access to data.

In this scenario a large amount of data is transferred by the legitimate user to global memory. In an attempt to process data faster the legitimate user copies data from global memory to shared memory as shown in Figure 5.3 (A); once the computation is finished, the legitimate user uses the *cudaMemcpyDeviceToHost* in order to copy the results back to the host.

Figure 5.3 (B), shows the malicious user action, requesting data directly from shared memory. This action is performed by allocating an array of the size of shared memory, transferring that data back to a buffer located in global memory, in an attempt to transfer the data back to the host by using the *cudaMemcpyDeviceToHost* function.

This scenario offers the possibility to the the malicious user to locate and retrieve a total or partial amount of data stored in shared memory. In order to recover data, it is assumed that the legitimate user is not writing multiple times at the same location in shared memory, otherwise only the last data written will be leaked. Furthermore, shared memory might be using wear levelling in order to increase the longevity of the memory (Static random-access memory (SRAM)), inherently allowing the malicious user to retrieve 48 KB of data when allocating a full buffer [150] [151].

5.5.3 Texture Memory Scenario

Texture memory requires the same amount of clock cycles as global memory in order to be accessed, however, texture memory offers the use of a cache and does not require coalesced memory accesses as explained in Section 3.3.1. These properties makes texture memory uniquely suited for different specific algorithms.

This scenario, enables the legitimate user to take advantage of the singular properties of texture memory, allocating memory texture by using the function *cudaBindTextureToArray*. Once the data is transferred onto the GPU, the legitimate user is able to process it's data faster, benefiting from the cache and the non-required coalesced memory accesses.

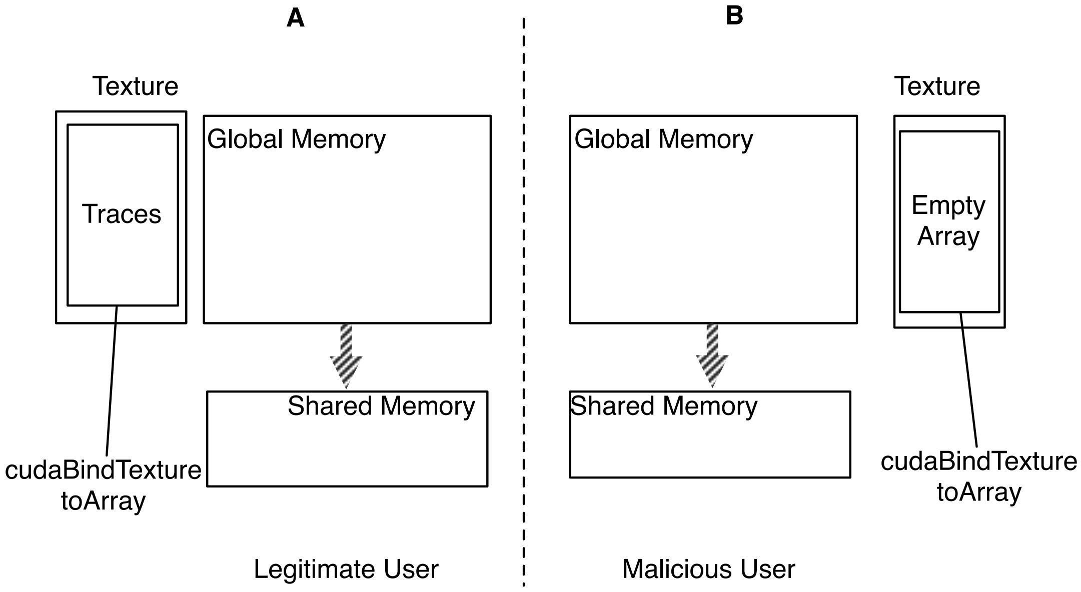


Figure 5.4
Texture Memory Retrieval

Figure 5.4 (A) shows the legitimate user processing data via the texture memory. Figure 5.4 (B) shows, the allocation of an empty array in texture memory, by the malicious user, in order to retrieve data stored in texture memory. It is important to note, that the only difference is that texture memory accesses DRAM through specific hardware and provides cached data to the user.

5.5.4 Experimental Results

This section demonstrates the results obtained by executing the different scenarios, on the consumer, mobile and high end GPU and discusses results obtained using the different versions of the CUDA framework.

Global Memory: In order to execute the first scenario, a two dimensionnal array of 2000 integer entries was created and stored in global memory. The legitimate user processing was simulated by simultaneously multiplying the integers by two. The array was then sent back from the device to the host.

The malicious user was simulated by allocating an array of the size of global memory, on the host, without transferring data to the GPU, the malicious user then instructed data to be copied from the device to the host.

The data retrieved by the malicious user was 100% identical to the data computed by the legitimate user. Hence, 100 % of the legitimate user's data remained in DRAM and was leaked by the GPUs. This process was carried out on the mobile, consumer and high end GPU demonstrating similar results.

The consumer and high end devices were also tested against CUDA version 5.5, 6.0 and 6.5, demonstrating similar data leaks. The vulnerability demonstrated on the three different GPUs can be attributed to the lack of secure software practices as discussed by [135] [143]. These security issues can also be solved by improving the GPU hardware and clearing the memory, after each use.

Shared Memory: The scenario was executed by sending an array of 2000 integer entries, stored in global memory. The array was then sent to shared memory in order to be processed and multiplied by two. The following steps were followed by the legitimate user in order to compute its data:

- Data transfer from the host to the device
- The two dimensional array is stored in shared memory
- The data stored in shared memory is multiplied by two and stored in the same cells.
- The original data (stored in global memory) is sent back from the device to the host.

In order to evaluate the data recovered by the malicious user, the data multiplied by two are left in shared memory and not transferred back to the host, in which case the legitimate user would be required to store the data in global memory before being able to transfer the data back.

The malicious user allocated an empty array in global memory, the empty array was then filled with the content of shared memory and transferred back from the device to the host. This process was realised on the three different devices.

By using CUDA version 5.5, 6.0 or 6.5, the malicious user was able to retrieve the content of shared memory by using standard I/O functions. In their work Di Pietro *et al.* [144] state that shared memory is zeroed after execution, however in the experiments described in this chapter, the data remained on the GPUs after the legitimate user processed its data. It is assumed that the behaviour of shared memory encountered by Di Pietro *et al.* is due to Error-correcting code memory (ECC) being activated on their GPU. The results presented by Di Pietro *et al.* did not take in account the fact that ECC is often disabled on computers as it allows to achieve better throughput performance [152] [153], and that consumer GPUs often do not possess ECC, as highlighted in [154] and in [155]. Hence, not reflecting the GPU landscape.

ECC is not activated on our consumer and high end GPUs in order to preserve the consistency of our tests, as ECC could not be enabled on the Nvidia Jetson Tegra K1.

Texture Memory: The texture memory was evaluated in a similar fashion to global and shared memory, with the difference that in order to store the data in texture memory a specific texture binding function was required on the host code.

The scenario was executed as follows; the legitimate user created an array of integer registered by the texture binding function, a second array of random number of the same size (2000 integer) was then allocated in global memory, the multiplication kernel was then launched from the CPU and only the data contained by the the second array stored in global memory was accessed and multiplied by two. The data stored in global memory was then send back to the host.

The data stored in texture memory were not used nor accessed by the legitimate user, as texture memory is a read-only memory. In order to retrieve the data stored in texture memory and multiply them by two, the results would have had to be stored in global memory, eventually corrupting the data leakage demonstration.

The malicious user was simulated by allocating an arrays of 2000 integer in global memory. The malicious user then proceeded to read texture memory and store the results into the global memory array. Finally the array stored in global memory is sent back to host, containing the data.

The data could be retrieved 100 % of the time, when using the 'printf' procedure, with CUDA version 6.0 and 6.5, however the data could only be copied back to the host 20 % of the time. When using CUDA 5.5 the data could be dumped back to the host 100 % of the time. The undefined behaviour of CUDA 6.0 and 6.5 leads to the conclusion that this was not a security mechanism implemented by Nvidia, but rather a quirk due to the CUDA driver as described in [144] [156].

Host and Devices Test Cases: Extensive experiments were carried out in order to demonstrate data remanence in off-the-shelf graphics cards under various realistic situations such as switching user, device reset and the hard reboot of the host machine. These results are organised into tables, "Y" meaning data remanence was observed on the GPU, "N" meaning the attempt of data recovery was unsuccessful and "N/A" stands for not applicable.

Table 5.2
User Switch

GPUs	GTX 295	Tesla K20m	Tegra K1
Global Memory	Y	Y	Y
Shared Memory	Y	Y	Y
Texture Memory	Y	Y	Y

Table 5.2 demonstrates and compares the behaviour of the mobile, consumer and high end GPUs, when a malicious process is run by a user after the legitimate user has run its algorithm. The results demonstrates that "user switching" does not affect data remanence on the GPU regarding the type of memory. The inherent sharing of memory at GPU level regardless of the user level separation on the host allows to highlight data leakage concerns, as well as potential for forensic investigations.

Table 5.3
GPU Reset

GPUs	GTX 295	Tesla K20m	Tegra K1
Global Memory	Y	Y	N/A
Shared Memory	Y	Y	N/A
Texture Memory	Y	Y	N/A

Table 5.3 shows the results of a GPU reset. This action on the GPU is carried out using the *nvidia_smi* tool available in the CUDA framework [135] [157]. It is demonstrated that a reset of the GPU does not affect the memory content of both the consumer and high end grade devices, however, it demonstrates that the mobile device does not have the capability to reset the GPU.

The possibility of data recovery for forensic investigation is demonstrated even after a GPU reset carried out via the official tools, released by Nvidia. The results also highlight that GPU resets should not be relied on, in order to erase or annihilate data remaining in memory.

Table 5.4
Hard Reboot of the Host Machine

GPUs	GTX 295	Tesla K20m	Tegra K1
Global Memory	N	N	N
Shared Memory	N	N	N
Texture Memory	N	N	N

Table 5.4 demonstrates that the GPU memory suffers from the same effect as host RAM, where when a hard reboot is carried out, all data in host RAM is lost. This occurs as data retention in the RAM requires power to be constantly applied to the device. As long as the GPU is powered, data can be recovered from all three of the memory types on the GPUs.

5.6 Strategies for Digital Forensic

Digital forensic investigators require a protocol and a methodology to follow, proceeding from the most volatile memory to the least volatile, allowing them to present significant evidence in court cases, in a reliable and verifiable fashion [158] [159]; however, extracting shared and texture memory occurs through global memory. In order to avoid overwriting any remanent content, global memory retrieval is prioritised.

In this section, a novel methodology is proposed, shortening the time of investigation and allowing higher confidence in data integrity. The methodology presented in [135] requires investigators to reverse engineer the proprietary driver, which is a time consuming and a complex task, leading to uncertainty in the data integrity. Data in memory might be changing during the acquisition proposed in [135], hence the proposed methodology might be hindered for some forensic analysis [160].

In contrast, this novel methodology does not require reverse engineering of the CUDA driver and follows the United States' Department of Justice recommendations that consists of "Preparation and Extraction" and "Identification" steps [161]. The novel methodology requires the controlled modification of program code running on the GPU while aiming at preserving the content of memory; this is not unlike mobile phone digital forensics where the requirements are similar [162] [163]. In GPU and mobile scenarios, non-invasive approaches are not always possible due to technical limitations and requirements.

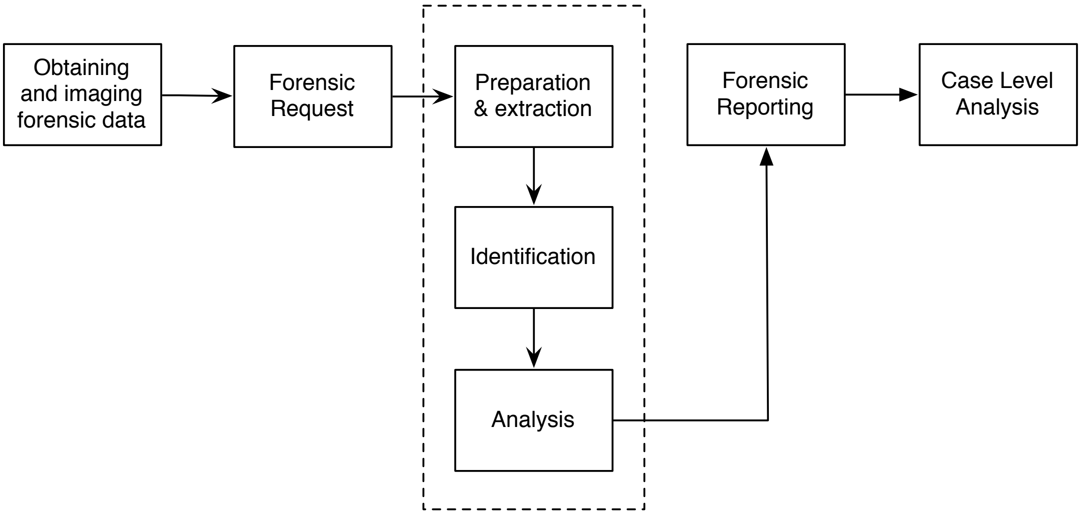


Figure 5.5
Digital Forensic Methodology Process Overview

The methodology focuses on the three intermediate steps of the forensic investigation proposed in [161], as shown in Figure 5.5. The US Department of Justice proposes a seven steps methodology, from the data obtention to the case level analysis. The novel methodology proposed in this section focuses on the preparation, identification and analysis steps of the methodology.

Figure 5.6 demonstrates the steps forensic investigators should follow in order to obtain forensically sound data. As described in Section 5.5.4, the GPU must remain powered during the data recovery process and the machine should be secured physically, ensuring an appropriate use of the power supply in order to maintain the data in a live state.

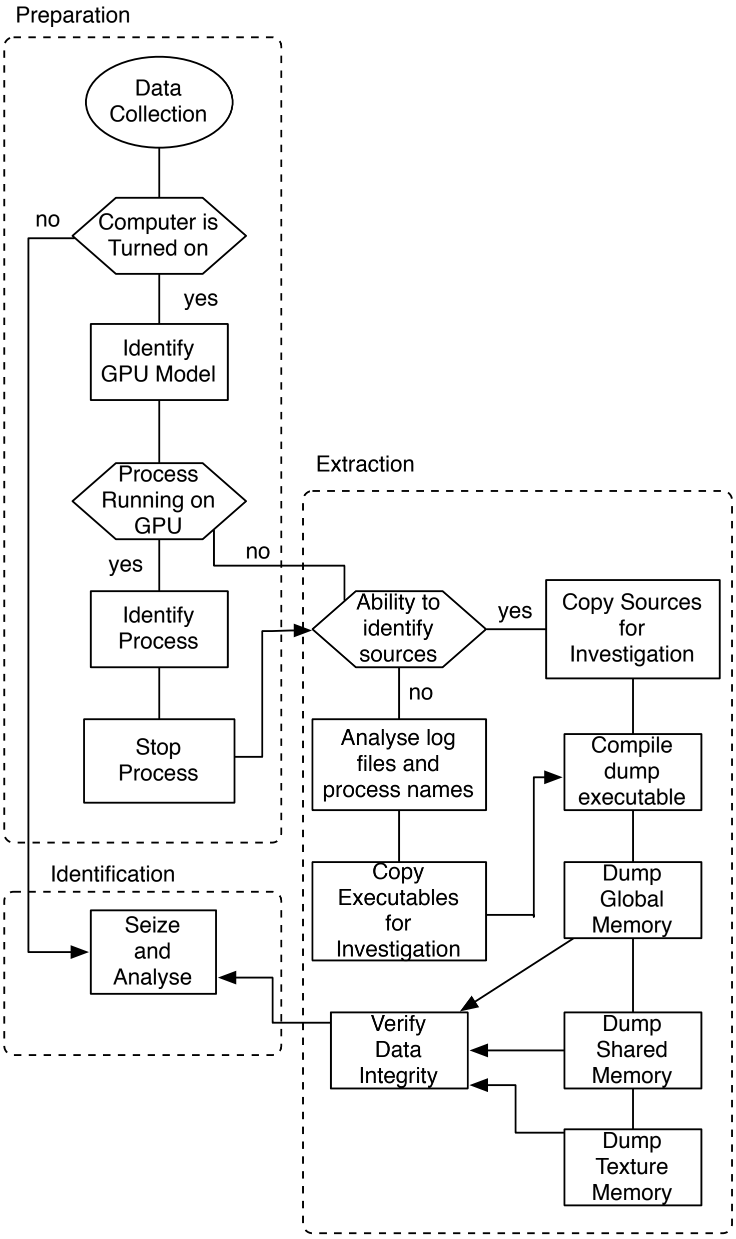


Figure 5.6
Digital Forensic Methodology

Preparation: The preparation phase is the most important part of the methodology and it requires the forensic investigator to clearly identify the model of the GPU encountered, helping the investigator to understand the underlying architecture of the graphics card. This will also help the forensic investigator to identify the environment of graphics card, such as a multi-GPU system, a virtualised environment or a mobile GPU. During this phase the investigator is also required to verify if one or multiple processes are currently running on the GPU. The process identification will further help the forensic investigator to determine what data is being processed, or what computations the GPU has been instructed

Extraction: In the case where a process is running on the GPU, the process in question needs to be identified from the host operating system. Once clearly identified, the process can be stopped, such that the forensic investigator is able to dump the different memories one by one. The forensic investigator shall also investigate the computer hard drive in order to find the sources of the algorithm running on the GPU, or the binary and make copies of them for the analysis phase satisfying any other evidential requirements, such as retaining and comparing file checksums.

Based on the identification of the GPU and associated architecture in the "Preparation" phase, the investigators can now compile an appropriate *"dump"* executable, allowing them to access one or more GPUs in the system to recover Global, Shared and Texture memory. This compilation must take place on another computer, to prevent unintentional interference with potential forensic evidence on the computer under investigation.

Identification: The data retrieved from the running of the executable can be analysed by using standard forensic tools as well as freely available CUDA tools. In order to analyse the process sources for investigation, the full featured *Nsight* editor provided by Nvidia can be used, where simulations, profiling and in depth analysis can be performed [41]. Furthermore, the executables retrieved can be analysed and tested on sample GPUs to generate a behavioural analysis of the software. The executable can be further analysed through decompilation by using *nvdisasm*, *nvidia-debugdump* tools or *cuobjdump* [41]. Analysis of the data dumped from global, shared or texture memory can be analysed using standard memory analysis tools such as the sleuth-kit [164] to retrieve strings in the recovered data. The reverse engineering of the binaries might also lead the forensic investigator to understand the process running on the GPU. Strategies for reverse engineering of CUDA binaries are described in the next section.

The proposed methodology and scenarios are subsequently validated by a study from Zhang *et al.* [165] titled *"Forensically Sound Retrieval and Recovery of Images from GPU Memory*, were the authors demonstrate how to retrieve 24 bit TIFF images from the GPU in a forensically sound manner by using the same techniques as described in Section 5.5 and Section 5.6 . The study demonstrates that pictures viewed in Windows Photo Viewer using GPU acceleration can be retrieved from GPU memory and identified visually. The study also demonstrates that softwares such as the Windows Photo Viewer might store data on the GPU in a different format than the one used on the host, but can be recovered and identified by applying a colour depth map test.

5.6.1 Anti-Forensic Measures

Memory Overwriting The most obvious means of mitigating threats revolving around recovery of data from GPU memory is to overwrite the memory contents, thus preventing techniques such as those discussed from recovering data successfully. By considering the performance implications of erasing GPU memory at time of initialisation, it is clear that for performance-optimised software, there remains motivation to avoid overwriting GPU memory, to reduce the performance overhead of erasing previous data from memory. These methods were proposed in [143] and [144].

Dynamic Parallelism To extend this technique, dynamic parallelism can be used, requiring a *kernel* composed of one or more threads, which itself contains another *kernel*, responsible for the erasure of sensitive data following its access. This technique would reduce the required zeroing time and can be easily implemented by programmers, or added in CUDA libraries, due to running within a second kernel separate from the original code.

Another potential technique is to plant misleading data within GPU memory, using a second kernel, running in parallel with the first, through dynamic parallelism. With two kernels writing, calculating and modifying data in GPU memory, forensic examination would be made significantly more complex, due to the ease with which decoy data may be placed in memory, for the forensic examiner to deal with.

Dead Man's Handle Monitoring The dead man's handle technique can also be used to run two processes on the GPU, which simultaneously monitor each other. In the event of one process being disturbed or terminated such as by someone attempting to terminate the process to begin dumping memory, the remaining process would initiate a memory-erasing routine, using the massively parallel hardware to wipe the entire GPU memory in a matter of seconds.

5.7 Strategies for Reverse Engineering

The reverse engineering of software is the process through which the means of operation of a piece of software can be discovered. The process of reverse engineering software for CPU architectures (such as x86 and x86_64) is well-documented [166] [167]. Despite this, the techniques commonly applied to CPU-based software are not necessarily optimal when dealing with CUDA, which is effectively another layer of abstraction on top of an existing computer, introducing its own nuances, aiding the reverse engineering process.

Reverse engineering of CUDA-based GPU binaries may be necessary, for example, to investigate the extent and impact of a security breach, where suspected malicious code is executed on a GPU [136]. Being able to identify the operations carried out may allow an investigator to establish the potential exposure of confidential information, or to identify what operations were carried out on data previously held on the GPU. Analysis may be necessary for the purpose of creating intrusion detection system signatures or the detection and classification of similar threats in future.

Reverse engineering can also be used to gain an understanding of code being executed. An example would be in software auditing where the use of reverse engineering could ensure that software handling sensitive data is not abusing its privileges. Additionally, as GPU applications often represent a commercial investment of the originating company, reverse engineering can pose a direct threat through the theft of Intellectual Property (IP) [134], as well as serve in the case of a digital forensic investigation.

Reverse engineering is split into two different categories, those of static and dynamic analysis. In static analysis, the source binary is not executed; instead the binary is loaded in a disassembler, in order to be parsed and the machine code is mapped to human readable instructions such as an assembly listing format. This process can be executed, with or without the availability of the source code. Various open source static analysis tools are available, from original Unix operating systems such as the `strings` utility which looks for readable text strings contained within a file, through to specialist Nvidia tools to extract information from CUDA binaries.

In contrast, dynamic reverse engineering involves the execution of the code, typically on a virtualised or debuggable physical environment [168], through the use of a debugger. This permits the flow of the code to be monitored, altered or interrupted by using *breaks* during the execution. These abilities allow to gain an insight into the operation of the code and a better understanding of the different functions of interest. Different variables can also be altered in order to understand the code under scrutiny. Debugging can also be used to identify system calls and functions of interest, such as the CUDA API libraries of the calls made by the software to the CUDA driver.

It is important to note that, within the context of analysis of CUDA code being executed on a GPU, a distinction between the host code and the device code must be made. While host code can be reverse engineered using standard techniques [169], there is currently very limited research into the background information necessary to investigate the CUDA-based code [133] [170].

The ability to investigate the operations of a CUDA-based software is of great importance, specifically given the well-documented use of GPUs for the unauthorised processing of potentially confidential data, as well as for use in compression and cryptography [171]. With the possibility of sensitive data being exposed to massively parallel processing on a daily basis, the ability for digital forensic investigators to respond to security incidents involving the potential compromise of these systems and rapidly investigate the precise nature of the suspect code involved, is critical to presenting an effective incident response.

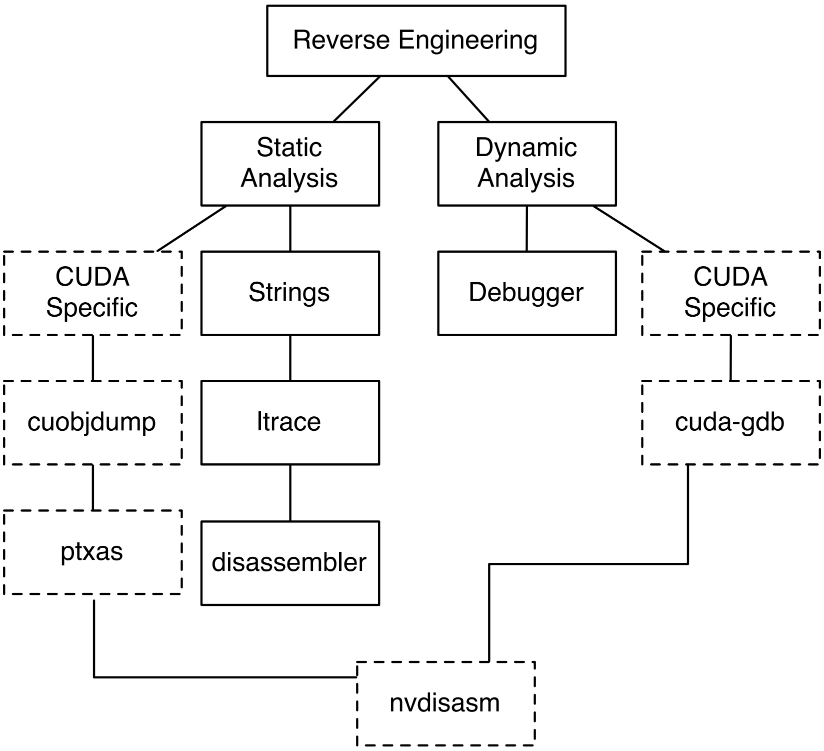


Figure 5.7
Digital Forensic Road Map

The SANS institute recommends static analysis as the first step when performing reverse engineering [172]. When analysing a cubin, different strategies can be adopted as shown in Figure 5.7. Multiple tools are available directly from the CUDA framework (dashed) to reveal the operations carried out by a piece of software. These tools are not specifically designed for reverse engineering, however, many standard / basic Unix utilities can be used to retrieve essential information from the binaries. The next section describes the different steps undertaken in the reverse engineering process of a software, by using the different tools provided by Nvidia and by the standard Unix tools, for static and dynamic analysis, demonstrating, a considerable potential for side-information to be identified simply from these basic utilities.

5.7.1 Static Analysis

The CUDA binary generated by the Nvidia compiler leaks a number of pieces af information potentially damaging for companies concerned about IP, however this information can also be used efficiently to discover the insight of a piece of software written by a malicious user and taking advantage of the massively parallel capabilities of a GPU. This section demonstrates different techniques to obtain valuable information against a simple open-source application and highlight the informations which could be recovered from the host ELF executable.

The first sample application studied in this work demonstrates the reverse engineering of a hash brute-forcing application, which is a task often offloaded to the GPU due to the highly parallel nature of the algorithm. The application accepts two different hash inputs – MD4/5 hashes and the dictionary containing the plain text of different words; both are subsequently transferred to the GPU in order to make a comparison between the hash and the elements stored in the dictionary. If a match is found the result is returned to the host CPU.

```
__cudaparm__Z5bruteforce_dict_a

__cudaparm__Z5bruteforce_md5C_b

__cudaparm__Z5bruteforce2_dict_a

__cudaparm__Z5bruteforce2_md4_b
```

Figure 5.8
Recovering all Function Instances and Arguments

Figure 5.8 demonstrates the output of the *strings* utility run against the ELF file produced by *nvcc*, allowing the forensic investigator to retrieve the name of every function in the code. The *strings* utility tool is designed to display all printable text strings of four or more characters in length when applied to an object, or a binary.

The output demonstrated in Figure 5.8 is the default behaviour of the *nvcc* compiler. This is naturally useful as the original variables and function names are not stripped out from the binary, allowing a forensic investigator to imply the behaviour of the software.

Both CUDA-specific tools and the common CPU reverse engineering tools can be used against the CUDA-produced ELF binary. Figure 5.8 shows the name of the functions used for a simple MD4 and MD5 brute-force application. Two functions have been found (bruteforce, bruteforce2) as well as their required arguments, the dictionary and the md4-5 hash — *dict_a* being the first argument and the *md4-5_b* being the second argument.

Being able to see the names of functions, as well as their incoming parameter definitions, can be useful when carrying out reverse engineering, as it offers a rapid way to gain familiarity with the layout of the code in question. Additionally, presuming the developer of the CUDA software was not taking active steps to avoid this analysis, such as deliberately naming functions in a meaningless manner, there may be considerable side information about the operation of the program able to be determined simply from the naming of functions and parameters.

```
__global__

void bruteforce(char dict, char md5){

...

}
```

Figure 5.9
Original Function Declaration

Figure 5.9 shows the original function declaration in the source code of the example presented in Figure 5.8. As shown previously the figure is named “bruteforce” and receives two distinct arguments “dict” and “md5”.

```
bruteforce

leftSalt

rightSalt

dictionary

password
```

Figure 5.10
Name of the Constant Variables in the Software.

The names of the constant variables can also be gathered in the executable, using the `strings` command. Figure 5.10 shows the constant variable names used in the application. This behaviour is unlike traditionally compiled C applications and may be useful for an investigator wishing to gain a quick understanding of the operation of the code.

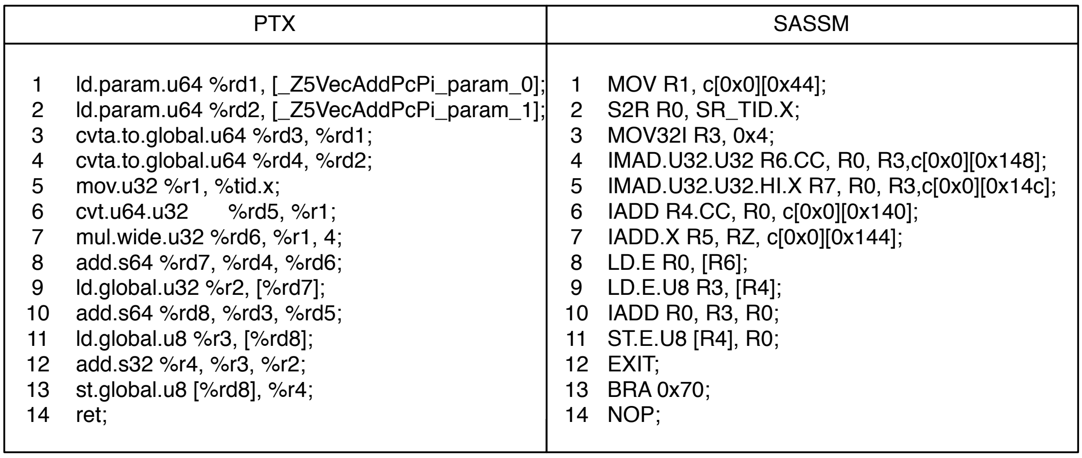
```
checkCUDAError(”Wrong Memory Allocation”)

checkCUDAError(”UnAllocated Variable”)

checkCUDAError(cudaError_t cuError)
```

Figure 5.11
CUDA Error Comments Embedded in the Code.

Figure 5.11 reveals the error defined by the developer of the original code, as well as standard errors defined by the framework. Custom error messages allow the forensic investigator to gain valuable understanding of the behaviour of the different algorithms and functions. By using CUDA specific tools the forensic investigator can retrieve all information initially for the author of code.



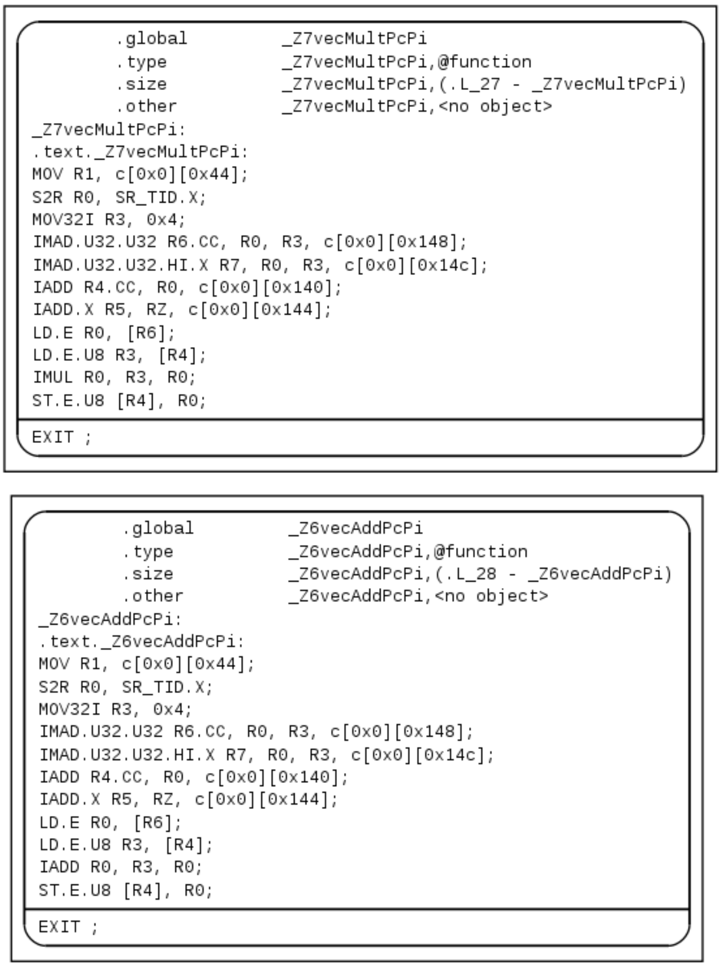
PTX	SASSM
ld.param.u64 %rd1, [_Z5VecAddPcPi_param_0];	MOV R1, c[0x0][0x44];
ld.param.u64 %rd2, [_Z5VecAddPcPi_param_1];	S2R R0, SR_TID.X;
cvta.to.global.u64 %rd3, %rd1;	MOV32I R3, 0x4;
cvta.to.global.u64 %rd4, %rd2;	IMAD.U32.U32 R6.CC, R0, R3,c[0x0][0x148];
mov.u32 %r1, %tid.x;	IMAD.U32.U32.HI.X R7, R0, R3,c[0x0][0x14c];
cvt.u64.u32 %rd5, %r1;	IADD R4.CC, R0, c[0x0][0x140];
mul.wide.u32 %rd6, %r1, 4;	IADD.X R5, RZ, c[0x0][0x144];
add.s64 %rd7, %rd4, %rd6;	LD.E R0, [R6];
ld.global.u32 %r2, [%rd7];	LD.E.U8 R3, [R4];
add.s64 %rd8, %rd3, %rd5;	IADD R0, R3, R0;
ld.global.u8 %r3, [%rd8];	ST.E.U8 [R4], R0;
add.s32 %r4, %r3, %r2;	EXIT;
st.global.u8 [%rd8], %r4;	BRA 0x70;
ret;	NOP;

Figure 5.12
Comparing PTX code and SASSM code

`cuobjdump` The `cuobjdump` tool allows the forensic investigator to dump information directly from the binary, such as the `.section` assembly directives, which can yield valuable information about the different variables, functions and arguments, as well as information about the memory layout of function arguments and the type of storage used for each variable. It is also possible to dump the PTX code, as well as the SASSM code and a standalone Cubin binary.

A second code sample is shown in Figure 5.12, to highlight the difference between the PTX code on the right and the SASSM code on the left. This code demonstrates a simple vector addition between two variables.

The PTX code shows the name of the function and the number of parameters received (line 1, 2). It shows that both are stored in global memory (line 3, 4). The vector addition is then performed. Similar instructions can be seen for the SASSM code, however, as PTX is a higher-level language, it is easier to understand and infer the kernel C source code, as discussed by Dong *et al.* [173]. In particular, PTX contains variable names and function names, conveying semantic information, unlike SASSM which is directly using registers.



```
        .global         _Z7vecMultPcPi
        .type           _Z7vecMultPcPi,@function
        .size           _Z7vecMultPcPi,(.L_27 - _Z7vecMultPcPi)
        .other          _Z7vecMultPcPi,<no object>
_Z7vecMultPcPi:
.text._Z7vecMultPcPi:
MOV R1, c[0x0][0x44];
S2R R0, SR_TID.X;
MOV32I R3, 0x4;
IMAD.U32.U32 R6.CC, R0, R3, c[0x0][0x148];
IMAD.U32.U32.HI.X R7, R0, R3, c[0x0][0x14c];
IADD R4.CC, R0, c[0x0][0x140];
IADD.X R5, RZ, c[0x0][0x144];
LD.E R0, [R6];
LD.E.U8 R3, [R4];
IMUL R0, R3, R0;
ST.E.U8 [R4], R0;
EXIT ;
```

```
        .global         _Z6vecAddPcPi
        .type           _Z6vecAddPcPi,@function
        .size           _Z6vecAddPcPi,(.L_28 - _Z6vecAddPcPi)
        .other          _Z6vecAddPcPi,<no object>
_Z6vecAddPcPi:
.text._Z6vecAddPcPi:
MOV R1, c[0x0][0x44];
S2R R0, SR_TID.X;
MOV32I R3, 0x4;
IMAD.U32.U32 R6.CC, R0, R3, c[0x0][0x148];
IMAD.U32.U32.HI.X R7, R0, R3, c[0x0][0x14c];
IADD R4.CC, R0, c[0x0][0x140];
IADD.X R5, RZ, c[0x0][0x144];
LD.E R0, [R6];
LD.E.U8 R3, [R4];
IADD R0, R3, R0;
ST.E.U8 [R4], R0;
EXIT ;
```

Figure 5.13
Representation of the Different Functions

ptxas The ptxas tool compiles PTX code into GPU-specific micro-code. After retrieving the PTX code from the ELF binary, the forensic investigator can use the ptxas tool to create an object file of the CUDA functions, if they wish to carry out further dynamic analysis and actually execute the code in a controlled environment.

Using the PTX code, it is also possible to add debug information to the object file, as well as line number information, to assist during later stages of the disassembly process.

nvdisasm Nvdisasm is an hybrid static and dynamic analysis tool, which allows a forensic investigator to extract information from the standalone CUDA binary (Cubin) and present this information in a readable format. As such, it is possible to create a flowchart of the different function calls used in the software, as well as their relation to each other as shown in Figure 5.13. The tool is also able to show register usage alongside each executed instruction and can therefore be considered a form of dynamic analysis tool, since it effectively executes the code, in order to establish which registers would be used to store values during execution. This is useful if attempting to establish the contents of a register at a given point, perhaps due to later usage of that register elsewhere in the PTX or disassembled code. In the case of a multi-tenant computing situation, where GPUs are shared between several independent users of a system such as in platform-as-a-service situations, this could be used to carry out pre-execution behavioural checks against CUDA software, to attempt to identify malicious behaviours, such as attempting to dump the contents of all GPU registers, or to allocate the full extent of available GPU memory in a single array. This could be attempts to exploit data remanence of GPUs to extract data processed by a previous user of the shared GPU.

5.7.2 Dynamic Analysis

Dynamic analysis is the second step of our proposed reverse engineering process. This analysis makes use of information previously retrieved from static analysis, such as the function flowchart, the PTX code and the object files generated. The CUDA framework comes with its own debugger, allowing a forensic investigator to run the host CPU code in a pseudo-virtualised environment.

cuda-gdb `Cuda-gdb` is the official CUDA debugger. It allows programmers to run their algorithm on the host system, while being able to use a set of debug instructions to pause and step through code, analyse memory requirements, variable values and thread positions. `Cuda-gdb` is based upon the GNU gdb utility, which is a standard x86 debugger.

While running code through the `cuda-gdb` debugger, the investigator is able to place breakpoints within the complied SASSM code, which is the machine-code that is actually executed on the GPU, in order to better understand the behaviour of the software, as shown in Figure 5.14. The first line places a breakpoint on the GPU kernel "VecAdd" and the third line then executes the main CPU code, thus launching the kernel on the GPU. The breakpoint is then triggered when the kernel loads, allowing for further debug commands to be executed, such as inspecting variable values or stepping through code line-by-line. For example, the investigator can request the debugger provide information on the state of the kernel, such as the number of threads (line 5), when the breakpoint is encountered, the code execution temporarily pauses.

```
(cuda-gdb) b _Z6vecAddPiS_S_

Breakpoint 1 at 0x4026ff

(cuda-gdb) r

(cuda-gdb) cuda kernel block thread

kernel 1, block(1,0,0), thread(1024,0,0)
```

Figure 5.14
Cuda-gdb primitives to analyse the behaviour of the code

Furthermore, the debugger can be used to ascertain more detailed information on nuances of the flow of the software and to complete any missing information that static analysis did not reveal. The PTX code obtained during static analysis can also be used to extend dynamic analysis, by making small adjustments to its assembler-like code, in order to force a specific set of operations to be carried out. The Just-In-Time (JIT) compilation capabilities of the framework are then used to create and execute the PTX code rapidly on the GPU, allowing for rapid modifications and testing cycles to be completed.

As shown in Figure 5.15, by loading both the object file and PTX code at runtime, the forensic investigator can modify the behaviour of the PTX code before it is JIT-compiled and executed, therefore allowing for further dynamic analysis of the code. By making small alterations to the instructions in the PTX code as needed, the JIT compiler then produces a new object file, which is executed, allowing for the rapid modification

Table 5.5
Reverse Engineering Strategy and Extraction Result Summary

	ELF Binary				Tools			
	ELF	CUBIN	PTX	SASSM	cuobjdump	PTXas	nvdisasm	cuda-gdb
Variables	Yes	Yes	No	No	No	No	No	No
Functions Name	Yes	Yes	Yes	Yes	Yes	No	Yes	Yes
Functions Arguments	Yes	Yes	No	No	No	No	No	No
CUDA Errors	Yes	Yes	No	No	No	No	No	No
PTX	Yes	Yes	N/A	No	Yes	No	No	No
SASSM	Yes	Yes	No	N/A	Yes	No	Yes	Yes
CUBIN	Yes	N/A	No	No	Yes	No	Yes	No
Code Logic	No	No	Yes	Yes	No	Yes	Yes	Yes

and testing of modified versions of the CUDA code. This allows different portions of the code to be examined in isolation and can reduce the time needed for dynamic analysis by allowing unneeded portions to be"short-circuited" out in the PTX, thus allowing an investigator to focus only on areas of interest. Being able to rapidly alter and execute modified code is a significant advantage to anyone attempting to reverse engineer the software, although this is only possible if a PTX section is present in the binary. It was found that this is the default behaviour of the nvcc compiler, meaning it is likely that CUDA binaries encountered elsewhere will contain this section, thus aiding dynamic analysis.

```
./vectorAdd

Vector Addition

Using Device 0: ”Tesla K20m”

Module Path <./vectorAdd_kernel64.\ac{PTX}>

loading module: <./vectorAdd_kernel64.\ac{PTX}>
```

Figure 5.15
Just In Time Compilation

The results achieved by the reverse engineering strategy, using the different types of outputs produced by the different formats generated by the nvcc compiler are summarised in Table 5.5 where "Yes" represents the possibility of retrieving data, "No" the impossibility of retrieving data and "N/A" indicates a given test was not applicable. The results demonstrate the possibilities of reverse engineering for forensic investigators and highlight possible intellectual property concerns regarding the format and informations yielded by the different binaries. Perhaps most significantly, these findings highlight that more information about a given CUDA program can be obtained from direct analysis of the binaries, which is not readily exposed using the NVidia tools. It is therefore important for developers to note that a binary which reveals no valuable information when inspected with the NVidia tools may still contain such information, accessible directly through the techniques discussed.

5.7.3 Extended Digital Forensic Methodology

The forensic methodology presented in Section 5.6 is further extended allowing forensic investigators to analyse the source code and the behaviour of an application based on the results presented earlier in Section 5.7, detailing reverse engineering strategies. Figure 5.16 shows the extend methodology, including the reverse engineering step.

The reverse engineering step starts with an incentive to identify the tools offered by the CUDA framework; as new tools may be added in the future, these may offer better reverse engineering support, as well as help forensic investigator understand the behaviour of an algorithm, based on the different informations leaked by the ELF binary. This step is then divided in static and dynamic analysis as recommended by NIST, at this stage the methodology follows the procedure described earlier in Section 5.7. The final steps join the previously described *Identification* step, as this step will allow the forensic investigator to corroborate the information gathered during the *Extraction* and *Reverse Engineering* steps, allowing him to draw conclusions or continue his analysis of the evidence gathered.

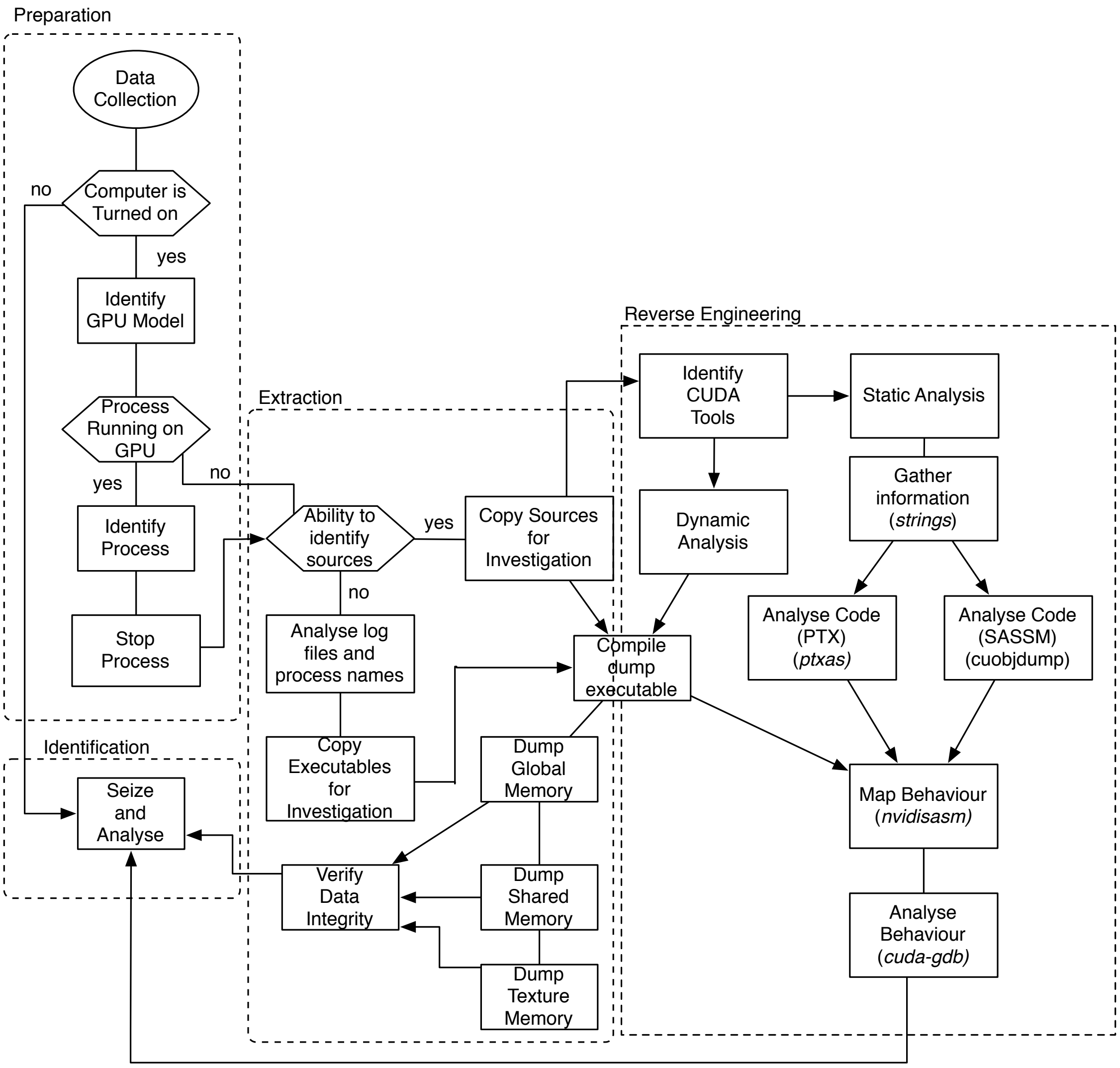


Figure 5.16
Digital Forensic Methodology Extended

5.7.4 Intellectual Property Protection

It should be noted that as with all security measures which attempt to prevent the extraction of intellectual property from compiled software code, the code in question must ultimately be executed on the GPU, meaning that attempts to make reverse engineering and disassembly more difficult are ultimately a form of security through obscurity. It is clear, however, that it is possible to produce binaries which are less easily reverse engineered, thus increasing the time and expertise needed to carry out analysis and gain an understanding of the binary in question.

In the field of CPU-executed code, on standard operating system platforms, obfuscated compilers exist [174], which are designed to produce unintuitive and difficult to reverse engineer binaries, which feature extraneous complexity in their compiled code. These complexities may have an impact on performance, if there is an increase in the number of instructions executed by the processor. For massively parallel processing (such as that carried out by GPU), performance of the code in question is significant, with the same code being executed in parallel across many hundreds of cores simultaneously. Any increase in the number of instructions carried out by the streaming multiprocessor would result in a significant reduction in performance across all cores. For a higher number of simultaneous threads being executed, the number of wasted clock cycles added by the obfuscation would increase, since each stream processor runs the same code and it would ultimately lead to more divergence of the threads.

While it would be possible to wrap the CUDA host ELF binary to make it more difficult to carry out static analysis, it is worth considering that ultimately the GPU must execute the Cubin binary. Such obfuscation, while not impacting on performance of the calculations, would be relatively easily overcome by using dynamic analysis. By using the `cuda-gdb` debugger to carry out dynamic analysis, it would be possible to step through execution of the binary until it had suitably decrypted or de-obfuscated the Cubin or the PTX code and then extract it for direct analysis.

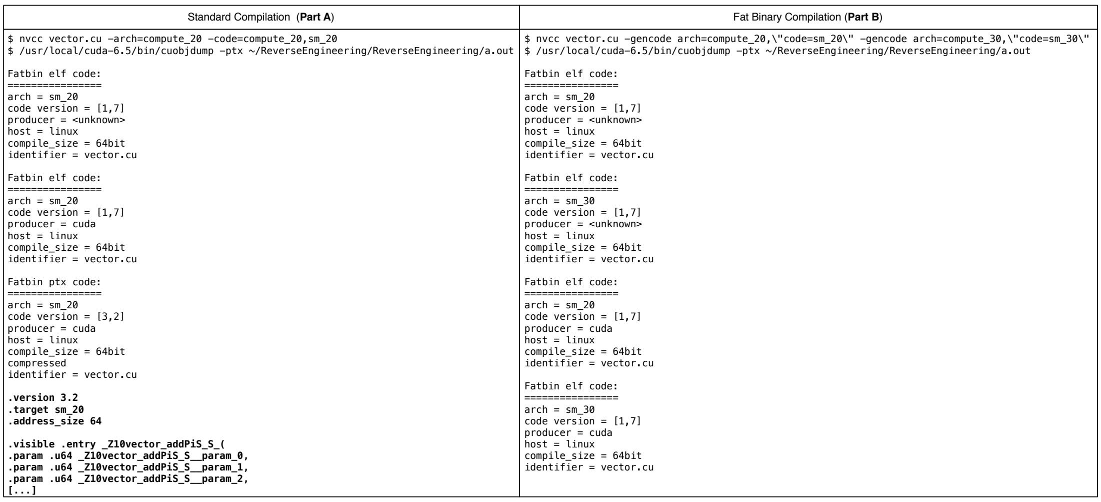
It is therefore recommend that, as per the demonstration of the reverse engineering process of a CUDA binary, a developer may attempt to hinder reverse engineering through the use of meaningless and repeated variable names and functions. This strategy is similar to that used in Java by `dexguard` and `proguard` [175], yet proves effective in slowing down static analysis of the code in question. Using this technique would offer some protection of both the Cubin and PTX code from simple static analysis, by posing a stumbling block to easy visual inspection of the code. Alternatively, a tool like `strip` could potentially be modified to support CUDA, allowing for the removal of readable strings from compiled code as demonstrated in Figure 5.17, `Z53456789` represents the obfuscated function name and `234` represents the first argument name of the function.

```
__cudaparm__Z53456789_234_a

__cudaparm__Z53456789_236_b
```

Figure 5.17
Obfuscated Functions and Arguments Names

Additionally, if the CUDA code is able to be targeted to a single GPU architecture and does not need forward compatibility, the PTX code present in the CUDA binary can be removed when the binary code for the architecture is present and it is found that the resulting modified binary still executed correctly on the target platform, despite the PTX listing being removed. If it is necessary for code to be executable on many different GPUs, a "fat" binary can be created for each GPU architecture and platform such as shown in Figure 5.18.



Standard Compilation (**Part A**)

```
$ nvcc vector.cu -arch=compute_20 -code=compute_20,sm_20
$ /usr/local/cuda-6.5/bin/cuobjdump -ptx ~/ReverseEngineering/ReverseEngineering/a.out

Fatbin elf code:
================
arch = sm_20
code version = [1,7]
producer = <unknown>
host = linux
compile_size = 64bit
identifier = vector.cu

Fatbin elf code:
================
arch = sm_20
code version = [1,7]
producer = cuda
host = linux
compile_size = 64bit
identifier = vector.cu

Fatbin ptx code:
================
arch = sm_20
code version = [3,2]
producer = cuda
host = linux
compile_size = 64bit
compressed
identifier = vector.cu

.version 3.2
.target sm_20
.address_size 64

.visible .entry _Z10vector_addPiS_S_(
.param .u64 _Z10vector_addPiS_S__param_0,
.param .u64 _Z10vector_addPiS_S__param_1,
.param .u64 _Z10vector_addPiS_S__param_2,
[...]
```

Fat Binary Compilation (**Part B**)

```
$ nvcc vector.cu -gencode arch=compute_20,\"code=sm_20\" -gencode arch=compute_30,\"code=sm_30\"
$ /usr/local/cuda-6.5/bin/cuobjdump -ptx ~/ReverseEngineering/ReverseEngineering/a.out

Fatbin elf code:
================
arch = sm_20
code version = [1,7]
producer = <unknown>
host = linux
compile_size = 64bit
identifier = vector.cu

Fatbin elf code:
================
arch = sm_30
code version = [1,7]
producer = <unknown>
host = linux
compile_size = 64bit
identifier = vector.cu

Fatbin elf code:
================
arch = sm_20
code version = [1,7]
producer = cuda
host = linux
compile_size = 64bit
identifier = vector.cu

Fatbin elf code:
================
arch = sm_30
code version = [1,7]
producer = cuda
host = linux
compile_size = 64bit
identifier = vector.cu
```

Figure 5.18
Comparison of Standard and Fat Compilation Process.

The first command executed in Figure 5.18 (Part A) shows the process to compile a simple CUDA code file, using the default compiler settings. The second command shows the retrieval of the PTX code from the binary file resulting from the compilation, using `cuobjdump`. As expected, it is possible to retrieve the PTX code from the binary (Bottom of Part A). During the "fat" binary compilation, as shown in Figure 5.18 (Part B), the first line shows the command required to compile a "fat" binary using nvcc. The second line shows that, upon inspection by `cuobjdump`, the ELF binary no longer contains a PTX section. While this dramatically increases the size of the ELF binary, as it contains the SASSM code for each target architecture, it hinders reverse engineering by preventing the use of the Just-In-Time technique described earlier.

5.8 Summary

Data remanence is the residual representation of data stored in memory after having been erased, or de-allocated [176]. Chapter 3 and 4 highlighted the different techniques for pattern matching on GPUs in different contexts and for different critical applications. This chapter evaluated data remanence in GPU memory on a range of devices including high end devices and mobile devices. Moreover a methodology for digital forensics was proposed and evaluated, as well as different strategies for reverse engineering of CUDA applications.

Different scenarios were established, in order to store data efficiently in global, shared and texture memory on three GPUs. The remanent data was then recovered using different techniques, allowing to avoid the compromise of these data. It was demonstrated that legitimate data remains in memory after being freed by the host, moreover, these data were exposed to malicious users and could be retrieved using different procedures.

Different anti-forensic measures were highlighted in order to prevent malicious users to access data after the termination of the main process. A methodology for digital forensics was also proposed based on the evaluation of the different scenarios, allowing forensic investigators to understand GPU architecture and memory types, while taking advantage of the techniques proposed to gather evidence of crimes committed on GPUs.

Strategies for reverse engineering of CUDA software were also detailed and analysed. Both static and dynamic reverse engineering techniques were considered and applied to different CUDA executables. Finally the digital forensic methodology proposed in Section 5.6 was augmented with a reverse engineering section strengthening the overall methodology and giving the possibility to forensic investigator to gather evidence via the CUDA executables and determine the ability and the potential of an executable.

With the increasing GPU market, the underlying architecture of the hardware might be subject to changes and the methodology might require changes in order to accommodate forensic investigators, however with the increasing amount of data processed on GPUs and with the current generalisations of GPGPU processing it is believed that forensic investigators will soon be confronted with crimes committed using massively parallel devices. This extended methodology offers them a step by step methodology.

Chapter 6

Conclusions and Future Work

6.1 Conclusion

Graphics Processing Units are able to provide the user with an off-the-shelf massively parallel device enabling significant data processing and throughput increases over a range of applications as demonstrated in this thesis. The work carried out highlighted that in order to take advantage of GPUs the data offloaded for computation requires careful consideration, and software optimisation in order to take full advantage of the hardware and of the different memory hierarchies available. The availability and popularity of massively parallel devices in recent years has led to unprecedented levels of data processing techniques allowing to further advance the fields of pattern matching.

This thesis presented research in single and multi pattern matching techniques, as well as digital forensic and reverse engineering strategies. By using the GPU as a mean of offloading data processing for single and multi pattern matching this work demonstrated the potential of off-the-shelf GPUs within different areas such as IDS and DNA sequencing. Furthermore, the importance of data compression due to the limited amount of memory has been highlighted. To this end, different algorithms have been implemented. The risk of data remanence and data leakages on GPUs have also been exposed. Different strategies such as anti-forensic and anti-reverse engineering techniques have been proposed, as well as an extended digital forensic methodology, for digital forensic investigators facing GPU malware cases, or crime committed with the help of GPUs.

Chapter 3 provides innovative parallel implementations for the two most popular single pattern matching algorithms, the KMP and the BMH algorithm. Both demonstrated improvement over their sequential versions. It was shown that by taking advantage of the different memory layouts available on the GPU, as well as making use of optimisations techniques such as loop unrolling, the KMP and BMH algorithms were able to demonstrate throughput in the order of a gigabit per second. It was however demonstrated that these algorithms were not performing well with an increasing number of patterns. In order to overcome the limitations of single-pattern matching algorithms a novel implementation of the failure-less Aho-Corasick algorithm using state tables was made, allowing each thread to match one single character, and be discarded at mismatch, increasing the throughput. The algorithm led to increased storage requirements due to the empty cells of the state table. However the use of compact state tables reduced the initial space requirements. The algorithm was conceived to scale in order to be used for log management of industrial systems and was developed with these features in mind.

Finally it was shown that single pattern matching algorithms were outperforming multi-pattern matching algorithms when matching one single pattern, demonstrating over 19 Gbps for the KMP and over 24 Gbps for the BMH, however, the single-pattern matching algorithms performances degraded with the increasing number of patterns. It was further noted that by using different memory types, single and multi-pattern matching were able to improve their overall matching performances. The failure-less Aho-Corasick algorithm presented using reduced state table increased its original throughput from 10 Gbps to 20 Gbps, over 1000 patterns on a single GPU by taking advantage of texture memory.

This chapter has attained multiple achievements; the main contributions are summarised as follows:

- The implementation of two high performance single pattern matching algorithms, utilising the ubiquitous GPU for offloading string matching operations. The implementation is freely available as a single-pattern matching library.
- An innovative GPU Log Processing library is developed for high performance passive log monitoring, easing the analysis of faults, and intrusions, within industrial systems. The algorithm is scalable allowing to process increasing amount of data at high rates.

In Chapter 4 a novel Highly Efficient Parallel Failure-less Aho-Corasick algorithm is presented. This novel algorithm aims at reducing the size of the trie stored in memory, as GPU memory is a scarce resource. The novel algorithm makes use of different algorithmic features to reduce and compress the data represented in the trie. To this aim, the nodes of the first version presented are stored in a 1D array, constructed in a breath-first construction fashion, each node only stores a bitmap of its children and an offset allowing to reduce the space complexity of each nodes to a minimum. Furthermore, the last nodes of the trie and similar suffixes are merged together. The first version of the HEPFAC offers excellent data compression performance, however, the initial design demonstrates a trade-off between space complexity and throughput. To this end, a second version of the algorithm is presented. In this version the memory scheme made use of a 2D breadth-first constructed trie. This core modification allowed the trie to be stored in texture memory, making use of the cache and allowing un-coalesced memory accesses. This novelty allowed to increase the throughput 13 folds when matching 1000 patterns over an alphabet of 52 characters. Moreover, the algorithm was analysed against different number of alphabets, demonstrating its behaviour for different fields ranging from medical (DNA Matching) to security applications (IDS). With the number of GPU models and architectures, and the difficulties of comparing algorithms running on different hardware, the performances were evaluated on the Amazon cloud, allowing other research institutions to evaluate the performance of their implementation against the work presented in this thesis and further advance the field of research.

Finally it was demonstrated that the two versions of the HEPFAC algorithm outperformed the state of the art storage requirements due to the proposed reduction scheme. It was also demonstrated that global memory allowed to increase the overall throughput of the algorithm against different alphabets, compared to the CPU. However it presented significant drawbacks in terms of coalesced memory accesses and divergence leading to throughput degradation and it exhibited 1 Gbps throughput. However it was demonstrated that by modifying the core of the HEPFAC algorithm, enhancing its two-dimensional locality and taking advantage of texture memory, an improvement of 10 fold over the first version of the HEPFAC could be achieved.

The main contribution of this Chapter is summarised as follows:

- A novel Highly Efficient Parallel Failure-less Aho-Corasick algorithm is introduced for high performance pattern matching with a reduced space complexity. The memory scheme presented is modular and can be used within a wide range of applications (medical, IDS, anti-viruse, etc).

Chapter 5 highlights potential security concerns of offloading data to the GPU in order to profit from their computational power, as proposed in Chapter 3 and Chapter 4. In this chapter different scenarios are presented to demonstrate the dangers of data remanence on GPUs. The scenarios are executed on different memory architectures, including global, shared and texture memory, on a range of GPUs, including, mobile, consumer, and high-end devices. It was demonstrated that a malicious user could implement different attacks in order to retrieve data stored on the GPU against the will of the legitimate user. To this end, anti-forensic measures were presented, allowing legitimate users to protect confidential data such as DNA from being leaked by the GPU after being used. Moreover, a novel methodology for data acquisition is presented, allowing digital forensic investigator to retrieve data stored in the different memory offered by the GPU. Furthermore, different strategies for reverse engineering are explored, in order for the user to understand how to preserve their IP. It is also demonstrated that multiple features are leaked by the binary generated by the CUDA framework. Reverse engineering techniques for CUDA binaries are also explored, making use of specific static and dynamic analysis. The forensic methodology is further enhanced by the addition of a novel reverse engineering methodology for CUDA binaries, allowing digital forensic investigators to examine the memory of the GPUs.

Finally the evaluation of the different memories demonstrated data remanence in GPU memory and allowed the design of a digital forensic methodology. Furthermore, the evaluation and analysis of the CUDA binaries permitted to highlight intellectual property concerns and data leakage through the binary, leading to the conception of the reverse engineering methodology intended for digital forensic investigators.

The main contribution of this Chapter is summarised as follows:

- A novel Digital Forensic and Reverse Engineering methodology is presented, highlighting security and privacy concern on off-the-shelf hardware. The methodology allows investigators, to gather and analyse remanent data stored in Graphics Processing Unit memory.

Within this thesis the potential of off-the-shelf GPUs for pattern matching has been demonstrated, optimisation methods for single and multi-pattern matching have been detailed and analysed, and security and privacy concerns on GPU hardware have been exposed along with countermeasures. Offloading pattern matching to the GPU is expensive, through consumption of computational power; through memory requirements; through security and privacy, however if done right the GPU can efficiently demonstrate high computational power, and increase the overall throughput of sequential single and multi-pattern matching applications. This thesis has described approaches to fulfil those aims.

6.2 Future Work

Further to the methods proposed in this work, additional research steps should be undertaken as follows.

6.2.1 Data Transfer

This thesis did not consider data transfer from the network to the GPU but rather focused on algorithmic theory for massively parallel devices and optimisations, in order to increase the throughput of a given single or multi-pattern algorithm. In other words it was assumed that the data acquisition was made transparently regarding the problems one could encounter transferring data from the network to the user-land where the CUDA framework operates. This limits the applicability of the results presented within the thesis to a real-life application for two main reasons.

Firstly, the data transferred to the GPU is static and has been generated using a synthetic data generator. Synthetic data allowed more control, and is used in industry for testing, however in real-life application one could encounter unknown parameters, such as a Denial of Service (DOS) attack. These scenarios could possibly limit the throughput of the IDS as discussed by Tuck *et al.* [177].

Secondly, the data offloaded to the GPUs are serialised and transferred via a single or multiple buffers, meaning, that the buffer sizes have been chosen to accommodate the synthetic files. For example, if network data were acquired from a network interface card and fed to the GPU, empty buffers, or partially full buffers would inevitably lead to performance degradation as there is no scheduling mechanism in place to alleviate this phenomenon as discussed by Sagman *et al.* [178].

6.2.2 Single Pattern Matching

Further work in the memory representation of both, the KMP and BMH should be carried out. Serialising the failure table, the bad character table and the pattern could enable different memory representation schemes and lead to throughput improvements when matching one or more patterns. Further investigation on the maximum length of the patterns should also be carried out. This could allow the users to fix a depth limit for the patterns and increase the data transfer from the Host to Device, as well as increasing the overall algorithms throughput. Other single pattern matching should be parallelised such as the Rabin–Karp [179] and the Bitap algorithm [180] in order to draw throughput comparisons between the different algorithms.

6.2.3 Highly Efficient Parallel Failure-less Aho-Corasick

The reduction algorithm demonstrated in this work merges sequentially similar branches in the trie. Although effective, a better reduction algorithm should be investigated in order to accommodate larger number of patterns while decreasing the time required for the reduction. Moreover, a reduction algorithm for the node expansion allegory should be investigated. The memory scheme of the node expansion allegory translates the 1D nodes into 2D nodes although, a number of cells will remain empty due to the nature of ASCII alphabet (as some symbols will not be used). Therefore a sparse matrix algorithm should be investigated in order to further reduce the storage space required. A convenient compressed row storage algorithm is discussed by Vasquez *et al.* [181]. This algorithm should allow to reduce the number of empty cells in the bitmap allegory discussed in Section 4.6 and further reduce the overall size of the bitmapped tree. This would however increase the computation required to look through the 2D node and may have a negative impact on the throughput of the algorithm.

6.2.4 Pattern Matching Prototyping and Deployment

Future work should consider the prototyping and deployment of the algorithms in a real time scenario, such as an intrusion detection system. By solving a real world problem in a data network, and attempting to detect intrusions, some of the previous suggestions of future work will be addressed. Moreover, analysing a data link will allow to detect different algorithm behaviours and identify unexpected actions undertaken. Furthermore, aspects of the deployment such as mismatches, unexpected actions, throughput increase, degradation and security and privacy will lead to valuable outcome, and lessons learnt will be extremely valuable to researchers, academia and future stakeholders.

6.2.5 Digital Forensic and Reverse Engineering

The ability to retrieve residual information of data stored in memory, and analyse massively parallel code is important. Further work into the different memory on a GPU and the type of data that could be retrieved should be carried out. Furthermore Windows, Mac, and Linux binaries, as well as newer versions of the CUDA framework should be analysed, in order to identify future data leaks and preserve privacy. Future work on an alternative method to gather residual information should be a priority, in order to ensure data preservation, and avoid data contamination. The CUDA framework should be reverse engineered, as well as Nvidia drivers in order to understand the data storage mechanism and the transfer component from the HtoD, in order to help future digital forensic investigations and remain up to date.

Bibliography

[1] D. Evans, "The internet of things how the next evolution of the internet is changing everything," tech. rep., Cisco, 2015. Accessed: 2016-01-01.

[2] M. A. Jamshed, J. Lee, S. Moon, I. Yun, D. Kim, S. Lee, Y. Yi, and K. Park, "Kargus: a highly-scalable software-based intrusion detection system," in *Proceedings of the 2012 ACM conference on Computer and communications security*, pp. 317–328, ACM, 2012.

[3] A. Gupta, S. Dutta, and V. Mangla, "Malware attacks on smartphones and their classification based detection," in *Contemporary Computing*, pp. 242–253, Springer, 2011.

[4] R. I. J. Bayne, E. Ferguson, "OpenCL acceleration of digital forensic methods," *Proceedings of CyberForensics*, 2014.

[5] M. Drory Retwitzer, M. Polishchuk, E. Churkin, I. Kifer, Z. Yakhini, and D. Barash, "Rnapattmatch: a web server for rna sequence/structure motif detection based on pattern matching with flexible gaps," *Nucleic Acids Research*, 2015.

[6] E. Sitaridi and K. Ross, "GPU-accelerated string matching for database applications," *The VLDB Journal*, pp. 1–22, 2015.

[7] S. A. Manavski and G. Valle, "Cuda compatible GPU cards as efficient hardware accelerators for smith-waterman sequence alignment," *BMC Bioinformatics*, vol. 9, no. 2, pp. 1–9, 2008.

[8] R. Sommer, V. Paxson, and N. Weaver, "An architecture for exploiting multi-core processors to parallelize network intrusion prevention," *Concurrency and Computation: Practice and Experience*, vol. 21, no. 10, pp. 1255–1279, 2009.

[9] N. Jacob and C. Brodley, "Offloading ids computation to the GPU," in *Computer Security Applications Conference, 2006. ACSAC '06. 22nd Annual*, pp. 371–380, Dec 2006.

[10] G. Scott, M. England, K. Melkowski, Z. Fields, and D. T. Anderson, "GPU-based postgreSQL extensions for scalable high-throughput pattern matching," in *Pattern Recognition (ICPR), 2014 22nd International Conference on*, pp. 1880–1885, IEEE, 2014.

[11] A. Kalia, D. Zhou, M. Kaminsky, and D. G. Andersen, "Raising the bar for using GPUs in software packet processing," pp. 409–423, May 2015.

[12] X. Bellekens, I. Andonovic, R. Atkinson, C. Renfrew, and T. Kirkham, "Investigation of GPU-based pattern matching," in *The 14th Annual Post Graduate Symposium on the Convergence of Telecommunications, Networking and Broadcasting (PGNet2013)*, 2013.

[13] X. J. A. Bellekens, C. Tachtatzis, R. C. Atkinson, C. Renfrew, and T. Kirkham, "Glop: Enabling massively parallel incident response through GPU log processing," in *Proceedings of the 7th International Conference on Security of Information and Networks*, SIN '14, (New York, NY, USA), pp. 295:295–295:301, ACM, 2014.

[14] X. J. A. Bellekens, C. Tachtatzis, R. C. Atkinson, C. Renfrew, and T. Kirkham, "A highly-efficient memory-compression scheme for GPU-accelerated intrusion detection systems," in *Proceedings of the 7th International Conference on Security of Information and Networks*, SIN '14, (New York, NY, USA), pp. 302:302–302:309, ACM, 2014.

[15] X. Bellekens, G. Paul, J. M. Irvine, C. Tachtatzis, R. C. Atkinson, T. Kirkham, and C. Renfrew, "Data remanence and digital forensic investigation for cuda graphics processing units," in *Integrated Network Management (IM), 2015 IFIP/IEEE International Symposium on*, pp. 1345–1350, May 2015.

[16] J. W. Kim, E. Kim, and K. Park, "Fast matching method for dna sequences," in *Combinatorics, Algorithms, Probabilistic and Experimental Methodologies*, pp. 271–281, Springer, 2007.

[17] A. Apostolico, *Pattern Matching Algorithms*. Oxford University Press, USA, 1997.

[18] J. James H. Morris and V. R. Pratt, "A linear pattern-matching algorithm," *Technical Report 40*, 1970.

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

[20] G. Navarro and M. Raffinot, *Flexible Pattern Matching in Strings: Practical On-Line Search Algorithms for Texts and Biological Sequences*. Cambridge University Press, 2002.

[21] R. S. Boyer and J. S. Moore, "A fast string searching algorithm," *Commun. ACM*, vol. 20, pp. 762–772, Oct. 1977.

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

[23] M. Crochemore, A. Czumaj, L. Gąsieniec, T. Lecroq, W. Plandowski, and W. Rytter, "Fast practical multi-pattern matching," *Information Processing Letters*, vol. 71, no. 3–4, pp. 107 – 113, 1999.

[24] M. O. Rabin and D. Scott, "Finite automata and their decision problems," *IBM J. Res. Dev.*, vol. 3, pp. 114–125, Apr. 1959.

[25] D. Gusfield, *Algorithms on Strings, Trees and Sequences*. Cambridge University Press, 1997. Cambridge Books Online.

[26] J. Daciuk, B. W. Watson, S. Mihov, and R. E. Watson, “Incremental construction of minimal acyclic finite-state automata,” *Comput. Linguist.*, vol. 26, pp. 3–16, Mar. 2000.

[27] A. Blumer, J. Blumer, D. Haussler, A. Ehrenfeucht, M. Chen, and J. Seiferas, “The smallest automation recognizing the subwords of a text,” *Theoretical Computer Science*, vol. 40, no. 0, pp. 31 – 55, 1985. Eleventh International Colloquium on Automata, Languages and Programming.

[28] W. B. W, “A taxonomy of finite automata minimization algorithms,” *Eindhoven University of Technology, Department of Mathematics and Computing Science Computing Science Section*, 1993.

[29] B. W. Watson, *Taxonomies and toolkits of regular language algorithms*. Eindhoven University of Technology The Netherlands, 1995.

[30] K. Fredriksson, “Succinct backward-dawg-matching,” *J. Exp. Algorithmics*, vol. 13, pp. 8:1.8–8:1.26, Feb. 2009.

[31] A. V. Aho and M. J. Corasick, “Efficient string matching: An aid to bibliographic search,” *Commun. ACM*, vol. 18, pp. 333–340, June 1975.

[32] O. AitMous, F. Bassino, and C. Nicaud, *Combinatorial Pattern Matching: 23rd Annual Symposium, CPM 2012, Helsinki, Finland, July 3-5, 2012. Proceedings*, ch. An Efficient Linear Pseudo-minimization Algorithm for Aho-Corasick Automata, pp. 110–123. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012.

[33] Y. Dandass, S. Burgess, M. Lawrence, and S. Bridges, “Accelerating string set matching in FPGA hardware for bioinformatics research,” *BMC Bioinformatics*, vol. 9, no. 1, p. 197, 2008.

[34] A. Tanenbaum and H. Bos, *Modern Operating Systems*. Pearson Education, 2014.

[35] G. Moore, *Cramming More Components Onto Integrated Circuits*. McGraw-Hill, 1965.

[36] L. J. Flynn, "Intel halts development of 2 new microprocessors." http://www.nytimes.com/2004/05/08/business/intel-halts-development-of-2-new-microprocessors.html. Accessed: 2015-05-30.

[37] Y. Zhou, *Hardware Acceleration For Power Efficient Deep Packet Inspection*. PhD thesis, Dublin City University, August 2012.

[38] M. Flynn, "Some computer organizations and their effectiveness," *Computers, IEEE Transactions on*, vol. C-21, pp. 948–960, Sept 1972.

[39] B. Wang, *Mitigating GPU Memory Divergence for Data-Intensive Applications*. PhD thesis, Auburn University, 2015.

[40] Y. Kreinin, "SIMD < SIMT < SMT: parallelism in NVIDIA GPUs." http://yosefk.com/blog/simd-simt-smt-parallelism-in-nvidia-gpus.html. Accessed: 2015-05-30.

[41] Nvidia, "OpenCL programming for the CUDA architecture." http://www.nvidia.com/content/cudazone/download/opencl/nvidia_opencl_programmingoverview.pdf. Accessed: 2015-05-30.

[42] N. B. Lakshminarayana and H. Kim, "Effect of instruction fetch and memory scheduling on GPU performance," *Workshop on Language, Compiler, and Architecture Support for GPGPU*, pp. 1–10, 2010.

[43] N. Wilt, *The CUDA Handbook: A Comprehensive Guide to GPU Programming*. Pearson Education, 2013.

[44] C. Li, Y. Yang, Z. Lin, and H. Zhou, "Automatic data placement into GPU on-chip memory resources," in *Code Generation and Optimization (CGO), 2015 IEEE/ACM International Symposium on*, pp. 23–33, Feb 2015.

[45] M. Akhloufi, *Applied Vision and Robotics Workshop 2012*. Moulay Akhloufi.

[46] S. Venkatasubramanian and R. W. Vuduc, “Tuned and wildly asynchronous stencil kernels for hybrid CPU/GPU systems,” in *Proceedings of the 23rd International Conference on Supercomputing*, ICS ’09, (New York, NY, USA), pp. 244–255, ACM, 2009.

[47] D. Kirk *et al.*, “NVIDIA CUDA software and GPU parallel computing architecture,” in *ISMM*, vol. 7, pp. 103–104, 2007.

[48] J. Peng and H. Chen, “Cugrep: A GPU-based high performance multi-string matching system,” in *Future Computer and Communication (ICFCC), 2010 2nd International Conference on*, vol. 1, pp. V1–77–V1–81, May 2010.

[49] M. Schatz and C. Trapnell, “Fast exact string matching on the GPU,” *Center for Bioinformatics and Computational Biology*, 2007.

[50] H. Li, B. Ni, M.-H. Wong, and K.-S. Leung, “A fast CUDA implementation of agrep algorithm for approximate nucleotide sequence matching,” in *Application Specific Processors (SASP), 2011 IEEE 9th Symposium on*, pp. 74–77, June 2011.

[51] K.-J. Lin, Y.-H. Huang, and C.-Y. Lin, “Efficient parallel knuth-morris-pratt algorithm for multi-GPUs with CUDA,” in *Advances in Intelligent Systems and Applications - Volume 2* (J.-S. Pan, C.-N. Yang, and C.-C. Lin, eds.), vol. 21 of *Smart Innovation, Systems and Technologies*, pp. 543–552, Springer Berlin Heidelberg, 2013.

[52] C.-L. Hung, H.-H. Wang, C.-Y. Chang, and C.-Y. Lin, “Efficient packet pattern matching for gigabit network intrusion detection using GPUs,” in *High Performance Computing and Communication 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS), 2012 IEEE 14th International Conference on*, pp. 1612–1617, June 2012.

[53] A. Tumeo, O. Villa, and D. Sciuto, “Efficient pattern matching on GPUs for intrusion detection systems,” in *Proceedings of the 7th ACM International Conference on Computing Frontiers*, CF ’10, (New York, NY, USA), pp. 87–88, ACM, 2010.

[54] L. Marziale, G. G. R. III, and V. Roussev, "Massive threading: Using {GPUs} to increase the performance of digital forensics tools," *Digital Investigation*, vol. 4, Supplement, no. 0, pp. 73 – 81, 2007.

[55] C.-H. Lin, S.-Y. Tsai, C.-H. Liu, S.-C. Chang, and J.-M. Shyu, "Accelerating string matching using multi-threaded algorithm on GPU," in *Global Telecommunications Conference (GLOBECOM 2010), 2010 IEEE*, pp. 1–5, Dec 2010.

[56] C.-H. Lin, C.-H. Liu, L.-S. Chien, and S.-C. Chang, "Accelerating pattern matching using a novel parallel algorithm on GPUs," *Computers, IEEE Transactions on*, vol. 62, pp. 1906–1916, Oct 2013.

[57] C.-H. Lin, C.-H. Liu, L.-S. Chien, S.-C. Chang, and W.-K. Hon, "PFAC library: GPU-based string matching algorithm." http://on-demand.gputechconf.com/gtc/2012/presentations/S0054-PFAC-Library-GPU-Based-String-Matching-Algorithm.pdf. Accessed: 2015-05-30.

[58] C. J. dos Santos Brito, L. G. Costa, J. M. X. Teixeira, and V. Teichrieb, "Operações paralelas sobre bases massivas de strings,"

[59] K.-J. Lin, Y.-H. Huang, and C.-Y. Lin, "Efficient parallel knuth-morris-pratt algorithm for multi-GPUs with CUDA," in *Advances in Intelligent Systems and Applications - Volume 2* (J.-S. Pan, C.-N. Yang, and C.-C. Lin, eds.), vol. 21 of *Smart Innovation, Systems and Technologies*, pp. 543–552, Springer Berlin Heidelberg, 2013.

[60] K. Dohi, K. Benkrid, C. Ling, T. Hamada, and Y. Shibata, "Highly efficient mapping of the smith-waterman algorithm on CUDA-compatible GPUs," in *Application-specific Systems Architectures and Processors (ASAP), 2010 21st IEEE International Conference on*, pp. 29–36, July 2010.

[61] S. Ryoo, C. I. Rodrigues, S. S. Stone, S. S. Baghsorkhi, S.-Z. Ueng, J. A. Stratton, and W.-m. W. Hwu, "Program optimization space pruning for a multithreaded

GPU," in *Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '08, (New York, NY, USA), pp. 195–204, ACM, 2008.

[62] C. S. Kouzinopoulos and K. G. Margaritis, "String matching on a multicore GPU using CUDA," in *Informatics, 2009. PCI '09. 13th Panhellenic Conference on*, pp. 14–18, Sept 2009.

[63] M. H. Sosnick and W. T. Hsu, "Implementing a finite difference-based real-time sound synthesizer using GPUs.," in *NIME*, pp. 264–267, 2011.

[64] Y. Huang, X. Pan, Y. Gao, and G. Cai, "A fast pattern matching algorithm for biological sequences," in *Bioinformatics and Biomedical Engineering, 2008. ICBBE 2008. The 2nd International Conference on*, pp. 608–611, IEEE, 2008.

[65] Z. Qu and X. Huang, "The improving pattern matching algorithm of intrusion detection," *Procedia Engineering*, vol. 15, pp. 2841–2846, 2011.

[66] K. Fredriksson, "Fast algorithms for string matching with and without swaps. unpublished manuscript, http://www.cs.uku.fi/ fredriks/pub/ papers/sm-w-swaps.pdf," 2000.

[67] J. Zhou, H. An, X. Li, M. Xu, and W. Zhou, "Implementation of string match algorithm BMH on GPU using CUDA," *Energy Procedia*, vol. 13, no. Complete, pp. 1853–1861, 2011.

[68] W. Frakes and R. Baeza-Yates, *Information Retrieval: Data Structures & Algorithms*. Prentice Hall, 1992.

[69] K. Ikeuchi, J. Wijekoon, S. Ishida, and H. Nishi, "GPU-based multi-stream analyzer on application layer for service-oriented router," in *Embedded Multicore Socs (MCSoC), 2013 IEEE 7th International Symposium on*, pp. 171–176, Sept 2013.

[70] S. Soroushnia, M. Daneshtalab, J. Plosila, and P. Liljeberg, *8th International Conference on Practical Applications of Computational Biology & Bioinformatics (PACBB 2014)*, ch. Heterogeneous Parallelization of Aho-Corasick Algorithm, pp. 153–160. Cham: Springer International Publishing, 2014.

[71] K. Kusudo, F. Ino, and K. Hagihara, "A bit-parallel algorithm for searching multiple patterns with various lengths," *Journal of Parallel and Distributed Computing*, vol. 76, no. 0, pp. 49 – 57, 2015. Special Issue on Architecture and Algorithms for Irregular Applications.

[72] S. Memeti and S. Pllana, "Accelerating DNA Sequence Analysis using Intel Xeon Phi," *ArXiv e-prints*, June 2015.

[73] E. Aragon, J. M. Jiménez, A. Maghazeh, J. Rasmusson, and U. D. Bordoloi, "Pattern matching in opencl: GPU vs CPU energy consumption on two mobile chipsets," in *Proceedings of the International Workshop on OpenCL 2013, 2014*, IWOCL '14, (New York, NY, USA), pp. 5:1–5:7, ACM, 2014.

[74] J. E. Stone, D. Gohara, and G. Shi, "Opencl: A parallel programming standard for heterogeneous computing systems," *IEEE Des. Test*, vol. 12, pp. 66–73, May 2010.

[75] S. Soroushnia, M. Daneshtalab, J. Plosila, T. Pahikkala, and P. Liljeberg, "High performance pattern matching on heterogeneous platform," *J. Integrative Bioinformatics*, vol. 11, no. 3, 2014.

[76] G. Vasiliadis and S. Ioannidis, "Gravity: A massively parallel antivirus engine," in *Recent Advances in Intrusion Detection* (S. Jha, R. Sommer, and C. Kreibich, eds.), vol. 6307 of *Lecture Notes in Computer Science*, pp. 79–96, Springer Berlin Heidelberg, 2010.

[77] R. Gerhards, "The syslog protocol," march 2009. Request for Comments RFC 5424 (Proposed Standard), IETF.

[78] M. Matsumoto and T. Nishimura, "Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator," *ACM Trans. Model. Comput. Simul.*, vol. 8, pp. 3–30, Jan. 1998.

[79] G. Vasiliadis, S. Antonatos, M. Polychronakis, E. Markatos, and S. Ioannidis, "Gnort: High performance network intrusion detection using graphics processors," in *Recent Advances in Intrusion Detection* (R. Lippmann, E. Kirda, and A. Trachtenberg, eds.), vol. 5230 of *Lecture Notes in Computer Science*, pp. 116–134, Springer Berlin Heidelberg, 2008.

[80] J. A. Anderson, C. D. Lorenz, and A. Travesset, "General purpose molecular dynamics simulations fully implemented on graphics processing units," *Journal of Computational Physics*, vol. 227, no. 10, pp. 5342 – 5359, 2008.

[81] P. Zikopoulos, C. Eaton, *et al.*, *Understanding big data: Analytics for enterprise class hadoop and streaming data*. McGraw-Hill Osborne Media, 2011.

[82] C.-W. Tsai, Y.-L. Yang, M.-C. Chiang, and C.-S. Yang, "Intelligent big data analysis: a review," *International Journal of Big Data Intelligence*, vol. 1, no. 4, pp. 181–191, 2014.

[83] J. Blom, T. Jakobi, D. Doppmeier, S. Jaenicke, J. Kalinowski, J. Stoye, and A. Goesmann, "Exact and complete short-read alignment to microbial genomes using graphics processing unit programming," *Bioinformatics*, vol. 27, no. 10, pp. 1351–1358, 2011.

[84] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "Starpu: a unified platform for task scheduling on heterogeneous multicore architectures," *Concurrency and Computation: Practice and Experience*, vol. 23, no. 2, pp. 187–198, 2011.

[85] T. J. Hansen, M. Mørup, and L. K. Hansen, "Non-parametric co-clustering of large scale sparse bipartite networks on the GPU," pp. 1–6, Sept 2011.

[86] T. Willhalm, N. Popovici, Y. Boshmaf, H. Plattner, A. Zeier, and J. Schaffner, "SIMD-scan: Ultra fast in-memory table scan using on-chip vector processing units," *Proc. VLDB Endow.*, vol. 2, pp. 385–394, Aug. 2009.

[87] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W.-m. W. Hwu, "Optimization principles and application performance evaluation of a multithreaded GPU using CUDA," in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, (New York, NY, USA), pp. 73–82, ACM, 2008.

[88] A. R. Brodtkorb, T. R. Hagen, and M. L. Soetra, "Graphics processing unit (GPU) programming strategies and trends in GPU computing," *Journal of Parallel and Distributed Computing*, vol. 73, no. 1, pp. 4 – 13, 2013. Metaheuristics on GPUs.

[89] T. B. Jablin, P. Prabhu, J. A. Jablin, N. P. Johnson, S. R. Beard, and D. I. August, "Automatic CPU-GPU communication management and optimization," *SIGPLAN Not.*, vol. 46, pp. 142–151, June 2011.

[90] T. B. Jablin, P. Prabhu, J. A. Jablin, N. P. Johnson, S. R. Beard, and D. I. August, "Automatic CPU-GPU communication management and optimization," in *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, (New York, NY, USA), pp. 142–151, ACM, 2011.

[91] P. Labs, "2013 annual report pandalabs." http://www.pandasecurity.com/mediacenter/src/uploads/2014/07/Annual-Report-PandaLabs-2013.pdf. Accessed: 2015-09-05.

[92] P. Labs, "2014 annual report pandalabs." http://www.pandasecurity.com/mediacenter/src/uploads/2015/02/Pandalabs2014-DEF2-en.pdf. Accessed: 2015-09-05.

[93] U. Fayyad, G. Piatetsky-Shapiro, and P. Smyth, "From data mining to knowledge discovery in databases," *AI magazine*, vol. 17, no. 3, p. 37, 1996.

[94] C. L. Lu, T. C. Wang, Y. C. Lin, and C. Y. Tang, “Robin: a tool for genome rearrangement of block-interchanges,” *Bioinformatics*, vol. 21, no. 11, pp. 2780–2782, 2005.

[95] H. Nguyen, *GPU gems 3*. Addison-Wesley Professional, 2007.

[96] C. Trapnell and M. C. Schatz, “Optimizing data intensive GPGPU computations for DNA sequence alignment,” *Parallel Comput.*, vol. 35, pp. 429–440, Aug. 2009.

[97] E. Ukkonen, “On-line construction of suffix trees,” *Algorithmica*, vol. 14, no. 3, pp. 249–260, 1995.

[98] M. Burtscher and K. Pingali, “An efficient CUDA implementation of the tree-based barnes hut n-body algorithm,” *GPU computing Gems Emerald edition*, p. 75, 2011.

[99] G. Vasiliadis, M. Polychronakis, and S. Ioannidis, “Midea: A multi-parallel intrusion detection architecture,” in *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS ’11, (New York, NY, USA), pp. 297–308, ACM, 2011.

[100] T.-H. Lee and N.-L. Huang, “An efficient and scalable pattern matching scheme for network security applications,” in *Computer Communications and Networks, 2008. ICCCN ’08. Proceedings of 17th International Conference on*, pp. 1–7, Aug 2008.

[101] C. Pungila and V. Negru, “A highly-efficient memory-compression approach for GPU-accelerated virus signature matching,” in *Information Security* (D. Gollmann and F. Freiling, eds.), vol. 7483 of *Lecture Notes in Computer Science*, pp. 354–369, Springer Berlin Heidelberg, 2012.

[102] S. Chang, *Data Structures and Algorithms*. Series on software engineering and knowledge engineering, World Scientific, 2003.

[103] R. Hyde, *The Art of Assembly Language, 2nd Edition:*. No Starch Press Series, No Starch Press, 2010.

[104] A. Aho, J. Hopcroft, and J. Ullman, *The design and analysis of computer algorithms*. Addison-Wesley series in computer science and information processing, Addison-Wesley Pub. Co., 1974.

[105] J. Bentley, *Programming Pearls*. ACM Press Series, Addison-Wesley, 2000.

[106] A. Aho, J. Hopcroft, and J. Ullman, *Data Structures and Algorithms*. Addison-Wesley series in computer science and information processing, Addison-Wesley, 1983.

[107] R. Vaarandi, "A breadth-first algorithm for mining frequent patterns from event logs," in *Intelligence in Communication Systems* (F. Aagesen, C. Anutariya, and V. Wuwongse, eds.), vol. 3283 of *Lecture Notes in Computer Science*, pp. 293–308, Springer Berlin Heidelberg, 2004.

[108] *SODA '96: Proceedings of the Seventh Annual ACM-SIAM Symposium on Discrete Algorithms*, (Philadelphia, PA, USA), Society for Industrial and Applied Mathematics, 1996.

[109] G. Vasiliadis, M. Polychronakis, S. Antonatos, E. Markatos, and S. Ioannidis, "Regular expression matching on graphics hardware for intrusion detection," in *Recent Advances in Intrusion Detection* (E. Kirda, S. Jha, and D. Balzarotti, eds.), vol. 5758 of *Lecture Notes in Computer Science*, pp. 265–283, Springer Berlin Heidelberg, 2009.

[110] J. Cheng, M. Grossman, and T. McKercher, *Professional CUDA C Programming*. EBL-Schweitzer, Wiley, 2014.

[111] G. S. Murthy, *Optimal loop unrolling for GPGPU programs*. PhD thesis, The Ohio State University, 2009.

[112] G. Murthy, M. Ravishankar, M. Baskaran, and P. Sadayappan, "Optimal loop unrolling for GPGPU programs," in *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pp. 1–11, April 2010.

[113] V. Sarkar, "Optimized unrolling of nested loops," *International Journal of Parallel Programming*, vol. 29, no. 5, pp. 545–581, 2001.

[114] M. Hugue, "Loop unrolling advantages and disadvantages." http://cmsc411.com/techniques/loop-unrolling-advantages-and-disadvantages. Accessed: 2015-09-29, University of Maryland.

[115] B. D. Noble, M. Satyanarayanan, G. T. Nguyen, and R. H. Katz, "Trace-based mobile network emulation," in *ACM SIGCOMM Computer Communication Review*, vol. 27, pp. 51–61, ACM, 1997.

[116] C. Williamson, "On filter effects in web caching hierarchies," *ACM Transactions on Internet Technology (TOIT)*, vol. 2, no. 1, pp. 47–77, 2002.

[117] J. Wang and Y. Hu, "Wolf - a novel reordering write buffer to boost the performance of log-structured file systems," in *Proceedings of the Conference on File and Storage Technologies*, FAST '02, (Berkeley, CA, USA), pp. 47–60, USENIX Association, 2002.

[118] V. M. Lo, J. Mache, and K. J. Windisch, "A comparative study of real workload traces and synthetic workload models for parallel job scheduling," in *Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*, IPPS/SPDP '98, (London, UK, UK), pp. 25–46, Springer-Verlag, 1998.

[119] M. Zink, K. Suh, Y. Gu, and J. Kurose, "Characteristics of youtube network traffic at a campus network – measurements, models, and implications," *Computer Networks*, vol. 53, no. 4, pp. 501 – 514, 2009. Content Distribution Infrastructures for Community Networks.

[120] M. Nicolae and S. Rajasekaran, "On string matching with mismatches," *Algorithms*, vol. 8, no. 2, pp. 248–270, 2015.

[121] M. A. Sahli, *Towards a Database System for Large-scale Analytics on Strings*. PhD thesis, King Abdullah University of Science and Technology, 2015.

[122] Amazon, "Amazon ec2 pricing." https://aws.amazon.com/ec2/pricing/. Accessed: 2015-10-19.

[123] G. Sakellari and G. Loukas, "A survey of mathematical models, simulation approaches and testbeds used for research in cloud computing," *Simulation Modelling Practice and Theory*, vol. 39, pp. 92 – 103, 2013. S.I.Energy efficiency in grids and clouds.

[124] O. Villa, A. Tumeo, and D. Sciuto, "Efficient pattern matching on GPUs for intrusion detection systems," tech. rep., Pacific Northwest National Laboratory (PNNL), Richland, WA (US), 2010.

[125] L. Wang, S.-H. Chen, J.-S. Su, and M.-J. Xu, "GPU-based regular expression match engine for deep packet inspection [j]," *Application Research of Computers*, vol. 11, p. 091, 2010.

[126] L. Wang, S. Chen, Y. Tang, and J. Su, "Gregex: GPU based high speed regular expression matching engine," in *Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS), 2011 Fifth International Conference on*, pp. 366–370, IEEE, 2011.

[127] P. Bakkum and K. Skadron, "Accelerating sql database operations on a GPU with CUDA," in *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, pp. 94–103, ACM, 2010.

[128] M. Alomari and K. Samsudin, "A framework for GPU-accelerated AES-XTS encryption in mobile devices," in *TENCON 2011 - 2011 IEEE Region 10 Conference*, pp. 144–148, Nov 2011.

[129] N.-F. Huang, H.-W. Hung, S.-H. Lai, Y.-M. Chu, and W.-Y. Tsai, "A GPU-based multiple-pattern matching algorithm for network intrusion detection systems," in *Advanced Information Networking and Applications - Workshops, 2008. AINAW 2008. 22nd International Conference on*, pp. 62–67, March 2008.

[130] M. Lee, J. hong Jeon, J. Kim, and J. Song, "Scalable and parallel implementation of a financial application on a GPU: With focus on out-of-core case," in *Computer and Information Technology (CIT), 2010 IEEE 10th International Conference on*, pp. 1323–1327, June 2010.

[131] D. Apostal, K. Foerster, A. Chatterjee, and T. Desell, "Password recovery using MPI and CUDA," in *High Performance Computing (HiPC), 2012 19th International Conference on*, pp. 1–9, Dec 2012.

[132] S. Lee, Y. Kim, J. Kim, and J. Kim, "Stealing webpages rendered on your browser by exploiting GPU vulnerabilities," in *35th IEEE Symposium on Security & Privacy (S&P)*, 2014.

[133] G. Vasiliadis, M. Polychronakis, and S. Ioannidis, "GPU-assisted malware," in *Malicious and Unwanted Software (MALWARE), 2010 5th International Conference on*, pp. 1–6, Oct 2010.

[134] G. Naumovich and N. Memon, "Preventing piracy, reverse engineering, and tampering," *Computer*, vol. 36, no. 7, pp. 64–71, 2003.

[135] C. Maurice, C. Neumann, O. Heen, and A. Francillon, "Confidentiality issues on a GPU in a virtualized environment," in *Financial Cryptography and Data Security* (N. Christin and R. Safavi-Naini, eds.), vol. 8437 of *Lecture Notes in Computer Science*, pp. 119–135, Springer Berlin Heidelberg, 2014.

[136] E. Ladakis, L. Koromilas, G. Vasiliadis, M. Polychronakis, and S. Ioannidis, "You can type, but you can't hide: A stealthy GPU-based keylogger," in *Proceedings of the 6th European Workshop on System Security (EuroSec)*, 2013.

[137] G. Vasiliadis, M. Polychronakis, and S. Ioannidis, “GPU-assisted malware,” *International Journal of Information Security*, vol. 14, no. 3, pp. 289–297, 2015.

[138] J. Danisevskis, M. Piekarska, and J.-P. Seifert, “Dark side of the shader: Mobile GPU-aided malware delivery,” in *Information Security and Cryptology – ICISC 2013* (H.-S. Lee and D.-G. Han, eds.), vol. 8565 of *Lecture Notes in Computer Science*, pp. 483–495, Springer International Publishing, 2014.

[139] S. Lee, Y. Kim, J. Kim, and J. Kim, “Stealing webpages rendered on your browser by exploiting GPU vulnerabilities,” in *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, SP '14, (Washington, DC, USA), pp. 19–33, IEEE Computer Society, 2014.

[140] E. J. Byrne, “Software reverse engineering: A case study,” *Software: Practice and Experience*, vol. 21, no. 12, pp. 1349–1364, 1991.

[141] S. Kato, “Implementing open-source cuda runtime,” in *Proceedings of the 54the Programming Symposium*, 2013.

[142] D. Balzarotti, R. D. Pietro, and A. Villani, “The impact of GPU-assisted malware on memory forensics: A case study,” *Digital Investigation*, vol. 14, Supplement 1, pp. S16 – S24, 2015. The Proceedings of the Fifteenth Annual {DFRWS} Conference.

[143] S. Breß, S. Kiltz, and M. Schäler, “Forensics on GPU coprocessing in databases–research challenges, first experiments, and countermeasures.,” in *BTW Workshops*, pp. 115–129, Citeseer, 2013.

[144] R. Di Pietro, F. Lombardi, and A. Villani, “Cuda leaks: information leakage in gpu architectures,” *arXiv preprint arXiv:1305.7383*, 2013.

[145] N. Firasta, M. Buxton, P. Jinbo, K. Nasri, and S. Kuo, “Intel avx: New frontiers in performance improvements and energy efficiency,” *Intel white paper*, 2008.

[146] K. Mayank, H. Dai, J. Wei, and H. Zhou, "Analyzing graphics processor unit (GPU) instruction set architectures," in *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium on*, pp. 155–156, March 2015.

[147] Nvidia, "Using inline PTX assembly in CUDA,"

[148] S. Cook, *CUDA Programming: A Developer's Guide to Parallel Computing with GPUs*. Applications of GPU computing series, Morgan Kaufmann, 2013.

[149] H. Huang, S. Zhu, P. Liu, and D. Wu, "A framework for evaluating mobile app repackaging detection algorithms," in *Trust and Trustworthy Computing* (M. Huth, N. Asokan, S. Čapkun, I. Flechais, and L. Coles-Kemp, eds.), vol. 7904 of *Lecture Notes in Computer Science*, pp. 169–186, Springer Berlin Heidelberg, 2013.

[150] S. Mittal, J. Vetter, and D. Li, "A survey of architectural approaches for managing embedded dram and non-volatile on-chip caches," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 26, pp. 1524–1537, June 2015.

[151] P. Van Aubel, D. J. Bernstein, and R. Niederhagen, "Investigating sram pufs in large CPUs and GPUs," in *Security, Privacy, and Applied Cryptography Engineering*, pp. 228–247, Springer, 2015.

[152] G. Shainer, A. Ayoub, P. Lui, T. Liu, M. Kagan, C. R. Trott, G. Scantlen, and P. S. Crozier, "The development of mellanox/nvidia GPUdirect over infiniband—a new model for GPU to GPU communications," *Computer Science - Research and Development*, vol. 26, no. 3, pp. 267–273, 2011.

[153] N. Maruyama, A. Nukada, S. Matsuoka, *et al.*, "Software-based ecc for GPUs," in *2009 Symposium on Application Accelerators in High Performance Computing (SAAHPC'09)*, vol. 107, 2009.

[154] I. S. Haque and V. S. Pande, “Hard data on soft errors: A large-scale assessment of real-world error rates in GPGPU,” in *Cluster, Cloud and Grid Computing (CCGrid), 2010 10th IEEE/ACM International Conference on*, pp. 691–696, IEEE, 2010.

[155] D. Fellner and S. Spencer, eds., *GH ’07: Proceedings of the 22Nd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware*, (Aire-la-Ville, Switzerland, Switzerland), Eurographics Association, 2007. 437077.

[156] F. Lombardi and R. Di Pietro, “Towards a GPU cloud: Benefits and security issues,” in *Continued Rise of the Cloud* (Z. Mahmood, ed.), Computer Communications and Networks, pp. 3–22, Springer London, 2014.

[157] Nvidia, “Nvidia SMI,” 2011. https://developer.nvidia.com/sites/NVML _cuda5/nvidia-smi.4.304.pdf.

[158] D. Brezinski and T. Killalea, “Guidelines for evidence collection and archiving,” 2002. https://www.ietf.org/rfc/rfc3227.txt.

[159] K. J. Jones, R. Bejtlich, and C. W. Rose, *Real Digital Forensics: Computer Security and Incident Response*. Addison-Wesley Professional, 2005.

[160] C. Malin, E. Casey, and J. Aquilina, *Malware Forensics Field Guide for Linux Systems: Digital Forensics Field Guides*. Digital forensics field guides, Elsevier Science, 2013.

[161] Department of Justice, “Digital forensics analysis methodology,” Aug. 2007. http://www.justice.gov/criminal/cybercrime/docs /forensics_chart.pdf.

[162] N. Son, Y. Lee, D. Kim, J. I. James, S. Lee, and K. Lee, “A study of user data integrity during acquisition of android devices,” *Digital Investigation*, vol. 10, Supplement, no. 0, pp. S3 – S11, 2013. The Proceedings of the Thirteenth Annual {DFRWS} Conference 13th Annual Digital Forensics Research Conference.

[163] A. Distefano, G. Me, and F. Pace, "Android anti-forensics through a local paradigm," *Digital Investigation*, vol. 7, Supplement, pp. S83 – S94, 2010. The Proceedings of the Tenth Annual {DFRWS} Conference.

[164] K. Jones, R. Bejtlich, and C. Rose, *Real Digital Forensics: Computer Security and Incident Response*. Addison-Wesley, 2006.

[165] Y. Zhang, B. Yang, M. Rogers, and R. Hansen, "Forensically sound retrieval and recovery of images from GPU memory," in *Digital Forensics and Cyber Crime* (J. James and F. Breitinger, eds.), vol. 157 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pp. 53–66, Springer International Publishing, 2015.

[166] E. Chikofsky and I. Cross, J.H., "Reverse engineering and design recovery: a taxonomy," *Software, IEEE*, vol. 7, pp. 13–17, Jan 1990.

[167] T. Systä and U. Tamperensis, "Static and dynamic reverse engineering techniques for Java software systems," 2000.

[168] G. Giunta, R. Montella, G. Agrillo, and G. Coviello, "gVirtuS: A GPGPU transparent virtualization component."

[169] E. Eilam, *Reversing: Secrets of Reverse Engineering*. Wiley, 2011.

[170] P. Stewin, J.-P. Seifert, and C. Mulliner, "Poster: Towards detecting DMA malware," in *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, (New York, NY, USA), pp. 857–860, ACM, 2011.

[171] S. Manavski, "CUDA compatible GPU as an efficient hardware accelerator for AES cryptography," in *Signal Processing and Communications, 2007. ICSPC 2007. IEEE International Conference on*, pp. 65–68, Nov 2007.

[172] D. Distler, "SANS institute - malware analysis: An introduction,"

[173] Q. Dong, T. Li, S. Zhang, X. Jiao, and J. Leng, "Ptx2kernel: Converting ptx code into compilable kernels," 2015.

[174] D. Low, "Protecting Java code via code obfuscation," *Crossroads*, vol. 4, pp. 21–23, Apr. 1998.

[175] K. Makan and S. Alexander-Bown, *Android Security Cookbook.* Packt Publishing, 2013.

[176] S. Skorobogatov, "Data remanence in flash memory devices," in *Cryptographic Hardware and Embedded Systems – CHES 2005* (J. Rao and B. Sunar, eds.), vol. 3659 of *Lecture Notes in Computer Science*, pp. 339–353, Springer Berlin Heidelberg, 2005.

[177] N. Tuck, T. Sherwood, B. Calder, and G. Varghese, "Deterministic memory-efficient string matching algorithms for intrusion detection," in *INFOCOM 2004. Twenty-third AnnualJoint Conference of the IEEE Computer and Communications Societies*, vol. 4, pp. 2628–2639 vol.4, March 2004.

[178] S. Kim, S. Huh, X. Zhang, Y. Hu, A. Wated, E. Witchel, and M. Silberstein, "GPUnet: Networking abstractions for GPU programs," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, (Broomfield, CO), pp. 201–216, USENIX Association, Oct. 2014.

[179] R. M. Karp and M. Rabin, "Efficient randomized pattern-matching algorithms," *IBM Journal of Research and Development*, vol. 31, pp. 249–260, March 1987.

[180] B. Dömölki, "A universal compiler system based on production rules," *BIT Numerical Mathematics*, vol. 8, no. 4, pp. 262–275.

[181] F. Vázquez, G. Ortega, J.-J. Fernández, and E. M. Garzón, "Improving the performance of the sparse matrix vector product with GPUs," in *Computer and Infor-*

mation Technology (CIT), 2010 IEEE 10th International Conference on, pp. 1146–1151, IEEE, 2010.

[182] P. Runeson, "A survey of unit testing practices," *Software, IEEE*, vol. 23, no. 4, pp. 22–29, 2006.

[183] J. Zhao, "Data-flow-based unit testing of aspect-oriented programs," in *Computer Software and Applications Conference, 2003. COMPSAC 2003. Proceedings. 27th Annual International*, pp. 188–197, IEEE, 2003.

Appendix A

Unit Testing

A.1 Introduction

This Appendix provides information on the Unit Testing process used to evaluate the accuracy of the HEPFAC algorithm, reduction and compression stages.

Unit testing is a software development practice allowing to independently scrutinise the operations of an algorithm, in order to evaluate its correctness [182] [183]. Information regarding the automatic unit testing designed for the HEPFAC algorithm are provided in the next section.

A.2 Unit Testing

In order to test the storage model, the trie reduction and the matching process described in Chapter 4, a number of tests have been created, and are organised as follows:

I: A defined number of strings are chosen from a text string.

II: A breadth first row major ordered trie is constructed. At this stage visual and textual information is provided for visual inspection.

III: The trie is truncated and reduced. This stage involves a visual and textual representation of the trie for visual inspection.

IV: The trie traversed in order to march all patterns present in the trie.

V: The patterns are searched against a text string on the GPU verifying that the patterns are present in the text at corresponding indices.

```
adegijaf
afbccdfd
cbgdiiib
cfbedhfe
dffciahf
eiffbbie
ghfhjijj
hgdgbcji
hgeibfjc
ijddgaig
```

Figure A.1
10 Patterns Chosen Randomly from a Text String

In order to evaluate the algorithm 10 patterns are chosen from a text string. These patterns are chosen by using the MT uniform pseudo-random number generator. Figure A.1 shows the 10 patterns chosen during the first stage of the evaluation.

The trie is then constructed and a textual representation of the trie is provided to the user.

```
treeArray[0].offset 1  -> prev_node 0 -> nodeID 0
treeArray[1].offset 8  -> prev_node 0 -> nodeID 1
treeArray[2].offset 10  -> prev_node 0 -> nodeID 2
treeArray[3].offset 12  -> prev_node 0 -> nodeID 3
treeArray[4].offset 13  -> prev_node 0 -> nodeID 4
treeArray[5].offset 14  -> prev_node 0 -> nodeID 5
treeArray[6].offset 15  -> prev_node 0 -> nodeID 6
treeArray[7].offset 16  -> prev_node 0 -> nodeID 7
treeArray[8].offset 17  -> prev_node 1 -> nodeID 8
treeArray[9].offset 18  -> prev_node 1 -> nodeID 9
treeArray[10].offset 19  -> prev_node 2 -> nodeID 10
treeArray[11].offset 20  -> prev_node 2 -> nodeID 11
treeArray[12].offset 21  -> prev_node 3 -> nodeID 12
treeArray[13].offset 22  -> prev_node 4 -> nodeID 13
treeArray[14].offset 23  -> prev_node 5 -> nodeID 14
treeArray[15].offset 24  -> prev_node 6 -> nodeID 15
treeArray[16].offset 26  -> prev_node 7 -> nodeID 16
treeArray[17].offset 27  -> prev_node 8 -> nodeID 17
treeArray[18].offset 28  -> prev_node 9 -> nodeID 18
treeArray[19].offset 29  -> prev_node 10 -> nodeID 19
treeArray[20].offset 30  -> prev_node 11 -> nodeID 20
treeArray[21].offset 31  -> prev_node 12 -> nodeID 21
treeArray[22].offset 32  -> prev_node 13 -> nodeID 22
treeArray[23].offset 33  -> prev_node 14 -> nodeID 23
treeArray[24].offset 34  -> prev_node 15 -> nodeID 24
treeArray[25].offset 35  -> prev_node 15 -> nodeID 25
```

Figure A.2
Breadth-First Row Major Order Constructed Trie

```
treeArray[26].offset 36  -> prev_node 16 -> nodeID 26
treeArray[27].offset 37  -> prev_node 17 -> nodeID 27
treeArray[28].offset 38  -> prev_node 18 -> nodeID 28
treeArray[29].offset 39  -> prev_node 19 -> nodeID 29
treeArray[30].offset 40  -> prev_node 20 -> nodeID 30
treeArray[31].offset 41  -> prev_node 21 -> nodeID 31
treeArray[32].offset 42  -> prev_node 22 -> nodeID 32
treeArray[33].offset 43  -> prev_node 23 -> nodeID 33
treeArray[34].offset 44  -> prev_node 24 -> nodeID 34
treeArray[35].offset 45  -> prev_node 25 -> nodeID 35
treeArray[36].offset 46  -> prev_node 26 -> nodeID 36
treeArray[37].offset 47  -> prev_node 27 -> nodeID 37
treeArray[38].offset 48  -> prev_node 28 -> nodeID 38
treeArray[39].offset 49  -> prev_node 29 -> nodeID 39
treeArray[40].offset 50  -> prev_node 30 -> nodeID 40
treeArray[41].offset 51  -> prev_node 31 -> nodeID 41
treeArray[42].offset 52  -> prev_node 32 -> nodeID 42
treeArray[43].offset 53  -> prev_node 33 -> nodeID 43
treeArray[44].offset 54  -> prev_node 34 -> nodeID 44
treeArray[45].offset 55  -> prev_node 35 -> nodeID 45
treeArray[46].offset 56  -> prev_node 36 -> nodeID 46
treeArray[47].offset 57  -> prev_node 37 -> nodeID 47
treeArray[48].offset 58  -> prev_node 38 -> nodeID 48
treeArray[49].offset 59  -> prev_node 39 -> nodeID 49
treeArray[50].offset 60  -> prev_node 40 -> nodeID 50
treeArray[51].offset 61  -> prev_node 41 -> nodeID 51
treeArray[52].offset 62  -> prev_node 42 -> nodeID 52
```

Figure A.3
Breadth-First Row Major Order Constructed Trie

```
treeArray[53].offset 63  -> prev_node 43 -> nodeID 53
treeArray[54].offset 64  -> prev_node 44 -> nodeID 54
treeArray[55].offset 65  -> prev_node 45 -> nodeID 55
treeArray[56].offset 66  -> prev_node 46 -> nodeID 56
treeArray[57].offset 67  -> prev_node 47 -> nodeID 57
treeArray[58].offset 68  -> prev_node 48 -> nodeID 58
treeArray[59].offset 69  -> prev_node 49 -> nodeID 59
treeArray[60].offset 70  -> prev_node 50 -> nodeID 60
treeArray[61].offset 71  -> prev_node 51 -> nodeID 61
treeArray[62].offset 72  -> prev_node 52 -> nodeID 62
treeArray[63].offset 73  -> prev_node 53 -> nodeID 63
treeArray[64].offset 74  -> prev_node 54 -> nodeID 64
treeArray[65].offset 75  -> prev_node 55 -> nodeID 65
treeArray[66].offset 76  -> prev_node 56 -> nodeID 66
treeArray[67].offset 0  -> prev_node 57 -> nodeID 67
treeArray[68].offset 0  -> prev_node 58 -> nodeID 68
treeArray[69].offset 0  -> prev_node 59 -> nodeID 69
treeArray[70].offset 0  -> prev_node 60 -> nodeID 70
treeArray[71].offset 0  -> prev_node 61 -> nodeID 71
treeArray[72].offset 0  -> prev_node 62 -> nodeID 72
treeArray[73].offset 0  -> prev_node 63 -> nodeID 73
treeArray[74].offset 0  -> prev_node 64 -> nodeID 74
treeArray[75].offset 0  -> prev_node 65 -> nodeID 75
treeArray[76].offset 0  -> prev_node 66 -> nodeID 76
treeArray[77].offset 0  -> prev_node 0 -> nodeID 0
treeArray[78].offset 0  -> prev_node 0 -> nodeID 0
treeArray[79].offset 0  -> prev_node 0 -> nodeID 0
treeArray[80].offset 0  -> prev_node 0 -> nodeID 0
```

Figure A.4
Breadth-First Row Major Order Constructed Trie

Figure A.2, A.3, A.4 provide the user with the possibility of a textual inspection. It also provides information on the *NodeID*, in order to identify the parent of every single node.

This part of the unit testing is also able to provide the nodes in a GraphViz friendly output, allowing a graphical representation of the trie, as shown in Figure 4.8 in Chapter 4. The next stage involves the identification of all terminal nodes.

```
Last Nodes Array :
67 68 69 70 71 72 73 74 75 76
57 58 59 60 61 62 63 64 65 66
47 48 49 50 51 52 53 54 55 56
```

Figure A.5
Identification of the terminal nodes, their parents, and grand-parents

Figure A.5 identifies the last nodes of the array, their parents and grand parents, this information is subsequently used by the branch reduction algorithm in order to reduce the branches. This information can be compared against the GraphViz output provided in the previous test described in Figure A.4. The trie is subsequently reduced.

```
treeArray[0].offset 1  -> prev_node 0 -> nodeID 0
treeArray[1].offset 8  -> prev_node 0 -> nodeID 1
treeArray[2].offset 10  -> prev_node 0 -> nodeID 2
treeArray[3].offset 12  -> prev_node 0 -> nodeID 3
treeArray[4].offset 13  -> prev_node 0 -> nodeID 4
treeArray[5].offset 14  -> prev_node 0 -> nodeID 5
treeArray[6].offset 15  -> prev_node 0 -> nodeID 6
treeArray[7].offset 16  -> prev_node 0 -> nodeID 7
treeArray[8].offset 17  -> prev_node 1 -> nodeID 8
treeArray[9].offset 18  -> prev_node 1 -> nodeID 9
treeArray[10].offset 19  -> prev_node 2 -> nodeID 10
treeArray[11].offset 20  -> prev_node 2 -> nodeID 11
treeArray[12].offset 21  -> prev_node 3 -> nodeID 12
treeArray[13].offset 22  -> prev_node 4 -> nodeID 13
treeArray[14].offset 23  -> prev_node 5 -> nodeID 14
treeArray[15].offset 24  -> prev_node 6 -> nodeID 15
treeArray[16].offset 26  -> prev_node 7 -> nodeID 16
treeArray[17].offset 27  -> prev_node 8 -> nodeID 17
treeArray[18].offset 28  -> prev_node 9 -> nodeID 18
treeArray[19].offset 29  -> prev_node 10 -> nodeID 19
treeArray[20].offset 30  -> prev_node 11 -> nodeID 20
treeArray[21].offset 31  -> prev_node 12 -> nodeID 21
treeArray[22].offset 32  -> prev_node 13 -> nodeID 22
treeArray[23].offset 33  -> prev_node 14 -> nodeID 23
treeArray[24].offset 34  -> prev_node 15 -> nodeID 24
treeArray[25].offset 35  -> prev_node 15 -> nodeID 25
treeArray[26].offset 36  -> prev_node 16 -> nodeID 26
```

Figure A.6
Reduced Breadth-First Row Major Ordered Array

```
treeArray[27].offset 37  -> prev_node 17 -> nodeID 27
treeArray[28].offset 38  -> prev_node 18 -> nodeID 28
treeArray[29].offset 39  -> prev_node 19 -> nodeID 29
treeArray[30].offset 40  -> prev_node 20 -> nodeID 30
treeArray[31].offset 41  -> prev_node 21 -> nodeID 31
treeArray[32].offset 42  -> prev_node 22 -> nodeID 32
treeArray[33].offset 43  -> prev_node 23 -> nodeID 33
treeArray[34].offset 44  -> prev_node 24 -> nodeID 34
treeArray[35].offset 45  -> prev_node 25 -> nodeID 35
treeArray[36].offset 46  -> prev_node 26 -> nodeID 36
treeArray[37].offset 47  -> prev_node 27 -> nodeID 37
treeArray[38].offset 48  -> prev_node 28 -> nodeID 38
treeArray[39].offset 49  -> prev_node 29 -> nodeID 39
treeArray[40].offset 50  -> prev_node 30 -> nodeID 40
treeArray[41].offset 51  -> prev_node 31 -> nodeID 41
treeArray[42].offset 52  -> prev_node 32 -> nodeID 42
treeArray[43].offset 53  -> prev_node 33 -> nodeID 43
treeArray[44].offset 54  -> prev_node 34 -> nodeID 44
treeArray[45].offset 55  -> prev_node 35 -> nodeID 45
treeArray[46].offset 56  -> prev_node 36 -> nodeID 46
treeArray[47].offset 57  -> prev_node 37 -> nodeID 47
treeArray[48].offset 58  -> prev_node 38 -> nodeID 48
treeArray[49].offset 59  -> prev_node 39 -> nodeID 49
treeArray[50].offset 60  -> prev_node 40 -> nodeID 50
treeArray[51].offset 61  -> prev_node 41 -> nodeID 51
treeArray[52].offset 62  -> prev_node 42 -> nodeID 52
treeArray[53].offset 63  -> prev_node 43 -> nodeID 53
```

Figure A.7
Reduced Breadth-First Row Major Ordered Array

```
treeArray[54].offset 64  -> prev_node 44 -> nodeID 54
treeArray[55].offset 65  -> prev_node 45 -> nodeID 55
treeArray[56].offset 66  -> prev_node 46 -> nodeID 56
treeArray[57].offset 67  -> prev_node 47 -> nodeID 57
treeArray[58].offset 67  -> prev_node 48 -> nodeID 58
treeArray[59].offset 67  -> prev_node 49 -> nodeID 59
treeArray[60].offset 67  -> prev_node 50 -> nodeID 60
treeArray[61].offset 67  -> prev_node 51 -> nodeID 61
treeArray[62].offset 67  -> prev_node 52 -> nodeID 62
treeArray[63].offset 67  -> prev_node 53 -> nodeID 63
treeArray[64].offset 67  -> prev_node 54 -> nodeID 64
treeArray[65].offset 67  -> prev_node 55 -> nodeID 65
treeArray[66].offset 67  -> prev_node 56 -> nodeID 66
treeArray[67].offset 0  -> prev_node 57 -> nodeID 67
treeArray[68].offset 0  -> prev_node 58 -> nodeID 68
treeArray[69].offset 0  -> prev_node 59 -> nodeID 69
treeArray[70].offset 0  -> prev_node 60 -> nodeID 70
treeArray[71].offset 0  -> prev_node 61 -> nodeID 71
treeArray[72].offset 0  -> prev_node 62 -> nodeID 72
treeArray[73].offset 0  -> prev_node 63 -> nodeID 73
treeArray[74].offset 0  -> prev_node 64 -> nodeID 74
treeArray[75].offset 0  -> prev_node 65 -> nodeID 75
treeArray[76].offset 0  -> prev_node 66 -> nodeID 76
treeArray[77].offset 0  -> prev_node 0 -> nodeID 0
treeArray[78].offset 0  -> prev_node 0 -> nodeID 0
treeArray[79].offset 0  -> prev_node 0 -> nodeID 0
treeArray[80].offset 0  -> prev_node 0 -> nodeID 0
```

Figure A.8
Reduced Breadth-First Row Major Ordered Array

Figure A.6, A.7, A.8 provide a visual inspection for the breadth-first row major ordered array. This step of the reduction is also provided with a GraphViz version in order to provide the user with a visual inspection of the trie.

```
47 48 49 50 51 52 53 54 55 56
 To ->
47 48 49 50 51 52 53 54 55 56
```

Figure A.9
Merged nodes

A list of merged nodes is also provided to the user, along with a list of terminal nodes, this allows the user to compare the output of this test against the visual representation made by the GraphViz output as shown in Figure A.9. Note that in this example the patterns provided have no common suffixes and therefore none of the branches are merged together. Textual and visual outputs are provided for inspection.

```
treeReduced[0].offset 1 -> Supposed Node ID 1
treeReduced[1].offset 8 -> Supposed Node ID 2
treeReduced[2].offset 10 -> Supposed Node ID 3
treeReduced[3].offset 12 -> Supposed Node ID 4
treeReduced[4].offset 13 -> Supposed Node ID 5
treeReduced[5].offset 14 -> Supposed Node ID 6
treeReduced[6].offset 15 -> Supposed Node ID 7
treeReduced[7].offset 16 -> Supposed Node ID 8
treeReduced[8].offset 17 -> Supposed Node ID 9
treeReduced[9].offset 18 -> Supposed Node ID 10
treeReduced[10].offset 19 -> Supposed Node ID 11
treeReduced[11].offset 20 -> Supposed Node ID 12
treeReduced[12].offset 21 -> Supposed Node ID 13
treeReduced[13].offset 22 -> Supposed Node ID 14
treeReduced[14].offset 23 -> Supposed Node ID 15
treeReduced[15].offset 24 -> Supposed Node ID 16
treeReduced[16].offset 26 -> Supposed Node ID 17
treeReduced[17].offset 27 -> Supposed Node ID 18
treeReduced[18].offset 28 -> Supposed Node ID 19
treeReduced[19].offset 29 -> Supposed Node ID 20
treeReduced[20].offset 30 -> Supposed Node ID 21
treeReduced[21].offset 31 -> Supposed Node ID 22
treeReduced[22].offset 32 -> Supposed Node ID 23
treeReduced[23].offset 33 -> Supposed Node ID 24
treeReduced[24].offset 34 -> Supposed Node ID 25
treeReduced[25].offset 35 -> Supposed Node ID 26
treeReduced[26].offset 36 -> Supposed Node ID 27
```

Figure A.10
Collated Terminal Nodes.

```
treeReduced[27].offset 37 -> Supposed Node ID 28
treeReduced[28].offset 38 -> Supposed Node ID 29
treeReduced[29].offset 39 -> Supposed Node ID 30
treeReduced[30].offset 40 -> Supposed Node ID 31
treeReduced[31].offset 41 -> Supposed Node ID 32
treeReduced[32].offset 42 -> Supposed Node ID 33
treeReduced[33].offset 43 -> Supposed Node ID 34
treeReduced[34].offset 44 -> Supposed Node ID 35
treeReduced[35].offset 45 -> Supposed Node ID 36
treeReduced[36].offset 46 -> Supposed Node ID 37
treeReduced[37].offset 47 -> Supposed Node ID 38
treeReduced[38].offset 48 -> Supposed Node ID 39
treeReduced[39].offset 49 -> Supposed Node ID 40
treeReduced[40].offset 50 -> Supposed Node ID 41
treeReduced[41].offset 51 -> Supposed Node ID 42
treeReduced[42].offset 52 -> Supposed Node ID 43
treeReduced[43].offset 53 -> Supposed Node ID 44
treeReduced[44].offset 54 -> Supposed Node ID 45
treeReduced[45].offset 55 -> Supposed Node ID 46
treeReduced[46].offset 56 -> Supposed Node ID 47
treeReduced[47].offset 57 -> Supposed Node ID 48
treeReduced[48].offset 58 -> Supposed Node ID 49
treeReduced[49].offset 59 -> Supposed Node ID 50
treeReduced[50].offset 60 -> Supposed Node ID 51
treeReduced[51].offset 61 -> Supposed Node ID 52
treeReduced[52].offset 62 -> Supposed Node ID 53
treeReduced[53].offset 63 -> Supposed Node ID 54
```

Figure A.11
Collated Terminal Nodes.

```
treeReduced[54].offset 64 -> Supposed Node ID 55
treeReduced[55].offset 65 -> Supposed Node ID 56
treeReduced[56].offset 66 -> Supposed Node ID 57
treeReduced[57].offset 67 -> Supposed Node ID 58
treeReduced[58].offset 67 -> Supposed Node ID 59
treeReduced[59].offset 67 -> Supposed Node ID 60
treeReduced[60].offset 67 -> Supposed Node ID 61
treeReduced[61].offset 67 -> Supposed Node ID 62
treeReduced[62].offset 67 -> Supposed Node ID 63
treeReduced[63].offset 67 -> Supposed Node ID 64
treeReduced[64].offset 67 -> Supposed Node ID 65
treeReduced[65].offset 67 -> Supposed Node ID 66
treeReduced[66].offset 67 -> Supposed Node ID 67
treeReduced[67].offset -1 -> Supposed Node ID 68
```

Figure A.12
Collated Terminal Nodes.

Figure A.10, A.11, A.12 provide the user with a textual and visual output in order to inspect the last node reduction process. In this particular example, the number of nodes is reduced by the number of patterns +1. This is due to the lack of branches being merged. This process is described by Equation 4.2 in Section 4.3.2.

Subsequent to the trie reduction, the trie is traversed in order to match all patterns and provide the user with a description of the trie.

```
a 0 -> 1
d 1 -> 8
e 8 -> 17
g 17 -> 27
i 27 -> 37
j 37 -> 47
a 47 -> 57
f 57 -> 67
Match at 0
a 0 -> 1
f 1 -> 8
b 9 -> 18
c 18 -> 28
c 28 -> 38
d 38 -> 48
f 48 -> 58
d 58 -> 67
Match at 1
```

Figure A.13
Matching Pattern 1 and 2

```
c 0 -> 1
b 2 -> 10
g 10 -> 19
d 19 -> 29
i 29 -> 39
i 39 -> 49
i 49 -> 59
b 59 -> 67
Match at 2
c 0 -> 1
f 2 -> 10
b 11 -> 20
e 20 -> 30
d 30 -> 40
h 40 -> 50
f 50 -> 60
e 60 -> 67
Match at 3
d 0 -> 1
f 3 -> 12
f 12 -> 21
c 21 -> 31
i 31 -> 41
a 41 -> 51
h 51 -> 61
f 61 -> 67
Match at 4
```

Figure A.14
Matching Pattern 3, 4 and 5

```
e 0 -> 1
i 4 -> 13
f 13 -> 22
f 22 -> 32
b 32 -> 42
b 42 -> 52
i 52 -> 62
e 62 -> 67
Match at 5
g 0 -> 1
h 5 -> 14
f 14 -> 23
h 23 -> 33
j 33 -> 43
i 43 -> 53
j 53 -> 63
j 63 -> 67
Match at 6
h 0 -> 1
g 6 -> 15
d 15 -> 24
g 24 -> 34
b 34 -> 44
c 44 -> 54
j 54 -> 64
i 64 -> 67
Match at 7
```

Figure A.15
Matching Pattern 6, 7 and 8

```
h 0 -> 1
g 6 -> 15
e 15 -> 24
i 25 -> 35
b 35 -> 45
f 45 -> 55
j 55 -> 65
c 65 -> 67
Match at 8
i 0 -> 1
j 7 -> 16
d 16 -> 26
d 26 -> 36
g 36 -> 46
a 46 -> 56
i 56 -> 66
g 66 -> 67
Match at 9
```

Figure A.16
Matching Pattern 9 and 10

```
verification done
Match at 0
Match at 1
Match at 2
Match at 3
Match at 4
Match at 5
Match at 6
Match at 7
Match at 8
Match at 9
verification reduced done
```

Figure A.17
Pattern Matching Verification Output

During the trie traversal, all patterns are matched; this is shown in Figure A.13, A.14, A.15 and Figure A.16. This is also acknowledged by a second matching process, matching all patterns against the trie as shown in Figure A.17. This process demonstrates the validity of the trie reduction, compression and the terminal node merging. A final test can be triggered to match the patterns against one or more text strings on the GPU.

```
Match at 8
Match at 1513139
Match at 1533723
Match at 1686587
Match at 2671179
Match at 2671752
Match at 3219378
Match at 3513334
Match at 3981157
Match at 4563424
Positive Matches in Text String 10
```

Figure A.18
Positive Matches Found on GPU in a Text String

Figure A.18 shows the results obtained by matching the trie against a defined text string, as well as the final number of matches. This allows to test the trie reduction algorithm against a number of different synthetic text strings and ensure its accuracy.