

Library Declaration Form



University of Otago Library

Author's full name and year of birth: Peng Wu,
(for cataloguing purposes) 28 October 1985

Title of thesis: Clgrep: A Parallel String Matching Tool

Degree: Master of Science

Department: Department of Information Science

Permanent Address: 111 North Road, Dunedin, NZ

I agree that this thesis may be consulted for research and study purposes and that reasonable quotation may be made from it, provided that proper acknowledgement of its use is made.

I consent to this thesis being copied in part or in whole for

- i) a library
- ii) an individual

at the discretion of the University of Otago.

Signature:

Date:

Clgrep: A Parallel String Matching Tool

Peng Wu

a thesis submitted for the degree of

Master of Science

at the University of Otago, Dunedin,

New Zealand.

March 18, 2013

Abstract

In this study, we widely investigate the problem of string matching in the context of Heterogeneous Parallel Computing. A overview of string matching is made, in which the different forms of string matching problem are distinguished, and the classifications of string matching algorithm are discussed.

As an alternative to grep for computational intensive string matching and in addition to support the research of the study, a parallel exact string matching utility ‘Clgrep’ is developed.

By experimental studies, we investigate the use of heuristics-based algorithms, specifically QS and Horspool algorithms, in the context of Heterogeneous Parallel Computing. The results suggest that the performance of Heterogeneous Parallel Computing matching, either on multi-core CPU or GPU, is highly related to the computational intensity of certain cases. When computational power is intensively required, the SIMD Parallel Computing model of Clgrep can be several times more efficient than corresponding sequential matching program.

Acknowledgements

This study has been a difficult task as the problem is novel and involves many programming challenges. Everyday, I am fighting with the errors in the program and the issues raised in the experiment. I do not remember how many hours were spent in the complex experiment. However, I can not forget the people helped me during the study.

I would like to thank my parents, without them my study would merely be a dream.

I would like to thank my supervisor, Dr Mariusz Nowostawski, for his enthusiasm, guidance, patience and support throughout the duration of this thesis.

I would like to thank my darling Rita Li for her encouragement and support.

I would also like to thank Stephen Hall-Jones, Gail Mercer, the past and present members of TSG, and others who work and contribute to the Information Science and Computing Science departments. I appreciate your suggestion, help, and support.

Finally, this study is based the findings of many previous studies. I would like to thank the researchers in the string matching area.

Contents

1	Introduction	1
1.1	Motivations	3
1.2	Research Objective	4
1.3	Contributions	4
1.4	Thesis Structure	5
2	On-line String Matching	6
2.1	The Definition of Exact String Matching Problems	7
2.2	An Overview of Exact String Matching Algorithms	8
2.3	The Algorithms Closely Related to The Study	11
2.3.1	Naive Matching Algorithm: A naive approach	11
2.3.2	Aho-Corasick Algorithm: A finite automaton multi-pattern matching algorithm	12
2.3.3	Karp-Rabin Algorithm: A hashing-based approach	12
2.3.4	Wu-Manber Algorithm: A hybrid approach	12
2.3.5	Boyer-Moore Algorithm and Its Variants: The root of heuristics approach	13
2.3.6	Summary	15
3	Open Computing Language (Welcome to The World of Heterogeneous Parallel Computing)	16
3.1	Parallel Computing	17
3.1.1	SISD	18
3.1.2	SIMD	18

3.1.3	MISD	19
3.1.4	MIMD	19
3.2	GPGPU Computing	20
3.2.1	CUDA	22
3.3	The Architecture of OpenCL	23
3.3.1	The Platform Model	23
3.3.2	The Execution Model	24
3.3.3	The Memory Model	24
3.3.4	The Programming Model	26
3.4	Summary	26
4	Related Works	27
4.1	Traditional Grep-like Tools	28
4.1.1	GNU grep	28
4.1.2	Agrep: Approximate grep	29
4.1.3	NR-grep : Non-deterministic Reverse grep	29
4.2	GPGPU Networking Intrusion Detection Systems (NIDS)	30
4.2.1	PixelSnort: The First Attempt	30
4.2.2	Evaluating GPUs for Network Packet Signature Matching: An Analysis of GPU-based String Matching	31
4.2.3	Gnort: A High Performance GPGPU NIDS based on Snort	32
4.3	String Matching Algorithm Evaluation on GPU	33
4.3.1	Evaluating The Performance of Approximate String Matching On GPU	33
4.3.2	String Matching on a multi-core GPU with CUDA	34
4.4	The Research Gaps	35
5	Clgrep: A New Approach	36
5.1	The Algorithms of Clgrep	37
5.1.1	Horspool Algorithm	37
5.1.2	Quick Search Algorithm	37

5.1.3	The Multi-pattern Matching Strategies of Clgrep	38
5.2	The Processing Mode and Matching Procedure of Clgrep	40
5.2.1	Processing Modes	40
5.2.2	Matching Procedure	41
5.2.3	The CG Mode of Clgrep	44
5.3	Optimizations	45
5.3.1	Optimized Data Partitioning	45
5.3.2	GPU Memory Optimizations	47
5.3.3	Other Optimizations	48
5.4	The Implementation of Clgrep	50
5.5	Summary	52
6	Experiments	53
6.1	The Aims of Experiment	54
6.2	Experiment Methodology	54
6.2.1	The Matching Cases	55
6.2.2	The Sequential Implementation	56
6.2.3	Experiment Procedure	56
6.2.4	Time Measurement	58
6.2.5	Experiment Setup	60
6.3	Results and Discussion	62
6.3.1	Single Pattern Matching	62
6.3.2	Multi-pattern Matching	67
6.3.3	The Scalability of Clgrep	75
6.4	Summary	79
7	Future Work	80
7.0.1	Other Forms of String Matching Problem	80
7.0.2	The Development of grep-like String Matching Utilities	80
7.0.3	The Evaluation of Appropriate Algorithms	80

7.0.4	Clgrep vs. grep	81
8	Conclusions	82
8.1	The Limitation of The Study	85
A	The Source Codes of Clgrep and The Bash Scripts Used in The Experiment	86
B	The Experiment Results	87
B.0.1	Single pattern matching	87
B.0.2	Multi-pattern matching	88
	References	89

List of Tables

5.1	The processing modes of Clgrep.	40
5.2	The compilation extensions of OpenCL that may potentially improve execution efficiency.	49
5.3	The options of Clgrep.	51
6.1	The matching cases of the experiment.	55
6.2	The non-matching costs of every single search.	58
6.3	The experiment settings and software versions.	60
6.4	The hardware settings of experiment.	61
6.5	English single pattern matching: The sequential implementation vs. BMH CG mode.	63
6.6	English single pattern matching: BMH vs. QS.	64
6.7	The results of matching 20 patterns concurrently in 100 MB English text.	70

List of Figures

2.1	Prefix Suffix and Factor.	8
2.2	The search window is shifted from left to right, and always searches the longest same Prefix.	9
2.3	Searching the longest matched Suffix from right to left.	9
2.4	The search process of Naive algorithm.	11
2.5	BM algorithm is researched backward, “n” is firstly compared with “s”.	13
2.6	After applied Bad-character heuristic, the search window is shifted.	14
2.7	A mismatch occurs after two exact matches.	14
2.8	Good-suffixes heuristic.	14
3.1	SISD	18
3.2	SIMD	18
3.3	MISD	19
3.4	MIMD	19
3.5	The architectures of CPU and GPU.	21
3.6	The platform model of OpenCL	23
3.7	ND-Range	24
3.8	The memory hierarchy of OpenCL	25
4.1	GNU grep: print lines matching a pattern.	28
5.1	The search window of QS algorithm is shifted from left to right. But instead of shifting the search window upon the last character, the shift of QS algorithm is determined according to the character right next to the last character of the search window. (In this case it shift upon “t” not “s”.)	38

5.2	The preprocessing stage of Clgrep.	41
5.3	The matching stage of Clgrep.	42
5.4	Data Partitioning.	42
5.5	The post-processing stage of Clgrep.	43
5.6	Matching text is divided when a single pattern is matched in CG mode.	44
5.7	Patterns we are searching for are grouped then assigned to different processors.	44
5.8	A demonstration of the data partitioning mechanism of OpenCL. . . .	45
5.9	Experiment results of the appropriate work-group size.	46
5.10	The caching subsystem of some GPUs.	47
5.11	A demonstration of collated accessing.	48
5.12	The architecture of Clgrep is based on functions.	50
6.1	Experiment procedure.	57
6.2	The raw data collected from a single experiment.	57
6.3	The entire process is carefully measured in the experiment.	58
6.4	An overview of the hardware experiment settings.	61
6.5	The results of Horspool algorithm.	62
6.6	The results of QS algorithm.	63
6.7	The result of DNA single pattern matching.	65
6.8	A more precise comparison between the multi-core mode of Clgrep and the sequential implementation.	66
6.9	The results of matching 10 patterns concurrently in English text. . . .	67
6.10	Matching 10 patterns in 100 MB English text.	68
6.11	The performance trend from 1 MB to 10 MB.	68
6.12	The internal comparison of Clgrep when matching 10 patterns concur- rently in English text.	69
6.13	The results of English multi-pattern matching (20 patterns).	70
6.14	The differences between each mode of Clgrep when matching 20 patterns concurrently in English text.	71
6.15	Matching 10 patterns concurrently in DNA sequence.	72

6.16	Matching 20 patterns concurrently in DNA sequence.	73
6.17	The performance differences between each mode of Clgrep when matching 20 DNA patterns concurrently.	74
6.18	English single pattern matching. The comparison between each mode of Clgrep.	75
6.19	English single pattern matching. Clgrep vs. The sequential implementation.	76
6.20	DNA multi-pattern matching. The internal comparison of Clgrep. . . .	77
6.21	DNA multi-pattern matching. Clgrep (BMH algorithm) vs. The sequential implementation.	78
6.22	DNA multi-pattern matching. Clgrep (QS algorithm) vs. The sequential implementation.	78
B.1	The results of single pattern matching.	87
B.2	The results of multi-pattern matching.	88

Chapter 1

Introduction

String matching is a special case of pattern matching [1]. The patterns are composed of only finite sequence of text symbols in string matching. Formally, it can be defined as finding one or more occurrences of a certain text pattern P in a large text T .

Applications requiring some forms of string matching function can be found everywhere, ranging from basic text editors, anti-virus applications, to Network Intrusion Detection Systems (NIDS). In many instances the process of string matching can be highly computationally intensive and time consuming [2]. To enhance the performance of string matching, not only algorithms but also processing techniques have been investigated in previous studies.

Recently, a number of research efforts have explored the use of Heterogeneous Parallel Computing in NIDS [3]. The idea of using either multi-core CPUs or many-thread Graphics Processing Units (GPUs) for parallel calculations is employed in several studies and presents positive results: by successfully utilizing the computational capability of GPUs and CPUs string matching is speeded up several times.

Grep is one of the most widely used command-line utilities for matching patterns in plain-text data [4]. Grep is challenged by the increasing amount of data but still merely supports working on a single core of CPU. Neither the Parallel Computing execution model nor GPGPU Computing is adopted in any variants of grep.

In fact, due to the nature of NIDS, in most studies [5, 6, 7, 8] only DFA-based (Deterministic Finite Automaton) string algorithms, specially Aho-Corasick algorithm [9] and its variants, have been evaluated with networking payload and NIDS signatures.

The potential matching performance of heuristics-based algorithms in natural language or DNA sequence remains an open question.

In this study, we explore how Heterogeneous Parallel Computing can be used in the context of grep-like string matching utility. A prototype called ‘Clgrep’ is implemented. As an alternative to grep, it features two algorithms, three processing modes, a number of optimizations, and can thus leverage the matching performance by both CPU and GPU.

In addition, a series of experiments is designed from a research perspective. Depending on Clgrep, the performances of different algorithms and processing modes are profiled in a variety of matching scenarios and discussed with comparisons. The uses of heuristics-based algorithms, specifically Quick Search [10] and Horspool algorithms [11], are investigated in the context of Heterogeneous Parallel Computing.

1.1 Motivations

This study is motivated by two factors:

1. The rapidly evolving computational capability of Parallel Computing devices, particularly modern GPU.
2. The increasing computational intensity of string matching processes.

At present, due to complexity and power consumption issues, the clock frequency of computing units is no longer evolving rapidly [12, 13]. In order to further increase the performance, the development of multi-core architecture is encouraged. Not only multi-core CPU but also GPU have become a de facto standard in PC. On the basis of the increasing computational capability and ubiquity, GPU with SIMD parallelism, has been widely accepted and applied in a number of areas. Overall, it shows outstanding result in the application where the same instruction is operated intensively [3].

On the other hand, although data are saved in various ways, text still remains one of the main forms partially for exchange information [14]. A significant amount of data is formatted in plain text and required to be filtered, searched, or verified. For instance, in molecular biology, in system administration, in networking, numerous pieces of information are generated and saved in linear files waiting to be processed every day [15, 16, 17, 18]. In the age of information explosion, the quantity of data in many fields tends to be increasing not in a linear but a more geometric manner.

String matching as a fundamental part is involved in most text-based processing described above, the efficiency of it is critical in many cases. Therefore, the goal of the study is to apply the idea of Heterogeneous Parallel Computing to develop a string matching utility specifically for intensive string matching, and moreover to investigate the use of specific string matching algorithms in the context of Heterogeneous Parallel Computing.

1.2 Research Objective

There are two objectives of this study.

1. The primary objective is to develop a string matching utility for computationally intensive exact string matching, and in addition to support the research (Objective 2) of the study.

The prototype should be able to perform string matching on heterogeneous computing units currently in a Parallel Computing manner. One or more heuristics-based algorithms should be implemented in the prototype, and support either matching a single pattern or a set of patterns. Moreover, a number of optimizations should be considered and examined.

2. The second objective is to investigate the use of heuristics-based algorithms in the context of Heterogeneous Parallel Computing.

In order to do so, an overview of string matching problem and algorithms will be made, and the classifications of different type approaches will be discussed. In addition, one or more typical heuristics-based algorithms will be implemented to perform string matching in Heterogeneous Parallel Computing manner. The matching performance of each algorithm and processing mode will be objectively measured in experiment, and discussed with comparisons.

1.3 Contributions

1. We developed an alternative to grep named Clgrep that features two efficient algorithms and multiple processing modes. Based on OpenCL, it offers great flexibility, not only being able to leverage the performance via using multi-core of CPU but also GPU. To the best of our knowledge, it is the first string matching utility based on OpenCL.
2. We investigated the use of two efficient heuristics-based algorithms in the context of Heterogeneous Parallel Computing. In the experiment of the study, both single pattern matching and multi-pattern matching are considered and, overall, more than 96 combinations of matching cases are included. It may be one of the most comprehensive experiments on this topic.

3. We critically reviewed different classifications of string matching algorithms. A number of constructive suggestions are advised. Furthermore, we examined a number of optimizations of OpenCL extensions that are rarely mentioned in any other studies.

1.4 Thesis Structure

The remainder of this thesis is structured as follows:

In Chapter 2, an overview of string matching problem and algorithm is made: the different forms of string matching problem are introduced, and the classifications of string matching algorithm are discussed; moreover, the algorithms that closely related to the later sections are presented.

In Chapter 3, the backgrounds of Parallel Computing, GPGPU Computing, and moreover OpenCL framework are given: 1) The idea of Parallel Computing is introduced, in addition the different executions models of Parallel Computing are presented; 2) the development of GPGPU Computing is introduced, then the differences between CPU and GPU are addressed; 3) the OpenCL framework is described in which the concepts that closely related to the study are highlighted.

In Chapter 4, the most related works of our study are reviewed, and, moreover, the research gaps in the previous studies are addressed.

In Chapter 5, the designs and implementation of Clgrep, the new string matching utility proposed in the study, are presented: the algorithms using in Clgrep are introduced, and the processing modes and matching procedure of Clgrep are described, and, further, the optimizations considered in the study are discussed.

In Chapter 6, the experiment methodology used in the study is introduced. The experiment results of the study are presented with discussions. The new approach Clgrep is examined in a variety of matching cases. More importantly, the use of two well-known heuristics-based algorithms are investigated in the context of Heterogeneous Parallel Computing.

In Chapter 7, the study is summarized, and, moreover, the findings of the study and the outline of future works is provided.

Chapter 2

On-line String Matching

String matching may be one of the most pervasive issues of the information science disciplines [15, 14]. The problem is very different if the text is preprocessed in advance or not. Generally speaking, if the large text is not structured in advance the problem refers to an on-line string matching problem. Otherwise, it normally refers to a string indexing problem. In the study we only focus on the problem of on-line exact string matching, and thus from now on, it is referred to as string matching in short.

During the study we widely investigated the string matching problem and algorithms in general: many previous studies related to the topic are reviewed, and, moreover, a number of applications using string matching algorithm are also examined. In this chapter, the different forms of string matching problem are distinguished then defined from the blurred boundaries in this area. Moreover, an overview of string matching algorithms is made in which the classifications of string matching algorithm are discussed. Finally, the algorithms closely related to the study are introduced specifically, in which the most important properties that are involved in our later discussion are emphasized.

2.1 The Definition of Exact String Matching Problems

According to the form of the matching problem, on-line string matching can be divided into: exact string matching; extended string matching, including regular expression matching; and approximate string matching also known as string matching with error [19]. In this study, only single pattern exact string matching and multi-pattern exact string matching are involved, and thus defined as follows.

Single pattern exact string matching is one of the most basic forms of string matching problems. Assuming P is a short text that we want to find, T is the large text that we search in, then the single pattern exact string matching problem is to find the exact occurrences of one pattern P in T .

Multi-pattern exact string matching is similar to single pattern exact string matching, but instead of one pattern, a set of patterns is searched for. Again, assuming P_1, P_2, P_3, \dots are the short texts that we want to search, and T is the large text that we search in, the multi-pattern exact string matching problem is to find the exact occurrences of a set of patterns including P_1, P_2, P_3, \dots , in T .

2.2 An Overview of Exact String Matching Algorithms

The first fast exact string matching algorithm named Morris-Pratt (MP) algorithm was proposed by Morris and Pratt in their well-known paper “A linear pattern-matching algorithm” in 1970 [20]. It was the first single pattern matching algorithm to have a linear time complexity in theory. Later, the MP algorithm was enhanced by Knuth, Morris, and Pratt in 1977, and this improved algorithm was named Knuth-Morris-Pratt algorithm (KMP) [21].

Soon after the KMP algorithm, another famous algorithm, named Boyer-Moore (BM), was introduced by Boyer and Moore in the same year [22]. Although the BM algorithm has a quadratic worst-case time complexity, it is participtionally efficient in practice and thus has later inspired many other algorithms.

Since 1977 many string matching algorithms have been proposed. Most of them are introduced in specific areas and discussed individually in the literature. The concepts behind each approach present a great variety in nature. Therefore, it is very difficult to draw the landscape of string matching algorithms. Previously, only few studies attempted to classify string matching algorithms into categories [23, 1, 19].

In [19], according to the way pattern is searched, string matching algorithms are grouped into prefix search approach, suffix search approach and factor search approach. Before discussing these approaches in details, the meanings of prefix, suffix and factor in terms of string matching are explained in a simple example below:

XYZ	XYZ	XYZ
X	Y	Z

Figure 2.1: Prefix Suffix and Factor.

Assuming X, Y , and Z are string, then X is the prefix, suffix, and factor of XY , YX , and YXZ , respectively.

Prefix Search Approach

In prefix search approach, the search is done from left to right. It always searches the longest same prefix in the text, as indicated in Figure 2.2. Among the most famous string matching algorithms, KMP is a typical prefix approach. It tracks the partial

matched prefix and thus avoids unnecessary comparisons.

This is the search text
Pattern \Rightarrow

Figure 2.2: The search window is shifted from left to right, and always searches the longest same Prefix.

Suffix Search Approach

Contrary to prefix search algorithm, in suffix approach the search is done backward, from right to left, as demonstrated in Figure 2.3. Furthermore, in this case, instead of prefix, the longest matched suffix is searched for. BM algorithm is the most well-known suffix search approach. Besides, Horspool algorithm used in Clgrep is also a typical suffix search approach [11].

This is the search text
Pattern
 \Leftarrow

Figure 2.3: Searching the longest matched Suffix from right to left.

Factor Search Approach

Like the suffix search approach, the search-of-factor approach is also done backward, but it searches for the shared factor between pattern and text. The algorithm used in NR-grep named Backward Nondeterministic DAWG algorithm (BNDM) is one of the typical factor search approaches [24].

The classification described above is straightforward. It intuitively reveals the nature of string matching algorithms. Nevertheless, it is likely that a number of the algorithms cannot be classified in such a simple way. For instance, the search of Two-Way algorithm is done in a specific order as its name implies [25]. It does not conform to any search type that is defined in the schema described above. Also, in the Karp-Rabin algorithm, hashing values of pattern and text are calculated. Then, the resemblances between pattern and text are examined as a whole rather than differentiated prefix, suffix or factor.

Therefore, we suggest that in order to further improve the schema, exceptions should be included. Algorithms, searched in a specific order or employing hash function, should

be distinguished from others.

In [23], another category is developed. According to the data structures that are used driving the matching, algorithms are categorized into four classes: 1) automaton-based algorithms, 2) heuristics-based algorithms, 3) hashing-based algorithms, and 4) bit-parallelism-based algorithms.

Automaton-based algorithms

An automaton-based algorithm firstly structures a finite state automaton from patterns, and then tracks the partial match of the pattern prefixes in the text during the search phase by state transition in the automaton. KMP and Aho-Corasick(AC) are well-known algorithms belonging to this category [9].

Heuristics-based algorithms

In heuristics-based algorithms, some characters in text can be skipped to accelerate the search according to specific heuristics. BM algorithm and its variants are all heuristics-based algorithms.

Hashing-based algorithms

Hashing-based algorithms, such as Karp -Rabin algorithm (KR), employ one or more hash functions and then compare pattern and text according to their hash values [26]. Instead of comparing characters one after another, they take advantage of roughly examining characters segment by segment. However, due to hash collisions, hashing-based algorithms required re-examination to identify exact match, in general.

Bit-parallelism-based algorithm

In Bit-parallelism-based algorithms, the operations of a non-deterministic finite automaton (NDF) are simulated to track the partial match of the prefix or the factor of the patterns. The Backward Nondeterministic DAWG Matching algorithm(BNDM) used in Nr-grep is a typical approach belonging to this category [27].

The schema introduced in [23] is well organized. It covers most techniques used in the string matching area. Besides, because it is compatible with the searching type-based schema described earlier, it can be applied simultaneously.

Overall, the nature of string matching algorithms is explored in the two classifications introduced in the section. The majority of string algorithms can be categorized into them. However, hybrid string matching algorithms employing more than one approach have not been considered and can hardly be classified in both schemas. To further

improve the schema, we suggest that hybrid approach classes should be included.

In this subsection, we presented an overview of the string matching algorithms. From discussing the classification of different approaches, the most important concepts related to our study were introduced. Although string matching algorithms are too many to be mentioned in a thesis or even listed here individually, algorithms in the same category are similar to each other, more or less.

2.3 The Algorithms Closely Related to The Study

A wide range of string matching algorithms is involved in the study; some of them are employed in Clgrep while others are used in the related works of the study. In this section, an introduction to these algorithms is given in which the most important properties of each algorithm that highly relate to the later discussions of the thesis, are explained in particular.

2.3.1 Naive Matching Algorithm: A naive approach

Naive algorithm is also known as Brute-Force algorithm [14]. As the most basic form of string matching algorithm, it exhaustively searches the sub-string of text that is identical to the pattern, as demonstrated in Figure 2.4. Assuming the length of pattern and text is n and m respectively, the time complexity of Naive string matching algorithm is $O(n * m)$ in any case.

This is the search text
Pattern
Pattern

Pattern

 \Rightarrow

Figure 2.4: The search process of Naive algorithm.

Despite Naive string matching algorithm being very inefficient from the time complexity point of view, it may minimize the other costs except comparison since there is no preprocessing phase in Naive algorithm and the comparisons can be done in any order.

2.3.2 Aho-Corasick Algorithm: A finite automaton multi-pattern matching algorithm

As mentioned earlier, Aho-Corasick (AC) algorithm is a typical automaton-based prefix searching multi-pattern matching algorithm. It structures a finite automaton table firstly at the preprocessing stage, and then tracks the partial prefix match during searching phase to avoid unnecessary comparisons [9].

Because the time complexity of AC algorithm is theoretically independent of the pattern set size, AC and its variants are widely used in NIDS and anti-virus scanning applications where a large set of patterns are required to be searched concurrently.

In the related works of the study, AC algorithm is implemented in more than one NIDS study to enhance the performance of intention defection. However, it is noteworthy that the size of DFA-table in AC algorithm is not constant but depends on the number of patterns, and thus structuring DFA table may be costly when a large number of patterns exist. As a result, the DFA-table of NIDS, which normally refers to “directory” in NIDS area, is built off-line in advance.

2.3.3 Karp-Rabin Algorithm: A hashing-based approach

Karp-Rabin algorithm (KR) was invented by Rabin and Karp in 1987 [21]. It utilizes a hash function to examine the similarity between text and pattern. For text of length n and pattern of length m , it has a worst-case quadratic time complexity $O(nm)$.

As a typical hashing-based algorithm, KR only filters possible matches. Due to hashing collision, in order to find an exact match, further examinations are required. However, similar to Naive algorithm, KR may take advantage of being able to search without order and special data structure support (shift table or DFA-table).

2.3.4 Wu-Manber Algorithm: A hybrid approach

Wu-Manber (WM) is a multi-pattern matching algorithm proposed in [28], specifically designed for a command line string matching utility – Agrep.

WM algorithm can be viewed as a hybrid approach because it employs both heuristics and hashing techniques. It improves the shift distance of a typical heuristics algo-

rithm by considering characters as blocks rather than looking at them one by one. As the probability of a block of characters appearing in the pattern is less than a single character, a longer shift distance is expected.

In WM algorithm, firstly, three tables are required to be precomputed at the preprocessing stage. Then, according to certain cases, a number of branches may occur to filter and verify matching candidates during the searching phase.

Although the concepts of WM algorithm are intuitive, its matching process is relatively complex and has many branches. The details of WM algorithm can be found in the original paper [28] as pseudo-code.

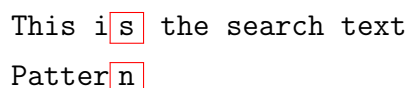
2.3.5 Boyer-Moore Algorithm and Its Variants: The root of heuristics approach

Boyer-Moore (BM) may be the most well-known heuristics-based algorithm overall [14, 15]. It was the first algorithm to introduce the use of heuristic rules in string matching. Many algorithms are inspired by it, including the algorithms employed in Clgrep.

Accordingly, BM algorithm, the foundation of heuristics-based approach, is introduced in this section. The search of BM algorithm is demonstrated first. Then, the two heuristics rules of it are explained in examples. Finally, the most important variants of BM algorithm are also mentioned.

The Search of BM Algorithm

As presented below, in BM algorithm the search is done backwards, from right to left. In the case of a mismatch, it uses two functions, named Good-suffix Heuristic and Bad-character Heuristic respectively, to shift the search window from left to the right.



This i **s** the search text
Patter**n**

Figure 2.5: BM algorithm is researched backward, “n” is firstly compared with “s”.

Bad-character Heuristic:

Let us consider the matching case in Figure 2.5. Apparently, “s” is not even a part of the pattern we are looking for. Thus, there is no point in comparing “s” to the rest of our pattern. The new comparison could start directly at “e” in the next phase, as demonstrated below.

Bad-character heuristic allows for avoiding many meaningless comparisons. In this case, the search window directly shifted seven characters, so a number of meaningless comparisons are avoided.

This is the s^earch text
 Patterⁿ

Figure 2.6: After applied Bad-character heuristic, the search window is shifted.

Good-suffixes Heuristic:

This^x is the search text
 istt^e is

Figure 2.7: A mismatch occurs after two exact matches.

In order to explain good-suffixes heuristic, let us consider the matching above where “is” is matched but a mismatch occurs when comparing “e” and “x”.

In this case, we already know the position of a good suffix of pattern “i”, so search windows can be safely shifted as below. Consequently, unnecessary comparisons are skipped, conforming to the good-character heuristic we described.

This^x is the search text
 isⁱtteis

Figure 2.8: Good-suffixes heuristic.

In the search phase of BM algorithm, both heuristics are applied. The shift distances from bad-character heuristic and good-suffixes heuristic are compared, then the longer distance is shifted by the algorithm.

The Variants of BM Algorithm

Despite the quadratic worst-case time complexity of BM algorithm, it has an excellent practical efficiency in general, and thus has been implemented in a variety of applications [19].

After the success of BM, a number of algorithms have been derived from it. The most famous variants include Horspool algorithm, Quick Search algorithm, Tuned Boyer-Moore, and Turbo Boyer-Moore algorithm [11, 10, 29, 30, 31].

As presented in several evaluations [32, 33, 34, 1], most of these variants are more efficient than the original BM algorithm. Among them, Horspool and Quick Search algorithms are employed in Clgrep and will be introduced later.

2.3.6 Summary

In this chapter, first, we clearly defined the on-line exact matching problem itself. Then, an overview of string matching algorithms was given. By introducing the classification of string matching algorithms, several important concepts were explained, and further the approach employed in the study was addressed in the landscape of string matching algorithms. Lastly, the algorithms closely related to this study were briefly described, and the most important properties of each algorithm were emphasized.

In short, every algorithm has advantages and disadvantages as suggested in a number of studies. No existing algorithm can be faster than another, in all cases.

Chapter 3

Open Computing Language (Welcome to The World of Heterogeneous Parallel Computing)

Heterogeneous Parallel Computing is defined as parallel computations using a variety of different types of computational units. This ideal is widespread: all computing systems, from mobile to supercomputers, are becoming heterogeneous parallel computing platforms. The computing community is racing to develop new languages and frameworks to ease the use of Heterogeneous Parallel Computing.

OpenCL is an open standard for Heterogeneous Parallel Computing System [35]. It includes a specification of language, API libraries and a runtime system, aiming to provide a uniform programming environment for developing portable code across different devices and architectures.

As a royalty-free standard, OpenCL is supported and conformed to by most mainstream hardware manufacturers including Apple Inc, Intel, AMD, Nvidia [13, 36]. Hence, it offers outstanding portability. OpenCL-based programs, implemented without vendor-specific extension, can be executed on any capable devices with conformant framework implementations. Furthermore, despite that both the standard and implementations of OpenCL are under rapid development, the backwards compatibility between each version is constantly maintained [37].

This study is inspired by the ideal of Heterogeneous Parallel Computing and based on

OpenCL framework. The relevant background is given in this chapter. The discussions start with a brief introduction of Parallel Computing and related execution models in which a basis of where OpenCL and the study are built is provided. Then, GPGPU Computing as an important aspect of the study is introduced. Finally, the architecture of OpenCL is presented in detail.

3.1 Parallel Computing

Parallel Computing is not a new concept; many problems are naturally distributed in parallel and thus can be solved efficiently in parallel [38]. Contrary to traditional serial computation, in Parallel Computing many operations are carried out simultaneously, problems are broken into discrete parts and then solved concurrently.

According to the level problems are partitioned to, Parallel Computing usually makes a distinction between bit-level parallelism, instruction-level parallelism, data parallelism, and task parallelism [38]. Among them, data-level and task-level parallelisms are more closely related to algorithms and applications being implemented, and thus are abstracted and explored to developers via frameworks such as OpenCL or CUDA [39, 40]. One of the most important objectives of these frameworks is to provide a consistent mechanism to address the issues introduced by data and task level Parallel Computing.

In Flynn's taxonomy, one of the earliest classifications of execution models and computer architectures, four classes of execution models are defined, according to parallel executions: Single Instruction, Single Data (SISD), Single Instruction, Multiple Data (SIMD), Multiple Instruction, Single Data (MISD), and Multiple Instruction, Multiple Data (MIMD) [41]. In order to more precisely discuss the differences between sequential execution and parallel execution, GPGPU Computing, and the SIMD parallelism approach used in Clgrep, the four execution models of Flynn's taxonomy are introduced as follows.

3.1.1 SISD

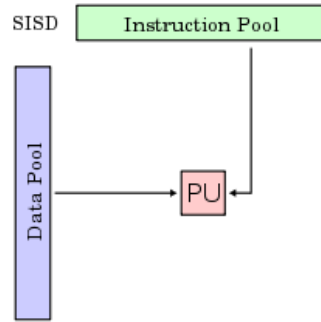


Figure 3.1: SISD

Single Instruction, Single Data (SISD) is demonstrated in Figure 3.1. In short, it is the traditional sequential order of execution, where instructions are operated on data in a sequence. Currently, most programs executed in SISD model include the sequential implementation of string matching algorithms in this study.

3.1.2 SIMD

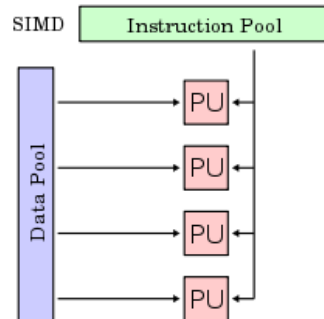


Figure 3.2: SIMD

Single instruction, multiple data (SIMD) is one of the typical forms of Parallel Computing. It describes computers or programs applying the same operation on multiple data simultaneously, and thus exploits a data-level parallelism.

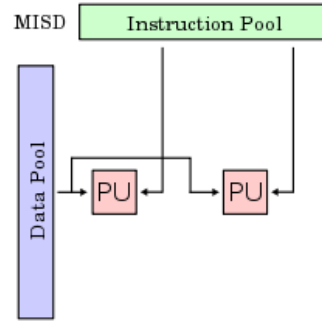


Figure 3.3: MISD

3.1.3 MISD

Multiple Instruction, Single Data (MISD) can be described as a pipeline. As demonstrated in Figure 3.3, in MISD, many computing units perform different operations on the same data. Unlike other models, MISD is a less common structure. Not many instances of this architecture exist, and one potential use is task replication [41].

3.1.4 MIMD

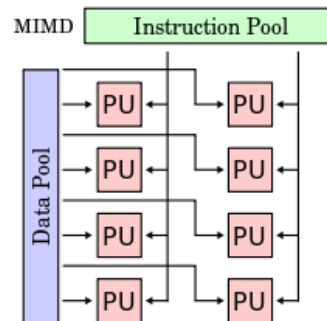


Figure 3.4: MIMD

Multiple Instruction, Multiple Data (MIMD) is the most obvious form of Parallel Computing, in which multiple data streams are processed by a number of processors asynchronously and independently. At any time, each processor may execute various instructions on different pieces of data in MIMD architecture.

After introducing the execution models of Flynn's taxonomy, it is important to notice that each model has its appropriate case. They are chosen depending on the prob-

lem. Moreover, they can be combined into hybrid models in practice or simulated by program.

3.2 GPGPU Computing

GPGPU Computing is defined as using Graphics Processing Unit (GPU) to perform computation traditionally handled by Central Processing Unit (CPU). In most cases, it can be viewed as a subset of Heterogeneous Parallel Computing, only if SIMD parallelism is employed [35]. As a concept, GPGPU Computing is employed in both this study and the related works to accelerate string matching. In this section, the development of GPGPU Computing is introduced. Moreover, the strengths and weaknesses of GPGPU Computing is discussed.

GPU was first introduced in the 80s to offload graphics-related processing from CPU. Initially, GPU was only used for providing several hardware implementations of fixed-functions for graphics acceleration, and it was not programmable.

Nevertheless, it has changed in the last decade. In 2002, after the first programmable pixel shader was released, developers were first able to write their own shader programs. At this point, a number of shader languages were developed to simplify the GPGPU programming process including Cg, GLSL, and HLSL. Then, in 2003, several GPGPU languages (BrookGPU and Sh) were introduced specifically toward GPGPU application development. Later, in 2006, both Nvidia and ATI released their proprietary framework aimed at GPGPU Computing. Now that multi-core CPUs are widely used, GPGPU Computing has evolved into a form of Heterogeneous Parallel Computing, in general.

Nowadays, a Heterogeneous Parallel Computing system consisting of both multi-core CPU and many-thread GPUs may be the most typical form of PC. The main difference between these two types of processor is that multi-core CPU is a MIMD processor designed to execute multiple streams of instructions asynchronously and independently, while GPU is SIMD processor optimized to execute the same stream of instructions over multiple data simultaneously.

Although it is possible to develop GPGPU Computing programs that work only in a non-parallel manner, it very unlikely, however, achieves a meaningful performance. Most applications attempt to exploit the power of GPU work in the SIMD Parallel Computing model, just as the massive parallel architecture of GPU indicated [3].



Figure 3.5: The architectures of CPU and GPU.

Besides, the threading management and the memory sub-system of GPU are also different from CPU. In modern GPUs, memory is divided into dedicated regions depending on specific uses, threads are efficiently managed by hardware: switching between threads in GPU only costs a few cycles. But, due to the cache and I/O limitations of GPU, transferring data is much more expensive on GPU than CPU.

One of the most obvious benefits of GPGPU Computing is the low cost of outstanding performance. According to recent research [12, 42, 5], compared with other alternatives GPU may be the most economic choice per GFLOPS. In addition, the ubiquity of GPUs also plays an important role in motivating the use of GPU. At present, GPGPU Computing applications not only target the Throughout Parallel Computing but a variety of day-to-day uses.

3.2.1 CUDA

Due to the increasing computational capability and flexibility of GPU, a number of frameworks have been introduced for GPGPU Computing development [43, 44, 45, 46, 47], one of the most notable being CUDA. It is employed in most, if not all, of our related works, thus an introduction of it is presented specifically in the following.

Compute Unified Device Architecture (CUDA) is a General-Purpose Parallel Computing framework introduced by Nvidia in 2006 [46]. It provides an environment that allows programmers to develop Parallel Computing application in high-level language.

CUDA was originally released as a platform for GPU Computing development only. But one year later, after the CUDA x86 compiler was released to support multi-core CPU in 2007, it had been officially marketed as Heterogeneous Parallel Computing platform. However, unlike OpenCL, CUDA programs are exclusive to Nvidia GPU.

Currently, CUDA provides both data parallelism and thread parallelism via three key abstractions: thread group hierarchy, shared memories and barrier synchronization. Most concepts of CUDA are very similar to OpenCL, except different terminologies are used. Besides, it is noteworthy that OpenCL and CUDA share a range of computational interfaces with each other.

3.3 The Architecture of OpenCL

As previously introduced, this study is based on OpenCL framework, an open royalty-free standard for Heterogeneous Parallel Computing. Therefore, before we present Clgrep, the necessary background of OpenCL is given.

As a specification OpenCL is defined from four aspects: 1) The Platform Model, 2) The Execution Model, 3) The Memory Model, and 4) The Programming Model.

3.3.1 The Platform Model

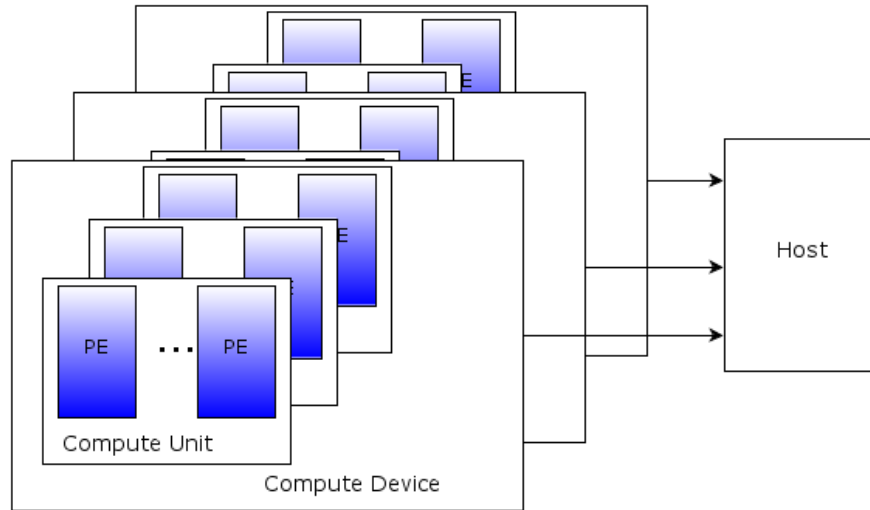


Figure 3.6: The platform model of OpenCL

As demonstrated in Figure 3.6, the Platform Model of OpenCL consists of two parts: host and compute device. The host is the overall controller. It connects to one or more OpenCL devices. The compute device is a collection of compute units (CUs) that can be further divided into Processing Elements (PEs). The actual calculations occur in PEs, eventually.

3.3.2 The Execution Model

The execution model of OpenCL also consists of two parts: host program and kernel. The host program is used to define the context for the kernels and manages kernel execution, and it is executed on the host device only. The kernel, on the other hand, is the basic unit of executable code and thus can be executed on one or more devices.

In OpenCL, when a kernel is submitted from the host for execution, an index space called ND-Range is defined. As presented in Figure 3.7, in ND-Range, each work-item is identified by a unique global ID in all and a unique local ID in the work-group. Furthermore, each work-group also is assigned with a unique work-group ID. Finally, the work-item in a given work-group executes in a single CU concurrently.

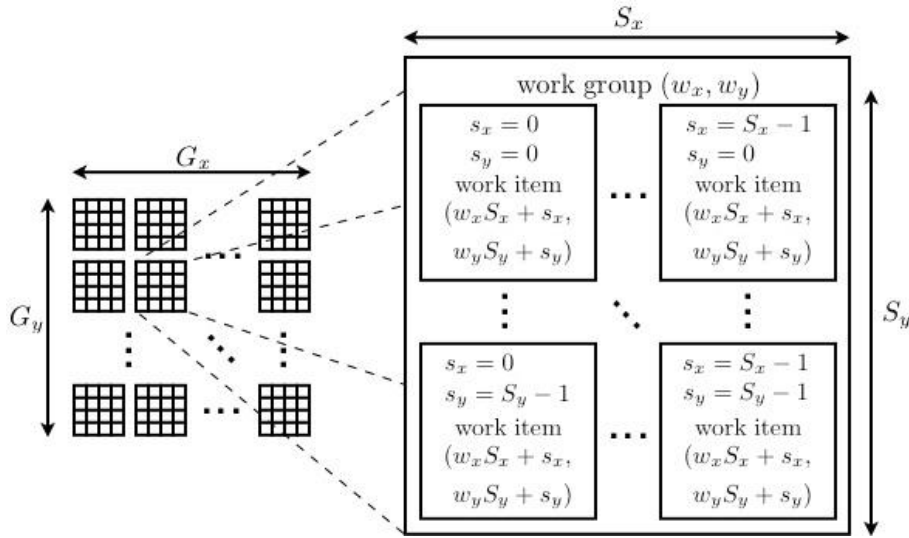


Figure 3.7: ND-Range

3.3.3 The Memory Model

The Memory Model of OpenCL is defined in a hierarchical structure in which memory is abstracted into four distinct regions:

Global Memory is accessible to all work-items in all work-groups. Depending on device capability, it may be cached. In the context of CPU, it simply is mapped onto the Host Memory (System Memory).

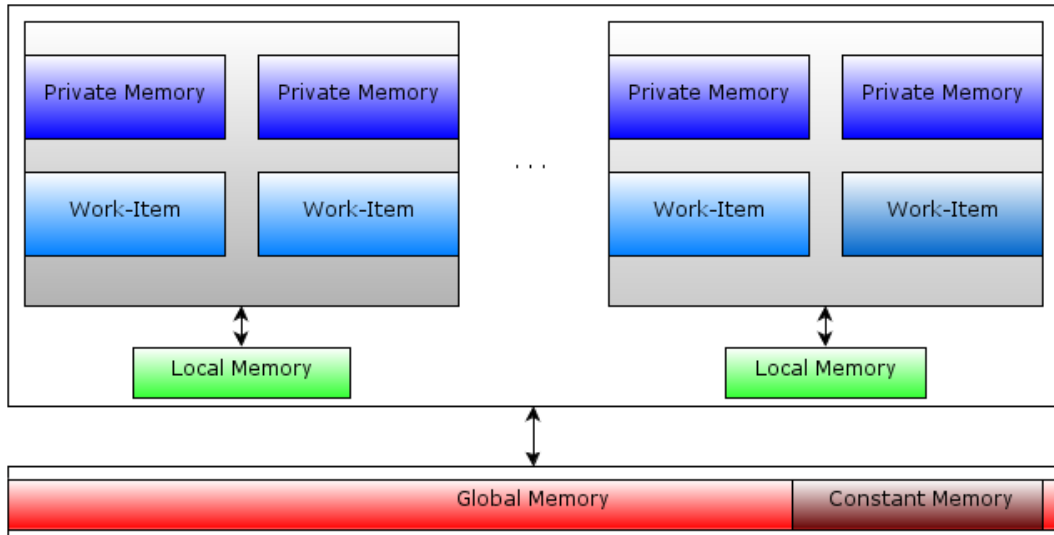


Figure 3.8: The memory hierarchy of OpenCL

Constant Memory also can be accessed by all work-items but where the values have to remain constant during the kernel execution. Similar to Global Memory, it may also be cached according to hardware capability.

Local Memory is private to a work-group and normally implemented as a dedicated region of memory in GPU. However, in terms of CPU, Local Memory is just a special mapped area of Global Memory.

Private Memory is only accessible to a single work-item. It normally is the fastest memory region available.

Apart from the memory hierarchy, a relaxed consistency memory model is employed in OpenCL. The state of memory for each work-item is not guaranteed to be consistent across the collection of work-times at all times. To overcome the limitation, a barrier-based synchronization mechanism is used.

Besides, currently, there is no Dynamic Allocation permitted in the kernel code. It is a very important limitation of OpenCL, especially for implementing string matching algorithms.

3.3.4 The Programming Model

In OpenCL, two programming models are specified:

- Data parallel programming model
- Task parallel programming model

In data parallel programming model, NDRange index space is defined, which then maps subset of data to work-items. However, unlike strictly data parallel mode, there is no one-to-one basis in OpenCL.

On the other hand, in task parallel programming model only a single instance of the kernel is executed, and no NDRange index range is defined. Task parallel programming model is much less common than the data parallel programming model, rarely used in the previous studies.

After we described OpenCL from four core aspects, it is important to note that as an open standard, OpenCL specifies fundamental concepts and requirements; more capabilities beyond those defined in the standard depend on the implementation of each vendor.

3.4 Summary

In this section, we firstly developed the context of the study via discussing Parallel Computing in general. Then, we introduced the development of GPGPU Computing, and addressed the differences between CPU and GUP. Lastly, we described the architecture of OpenCL, where a number of important concepts highly related to the design and implementation of Clgrep were emphasized.

Chapter 4

Related Works

In the study, we aim to develop a string matching utility and investigate the use of heuristics-based matching algorithms in the context of Heterogeneous Parallel Computing. In order to determine what is required from such a study, a review of the current state of related works is made.

The works closely related to the study can be grouped into three categories:

- The study of traditional grep-like tools
- The study of GPGPU Networking Intrusion Detection Systems (NIDS)
- The study of evaluating string matching algorithm in the context of GPGPU Computing

In this chapter, these areas are critically reviewed, and some issues and ideas that previous works have not addressed are highlighted.

4.1 Traditional Grep-like Tools

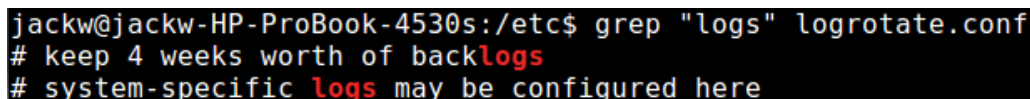
The name grep-like tool, comes from a famous command-line utility for searching patterns of plain-text [48]. The original grep was developed for the Unix OS in 1973 by Ken Thompson. Since then, grep-like string matching utility has been a standard tool in most Unix-like OS. They are used in a wide range of domains and numerous cases, including: text editing, program comprehension, software maintenance, file systems indexing, system administration, genome sequence alignment and the like [4].

There are a number of grep-like string matching utilities available. Early variants included egrep and fgrep on Unix System. Later, both of them were improved and merged into GNU grep, the most widely used grep program. After GNU grep, Agrep was developed between 1988 and 1991 by Udi Manber and Sun Wu as the first approximate string matching tool [49, 28]. Then in 2000 another influential grep variant NR-grep was presented by Gonzalo Navarro in his research paper “NR-grep: A Fast and Flexible Pattern Matching Tool” [24].

In this section these most well-known grep-like tools are reviewed, and their algorithms and concepts highlighted.

4.1.1 GNU grep

At present, GNU grep as a part of GNU tool-chain is included in almost every GNU/Linux distribution, not only to be used explicitly by users but also seamlessly by applications such as vim and geany [4]. As its name implies, GNU grep is a GPL licensed open-source project and thus can be used freely. After having been contributed to by a list of developers, grep has become arguably the most feature-rich string matching tool offering more than 20 control options.



```
jackw@jackw-HP-ProBook-4530s:/etc$ grep "logs" logrotate.conf
# keep 4 weeks worth of backlogs
# system-specific logs may be configured here
```

Figure 4.1: GNU grep: print lines matching a pattern.

GNU grep employs several algorithms to different matching tasks. According to the latest source code of it (version 2.10), an algorithm very similar to Horspool called Boyer-Moore-Gosper, is dedicated to exact single pattern matching. Then, both Commentz-

Walter algorithm and Aho-Corasick algorithm are employed for multi-pattern matching. Moreover, a DFA-based algorithm similar to Aho-Corasick is used for regular expression matching.

4.1.2 Agrep: Approximate grep

Agrep is another famous variant of grep, well-known for its flexible approximate searching function. Unlike GNU grep, Agrep is a proprietary program belonging to University of Arizona. In the information retrieval software GLIMPSE, it is employed as the engine for searching and indexing large file systems [16].

Different from the tradition of grep, Agrep does not stick to the basis of line. Instead, it can search according to user-specified delimiters. In terms of matching efficiency, Agrep utilizes a collection of algorithms to optimize the performance for the different cases, including: Horspool algorithm for most exact single pattern matching and part of multi-pattern matching tasks, an original algorithm named Wu-Mamber for multi-pattern matching, and an algorithm based on Bitap algorithm for approximate string matching and regular expression matching.

The design of Agrep is sophisticated, particularly in the branching process of algorithms selection. A more comprehensive description can be found in [49].

4.1.3 NR-grep : Non-deterministic Reverse grep

In 2001 NR-grep was presented for efficient search of complex patterns [24]. Unlike GNU grep or Agrep, NR-grep is based on a single uniform algorithmic concept: the bit-parallel simulation of a non-deterministic suffix automaton. It employs only the Backward Non-deterministic Dawg Matching (BNDM) algorithm to solve both exacting matching, approximate matching and regular expression matching.

The performance of NR-grep is competitive with other alternatives particularly when matching long complex patterns. However, it is important to notice that the BNDM algorithm is limited by the pattern length. When the pattern length is longer than the memory-word size of the machine, the performance of BNDM will be significantly decreased [14]. Like Agrep, NR-grep is also a proprietary program that can only be used on a non-profit basis.

In this section, we reviewed the major members of the grep-like string matching family. For each of them, the features, the main algorithms, and the limitations are introduced. Overall, GNU grep, Agrep and NR-grep are all well-designed and still playing an important role in Unix-like OS. Most of them focus on algorithm design and selection rather than the processing model. Advanced concepts such as GPGPU Computing or multi-core technology have not been adopted in this area so far.

4.2 GPGPU Networking Intrusion Detection Systems (NIDS)

In NIDS, numerous patterns are matched to detect intrusion. For instance, Snort, one of the most widely used NIDS, already contained over 20000 rules on 19 Jun 2012¹, and the number is still increasing every month. According to [50, 51], over 70% of the total CPU processing time of modern NIDS is used for matching patterns. In consequence, efficient string matching is a very important subject in the NIDS area.

To overcome the computational limitation of the CPU in NIDS, many research efforts have been contributed to this profitable area, and specialized hardware has been used in NIDS for years. More recently, a number of studies have attempted to utilize modern GPUs for high throughput networking intrusion detection and achieved a very positive result.

4.2.1 PixelSnort: The First Attempt

In [52], GPU was firstly used for offloading computation from CPU to improve the performance of NIDS. A GPU-based Snort named PixelSnort was developed in Cg language.

However, compared with the conventional Snort, only marginal improvement is presented by PixelSnort. The result is limited by both the algorithm and the hardware used in the study, particularly the computational capability of GPU at the time. In NIDS signature matching, numerous patterns are required to be matched, and thus as a single pattern matching algorithm the simplified Knuth-Morris-Pratt (KMP) algo-

¹According to the official website of Snort.<http://www.snort.org/>

rithm of PixelSnort may not be reliable. Besides, although the Nvidia 6800 GT GPU used in the experiment was a high-end GPU back then, it is only equipped with sixteen 400 MHz pixel shader processors, incomparable to new generation GPUs used in later studies.

The results of PixelSnort have been criticized by some studies [5, 51], but as it is the the first GPGPU NIDS research, we believe it deserves more respect, specifically when considering the development difficulties with Cg shader language.

4.2.2 Evaluating GPUs for Network Packet Signature Matching: An Analysis of GPU-based String Matching

A detailed architectural and micro-architectural analysis of GPU-armed NIDS signature matching was presented in [42]. There SIMD processing for NIDS was firstly quantified.

In the study a programmable prototype of NIDS is developed based on CUDA. Then, the performance of GPU-armed string matching is examined: a Nvidia G8800 GTX GPU is compared with both a Xeon E5345 CPU and a Niagara-based 32-threaded system.

This study discusses the advantages and challenges of GPU-based string matching, and presents a detailed analysis of memory accessing optimization. A positive result is shown at the end of the study: compared with CPU and Niagara-based 32-threaded system, an observed two to nine times better performance is achieved by the prototype system of the study.

However, with 1) string algorithm being merely referred to as DFA or XFA approaches without specific details in the discussion of the study, the algorithms examined in the experiment remains unclear. Also, 2) the description of experiment settings and procedure are not detailed. It is therefore very difficult to repeat the experiment of the study.

Besides, it is noteworthy that 1) this study focuses on theoretical analysis rather than practical experiment; 2) as a NIDS study, only string matching comportment of NIDS is considered in the study.

4.2.3 Gnort: A High Performance GPGPU NIDS based on Snort

In 2008 a more comprehensive GPGPU NIDS study was presented in which a high performance GPU-armed NIDS named Gnort was proposed [5].

In the study an analysis of the architecture of GPU and typical processing stages of NIDS is given. The Aho-Crasick string matching algorithm is implemented based on CUDA framework. Then, in the experiment, the performance of the prototype system Gnort is compared with other algorithms including both BM and KMP algorithms on GPU. The scalability of Gnort is evaluated in both laboratory environment and real scenario. Besides, several optimizations such as utilizing texture cache and using DMA are discussed in the study, although some of them are not employed in Gnort.

Overall, the experiment results of the study suggest GPU-armed string matching is very efficient. A maximum throughput of 2.3 Gbit/s is achieved by Nvidia GeForce 8600GT video card by Gnort.

Inspired by the success of Gnort, a number of studies [6, 7, 8, 49, 28] aim to enhance the performance of Gnort. Most prototype systems of those studies, if not all, are based on CUDA and employ Aho-Corasick algorithm or its variants. Some of the discoveries are remarkable in terms of NIDS, but not related to our study.

The common scenario of NIDS is different from grep-like string matching utility where only networking payloads and large numbers of NIDS signature patterns are considered. But the performance evaluation methodology and the optimizations strategy used in these GPGPU Computing NIDS studies are closely related to our study.

4.3 String Matching Algorithm Evaluation on GPU

Algorithm evaluation is a very costly but important subject in the string matching area. A number of experiments on string matching algorithms have been reported previously [32, 33, 34, 53, 1], but only a few studies focus on evaluating string matching algorithm in the context of GPGPU Computing.

4.3.1 Evaluating The Performance of Approximate String Matching On GPU

In [54], an improved NFA Bitap algorithm was implemented based on CUDA. Then, the efficiency of it was evaluated in DNA sequence matching.

The experiment of the study is well-designed. It measures the matching performance of each approach by practical time, and employs real genome sequence as matching text and patterns: a sample set of DNA from the fruit fly is selected as the matching text (about 160 MB), and a number of arbitrary substrings are generated as patterns.

After a number of experiments, when the computation intensity is increased to the peak (the edit distance of approximate matching is 15), compared to the sequential implementation of the same algorithm on AMD Opteron 2.4 GHz CPU, a significant speed-up is achieved by Nvidia Tesla GPU.

Besides, memory accessing optimizations with the work flow of CUDA framework are also clearly explained and discussed in the study. Unlike any NIDS study, instead of using the texture memory this study utilizes the shared memory of CUDA GPU to enhance the performance of I/O operation during matching.

To the best of our knowledge, [54] is the first study to investigate the use of GPU for approximate string matching. However, in the study, the algorithm implementation details are not available, and the way of parallelization is not mentioned at all.

More importantly, several critical assumptions are held in the experiment. Firstly, it assumes the size of NFA table has to be fitted into the shared memory of GPU. Secondly, it assumes the edit distance of the approximate string matching must be smaller than 16.

Besides, Tesla, the GPU used in the study, is a workstation-level GPU from Nvidia.

It is incredibly fast in both core frequency and memory bandwidth and makes it very difficult to compare the results of this study with others.

4.3.2 String Matching on a multi-core GPU with CUDA

In [55], a number of well-known string matching algorithms were implemented based on CUDA, including Naive, KMP, Horspool, and QS algorithms. Then, the performance of these algorithms was evaluated in the context of single pattern DNA sequence matching.

In the experiment of the study, a very high-end Nvidia GTX 280 GPU is compared with a 2.4 GHz Intel Xeon CPU. The GPU has 30 multiprocessors and 240 cores in total, and can support as much as 30720 active threads. Considering the repeatability of the experiment three specific genome sequences are used with randomly constructed patterns. The study calculates the result by practical running time. It is the total time in seconds for algorithm to locate every occurrence of a pattern in text. Furthermore, to control the error of experiment, the final results of the study are averages of 100 runs.

Overall, a positive result is demonstrated in the study. A 24 times speed-up of Naive algorithm is reported in an extreme case during the experiment. The study suggests the GPU-armed implementations of Horspool, Naive, KMP, and QS algorithms are all more efficient than their corresponding serial implementations, particularly when matching short pattern in a large text.

Besides, the study investigate the use of share memory to improve matching performance. Based on the experiment results, it suggests that by using shared memory of CUDA GPU, the parallel implementation of string algorithms may be accelerated by 2 to 24 times depending on algorithm. Among these algorithms examined in the study, Horspool and QS benefit most from the use of shared memory.

‘String Matching on a multi-core GPU with CUDA’ is the first study that focuses on the heuristics-based string matching algorithms, and implements Naive, Horspool and QS algorithm based on CUDA. The experiment methodology of the study is well-designed.

Unfortunately, the result presentation is slightly unclear in the report of the study. Specifically, there is no corresponding explanation when no speed-up is achieved by GPU and visualized in the chart. Besides, only DNA sequence single pattern matching

is considered in the study.

4.4 The Research Gaps

In this chapter we reviewed the most related works of the study crossing different fields. A number of research gaps in the previous studies are addressed and thus summarized in the following.

Gap 1: Although GPGPU Computing has been employed in a number of NIDS studies for improving matching performance, most traditional grep-like tools still merely support a single core CPU. The concept of Heterogeneous Parallel Computing has not been adopted in this area.

Gap 2: Previously, most studies attempt to utilize only the GPU for accelerating string matching, and always compare the performance of GPU to a single core of CPU. Processing on both CPU and GPU currently in a heterogeneous platform has not even been considered, and thus the use of such an approach remains as an open question.

Gap 3: As a DFA-based approach Aho-Corasick algorithms were employed and evaluated in most NIDS studies, but on the other hand, heuristics-based algorithms such as Horspool and QS were merely evaluated in DNA sequences single pattern matching. The uses of these approaches in terms of multi-pattern matching and English text matching have not been investigated.

Gap 4: The majority of GPGPU Computing studies are based on CUDA, a proprietary framework exclusive to Nvidia hardware. On the other hand, OpenCL as a royalty-free open standard has not been widely considered in the past.

Chapter 5

Clgrep: A New Approach

Inspired by the idea of Heterogeneous Parallel Computing, Clgrep, a new exact string matching utility, is proposed in this study. It is developed as an alternative to grep for computationally intensive string matching, and in addition is a tool for supporting the research purposes of the study.

Clgrep is a command-line string matching utility. It inherits the classical form of grep family and can seamlessly collaborate with many other programs available in Unix-like OS. But different from any other variants of grep, Clgrep is based on OpenCL framework and thus is able to fully utilize the computational capability crossing heterogeneous computing devices in parallelism. In Clgrep, two efficient algorithms and three processing modes can be easily switched with options. Either single pattern matching or multi-pattern matching can be executed on CPU, GPU, or even CPU and GPU concurrently.

In this chapter, the unique designs and implementation of Clgrep are presented in detail. We start with introducing the algorithms used in Clgrep. Then, the different processing modes and matching procedure of Clgrep are described. Moreover, we discuss the optimizations considered in this study. Finally, the architectural elements of Clgrep are introduced.

5.1 The Algorithms of Clgrep

Algorithms are one of the most important aspects of string matching utility. After we reviewed many studies and applications, two efficient heuristics-based string matching algorithms are employed in Clgrep, including Horspool and QS algorithms. In this section, a brief introduction to these algorithms are given, and the important properties of them are emphasized. Furthermore, the multi-pattern strategies of Clgrep are also introduced, and the strengths and weaknesses are addressed.

5.1.1 Horspool Algorithm

The Horspool (BMH) algorithm was introduced in 1980 by Nigel Horspool and originally named as Simplified Boyer and Moore algorithm [11]. It is a simplification of BM algorithm in which only the bad-character shift of BM algorithm is applied (A detailed description of bad-character can be found in Chapter 2 of the thesis.). Similar to the BM, Horspool algorithm also has a quadratic time complexity in worst cases, but behaves in a sub-linear manner in practice.

As can be seen in the related work section, due to the outstanding efficiency and similarity, Horspool algorithm is widely used in applications. It is believed that Horspool is one of the most used heuristics-based string matching algorithms overall, being implemented within the search function of many text editors and Integrated Development Environments (IDEs) [14]. The detailed description of Horspool algorithm can be found in [11].

5.1.2 Quick Search Algorithm

Quick Search (QS) algorithm was proposed in 1990 by Sunday [10]. It also can be viewed as a variant of BM algorithm, in general. However, unlike BM algorithm, the search of QS algorithm is done forward, from left to right. Moreover, the bad-character heuristic of QS algorithm is also slightly different from BM algorithm. During the search phase of QS algorithm, instead of shifting the search window upon the last character, the shift of QS algorithm is determined according to the character right next to the last character of the search window, and thus more potential shift distance is expected.

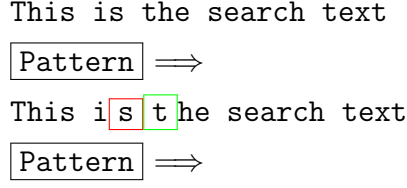


Figure 5.1: The search window of QS algorithm is shifted from left to right. But instead of shifting the search window upon the last character, the shift of QS algorithm is determined according to the character right next to the last character of the search window. (In this case it shift upon "t" not "s".)

In several studies QS algorithm demonstrates a very strong performance, especially for matching short patterns in large texts alphabets [32, 33]. It is noteworthy that QS algorithm has been listed on the TODO list of GNU grep for decades. A comprehensive description of QS can be found in the original paper.

5.1.3 The Multi-pattern Matching Strategies of Clgrep

At present there is no algorithm specifically implemented in Clgrep for matching a set of patterns. Instead, QS and Horspool algorithms are also used for multi-pattern matching but in a distinctive manner. While matching for a set of patterns by Clgrep, the shift distances of each pattern are tracked, and then only the minimal distance is shifted to avoid losing any exact match by algorithms. Apparently, there are a number of advantages and limitations in such an approach the details of which are discussed as follows.

Advantages:

1. Firstly, compared to most multi-pattern matching algorithms, the pre-computation stage of QS algorithm or Horspool algorithm is much simpler and more efficient.
2. Secondly, because the heuristics table of Quick Search or Horspool algorithm requires only a limited space, it can be easily fitted into the Local Memory of the OpenCL computing device to avoid continually accessing the Global Memory, and thus significantly improves the performance.

3. Furthermore, as shown in [56] and [5], the performance of applying BM or its variant algorithms to a set of patterns is not unacceptable. Actually, it is fairly efficient when the number of patterns is less than 100, and arguably more efficient than AC algorithm when the number of patterns is below 30.
4. Besides, multi-pattern algorithms are more complex and branching than QS or Horspool algorithm. As result, they may not be appropriate in GPGPU Computing application where I/O operations are much more costly than computations.

Disadvantages:

Apparently, QS and Horspool algorithms are not specifically designed for multi-pattern matching. They require more computational capability particularly when matching a very large set of patterns concurrently.

5.2 The Processing Mode and Matching Procedure of Clgrep

The design and implementation of Clgrep are different from other alternatives of grep. In order to understand the strengths and weaknesses of Clgrep, and later explain the experiment results of the study, the processing modes and matching procedure of Clgrep are introduced in this section.

5.2.1 Processing Modes

As listed in the table below, Clgrep offers three processing modes including CPU mode, GPU mode, and CG mode. Each mode can be easily switched between by providing options in commands, and the matching tasks can be partitioned and executed on the multi-core CPU, GPU, or even CPU and GPU simultaneously. Despite that Clgrep can be extended to support multiple CPUs and GPUs, it assumes only one CPU and GPU by default.

Processing Mode	GPU Mode	Multi-core CPU Mode	CG Mode
Matching is executed on	GPU	CPU	Both CPU and GPU

Table 5.1: The processing modes of Clgrep.

5.2.2 Matching Procedure

Due to the nature of OpenCL-based program, the matching procedure of Clgrep is verbose. In addition to traditional sequential application, a number of extra processes are required by OpenCL framework and the Parallel Computing mechanism of Clgrep.

In this section, we distinguish and introduce the matching procedure of Clgrep in three stages: Preprocessing (Pre-computation) stage, Matching stage, and Post-processing stage.

Preprocessing Stage:

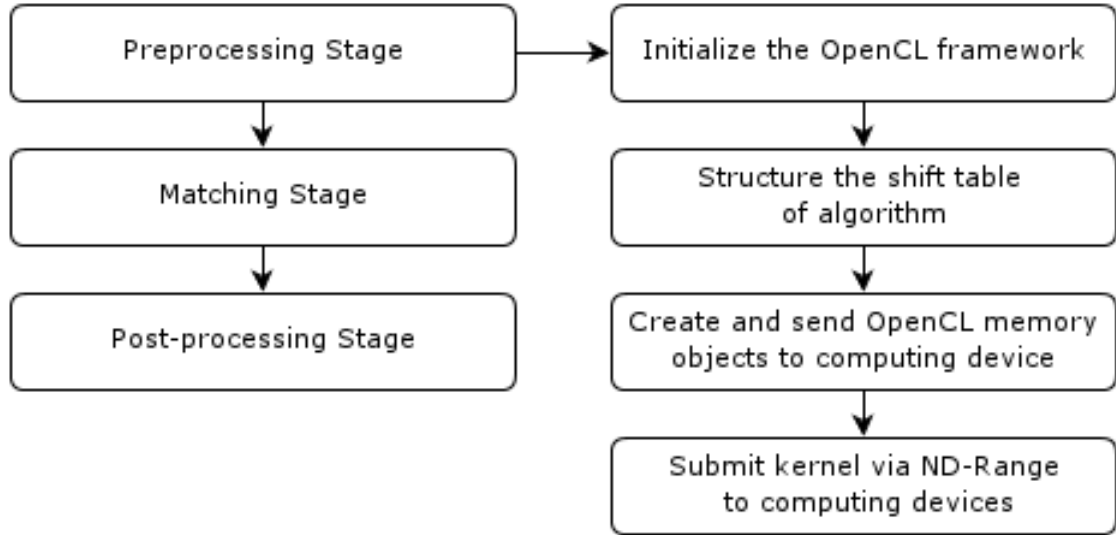


Figure 5.2: The preprocessing stage of Clgrep.

As demonstrated in Figure 5.2, at the beginning of the preprocessing stage the OpenCL framework is initialized in which the programming context is created for Clgrep, and computing devices are prepared for calculation. In addition, the host program and kernels are built.

Next, according to the pre-computation function of each algorithm the shift table is structured. Moreover, the OpenCL memory objects are created from matching text and patterns then transferred from the host to the Global Memory of computing devices. Finally, ND-range is defined and kernel instances are submitted with arguments to computing devices.

Matching Stage:

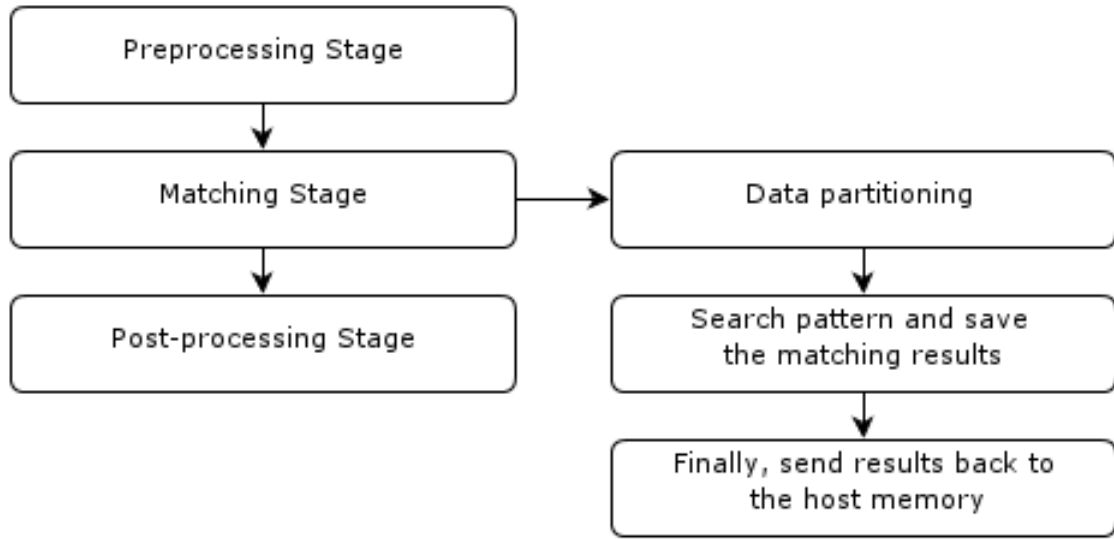


Figure 5.3: The matching stage of Clgrep.

During the matching stage, tasks are parallelized in the SIMD manner. As demonstrated in Figure 5.4, text is partitioned into blocks with overlap and then processed by computing units respectively.

In Clgrep, a hierarchy subsystem for saving and reporting matching results is employed to control the expensive I/O costs. 1) When an exact match is addressed, the result is saved in the Local Memory of computing device immediately. 2) Whenever the tasks assigned to each work-item are finished, the collected results of a certain work-item are moved from Local Memory to Global Memory, temporarily. 3) At the end of the matching stage, when all the exact matches are located, the results are sent back to the host at once.

The detailed reasons for employing the complex hierarchy will be explained later when we discuss the optimizations of Clgrep.

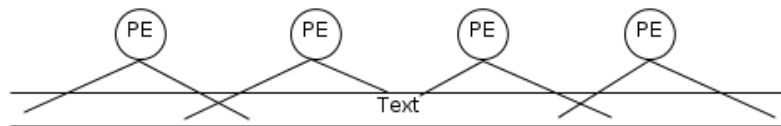


Figure 5.4: Data Partitioning.

Post-processing Stage:

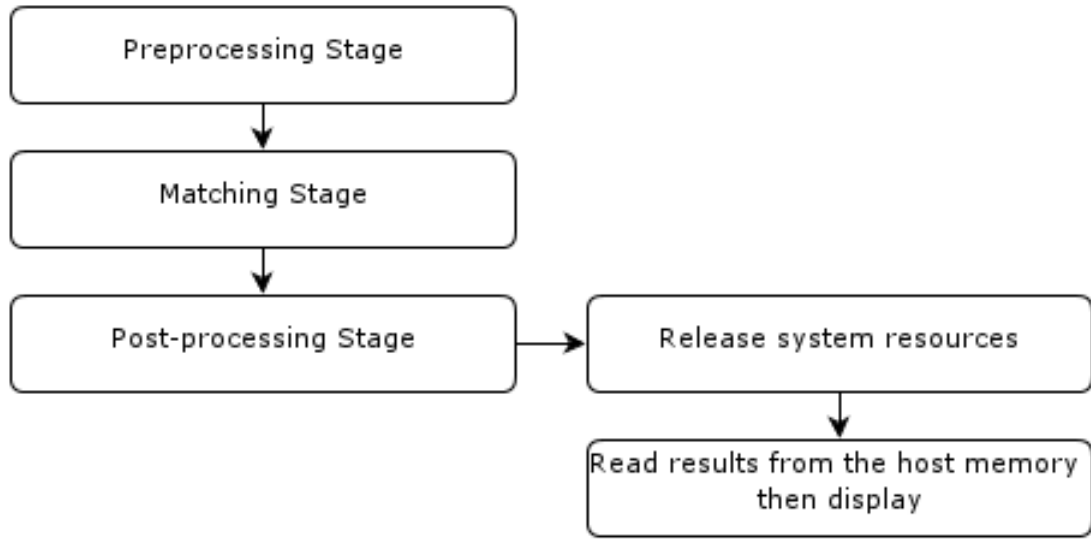


Figure 5.5: The post-processing stage of Clgrep.

As presented in Figure 5.5, the post-processing stage of Clgrep is simple. After all the matching tasks distributed via ND-range are finished, the system resources used by OpenCL are released, then the matching results are read from the Host Memory and displayed through standard output.

5.2.3 The CG Mode of Clgrep

The behavior of CG mode is slightly different from others. As demonstrated in the figure below, when a single pattern is matched in CG mode, the text is divided and processed separately by CPU and GPU.

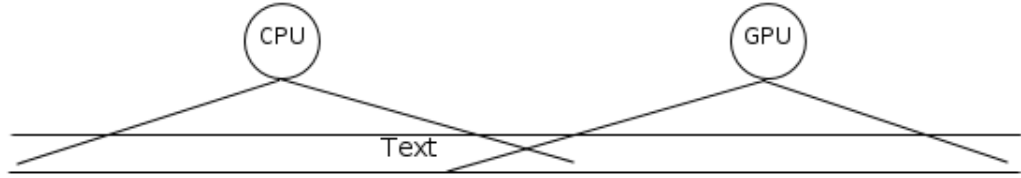


Figure 5.6: Matching text is divided when a single pattern is matched in CG mode.

On the other hand, when a set of patterns is matched in CG mode, instead of dividing the text, patterns are grouped and processed separately. This strategy may potentially increase the shift distance of the algorithms that employ bad-character heuristic, especially if the patterns are grouped optimally.

Currently, there are no specific grouping strategies built in Clgrep. Patterns are separated according to the length, where shorter patterns are assigned to GPU and the opposites are assigned to CPU, because the computational cost of matching shorter patterns is more likely to be more expensive.

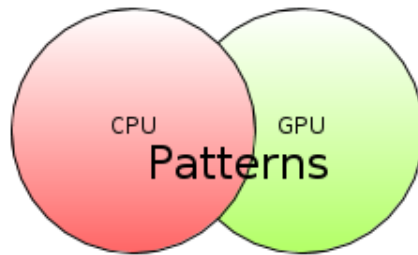


Figure 5.7: Patterns we are searching for are grouped then assigned to different processors.

5.3 Optimizations

Optimizations are critical to the performance of Heterogeneous Parallel Computing, especially on GPU. Hence, in this study, a number of optimizations are considered and discussed in this section.

We start with introducing the strategy used for data partitioning. Then, the optimizations related to memory are discussed. Finally, other optimizations such as compiling options and the built-in functions of OpenCL framework are also mentioned.

5.3.1 Optimized Data Partitioning

Data partitioning is a very important aspect of Parallel Computing programs. As demonstrated in our experiment, it dramatically affects the performance and varies from case to case.

In OpenCL, work-items are the basic units for partitioning data. It is a single instance of the kernel (program) on a specific set of data. Tasks are assigned to work-items firstly, and then organized into work-groups. Each work-group executes on a single compute unit of the computing device concurrently.

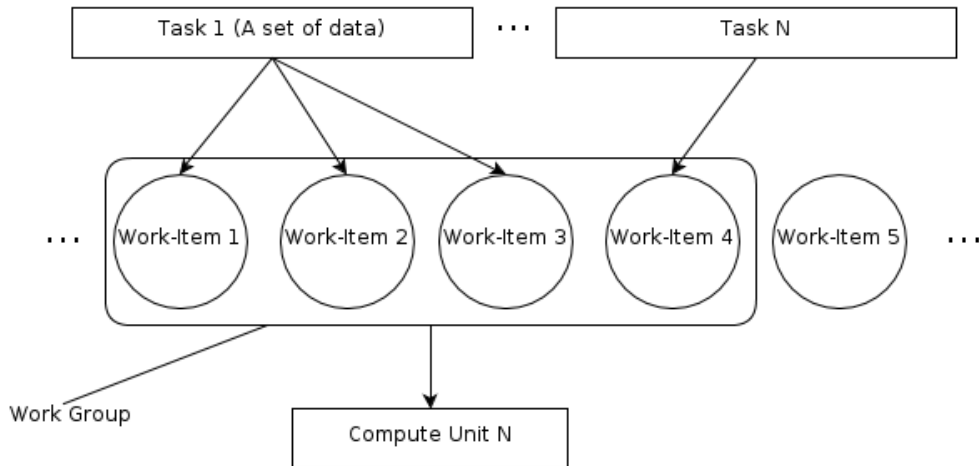


Figure 5.8: A demonstration of the data partitioning mechanism of OpenCL.

To achieve the highest level of performance, in this study a simple experiment was performed to determine the appropriate size of work-group. During the experiment a certain work-item number was specified, and then the same task was executed repeatedly with different work-group sizes.

The time we are measuring in this experiment is the practical time (wall-clock time) including both computation and I/O cost¹.

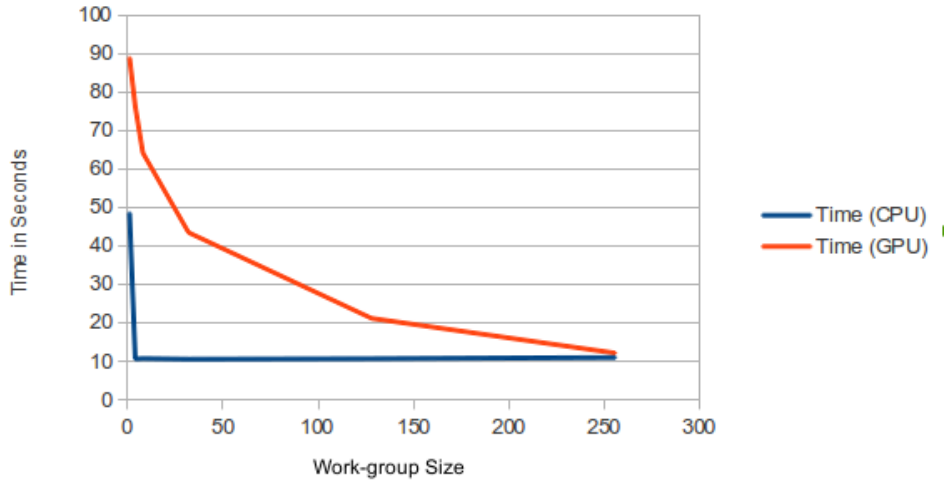


Figure 5.9: Experiment results of the appropriate work-group size.

As shown in figure above, while the work-group number is larger than the computing units (cores) of CPU (8 cores in our case), a constant performance is achieved by CPU. The results suggest that multi-core CPU is not sensitive to the size of work-group, only if every core is fully used.

On the contrary, the size of work-group is much more important to GPU. As can be seen in Figure 5.9, the red line is up and down according to the size of work-group. In order to fully utilize the computational capability of GPU, finally, the maximum size (256 in our case) of work-group is suggested in the experiment.

The experiment results reflect two simple facts: 1) the performance of GPU is limited by the memory latency; 2) one of the best ways to hide the memory latency of GPU is to generate as many as possible active work-items and keep GPU busy by massive calculations.

¹More detail about the time measurement of this experiment can be found in section 6.23 where we discuss the experiment methodology of the study.

5.3.2 GPU Memory Optimizations

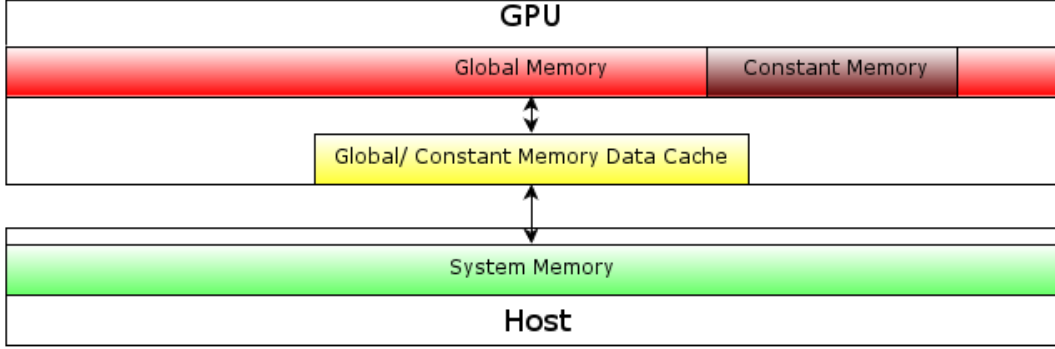


Figure 5.10: The caching subsystem of some GPUs.

As we mentioned in Chapter 3, a multi-level memory structure is defined in OpenCL. In the hierarchy, either the permission or the efficiency of memory access varies from region to region, in particular in the GPU.

To further excavate the potential bandwidth of GPU, ‘Texture Memory’ is used in a number of NIDS studies. It actually refers to a specific memory access method of CUDA rather than a memory region. As indicated by the yellow area of Figure 5.10, a number of texture caches are bonded with Global Memory. While data is formatted in 2D or 1D image type, the caching subsystem may be utilized accordingly. This particular access method is thus referred to as Texture Memory in terms of CUDA.

However, as the experiment results suggest in [8], the caching efficiency of the Texture Memory largely depends on the spatial locality of organized data. It may not always work ideally, and in some cases the performance may even be negatively affected. Furthermore, the cache subsystem is not specified in OpenCL as a part of the standard. In other words it may not be available in many devices.

Therefore, we take advantage of the constant size heuristic table (shift table) of the algorithms used in Clgrep. Instead of using the texture cache of GPU, Local Memory is employed to optimize the I/O operations in Clgrep. Either in GPU mode or CG mode, the heuristic table is updated from Global Memory to Local Memory at the beginning of matching stage. Furthermore, the patterns also are copied to Local Memory, if not too big.

5.3.3 Other Optimizations

Collated Accessing

Apart from data partitioning and memory access optimizations that have been introduced above, collated accessing and JIT (Just-In-Time) compiling options are further investigated in the study.

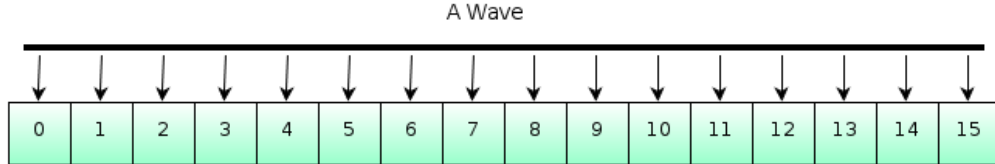


Figure 5.11: A demonstration of collated accessing.

When accessing Global Memory of the GPU, data is read or written in Wave, also known as Warp in CUDA, in which a segment of data is bundled as a whole. Therefore, compared with accessing data located in different segments, collated accessing data in the same segment only requires a few clock cycles. Correspondingly, if accessing memory in a non-collated manner, data accessed within the same Wave has to be partly wasted.

Collated accessing is an important concept of modern GPU, and the implementations of it are different from device to device. At present, 16 memory units are commonly bundled then accessed in a Wave. Further, segment offset is not permitted in collated accessing, in general.

Unfortunately, due to the nature of Horspool and QS algorithms, collated accessing is impractical in Clgrep. The main idea of these algorithms is to avoid unnecessary comparisons during the searching phase. They are attempting to shift the comparison position amongst text non-sequentially. Collated accessing is highly unlikely to occur in such a manner.

The Extensions of OpenCL

During the development of Clgrep we also looked into the optional extensions of OpenCL. There are a number of extensions that may potentially be employed to improve the performance of Clgrep on advanced modern GPUs.

Built-in functions are part of the OpenCL extensions. In some GPUs, they are hardware implemented and thus offer an incomparable efficiency. However, we found that most of them are designed for vector data or float number calculations, and are not appropriate in the string matching algorithms used in this study.

Moreover, we investigate the OpenCL program compilation extension under AMD APP SDK. As listed in the table below, three options are examined by the experiment. Only very limited performance improvement, however, is achieved. Finally, considering the compatibility issues, none are used in Clgrep.

Options
-cl-no-signed-zero
-cl-unsafe-math-optimizations
-cl-fast-finite-math-only

Table 5.2: The compilation extensions of OpenCL that may potentially improve execution efficiency.

5.4 The Implementation of Clgrep

Clgrep is implemented in the C programming language, and consists of a number of functions, including `opt_check()`, `qs_match()`, `bmh_match()`, `set_qs_match()`, `set_bmh_match()`, and `res_report()`.

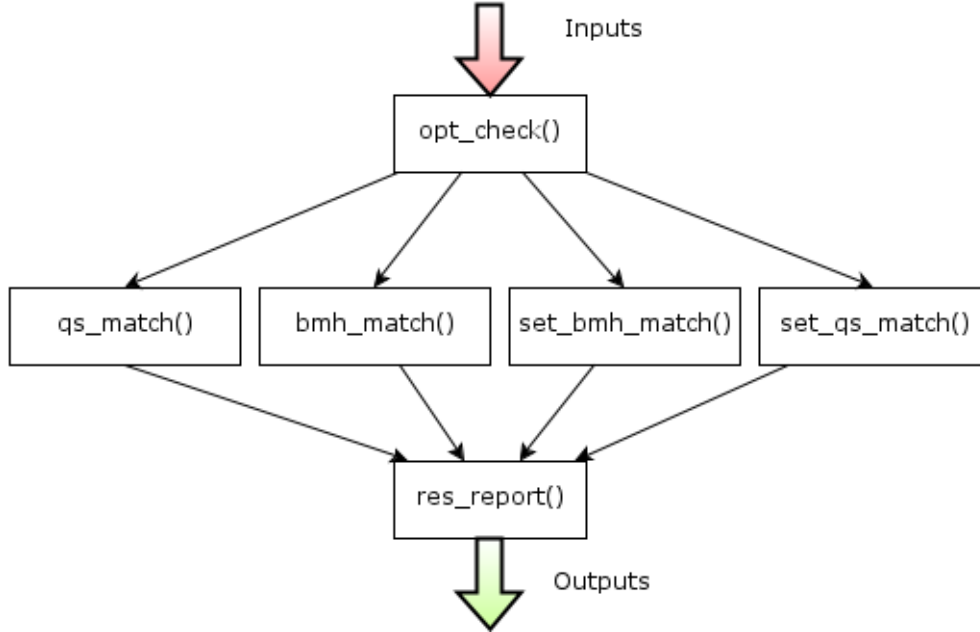


Figure 5.12: The architecture of Clgrep is based on functions.

In Clgrep, the `opt_check()` function is implemented for examining the inputs from users. It identifies the options switched by users, and reports errors when unexpected parameters are entered. For example, when GPU mode is selected but no OpenCL-capable GPU is available, `opt_check()` will then raise an error.

Because the functionality of Clgrep is simple, there are only few branches in `opt_check()` function. The options currently available in Clgrep are listed in Table 5.4.

The single pattern matching feature of Clgrep is provided by `qs_match()` and `bmh_match()` functions. QS and Horspool algorithms are implemented in `qs_match()` and `bmh_match()` respectively. Both of them consist of two parts: one is the host program, and another is the kernel code that separated from the C source code and only being compiled at run time.

Besides, `set_qs_match()` and `set_bmh_match()` functions are implemented for multi-pattern matching. Similar to `qs_match()` and `bmh_match()`, they also include two parts

‘-q’ for switching to Quick Search algorithm (default)
‘-h’ for switching to Horspool algorithm
‘-m’ for indicating multi-pattern matching (default)
‘-s’ for indicating single-pattern matching
‘-i’ for selecting multi-core CPU processing mode (default)
‘-g’ for selecting GPU processing mode
‘-cg’ for selecting CG processing mode

Table 5.3: The options of Clgrep.

– the host program and kernel code. But, as we addressed previously, their matching processes are slightly different from single pattern matching functions.

The codes, used in pre-processing stage, are located in the host program. It includes the pre-computation function of algorithm and the initiation functions of OpenCL framework. When a single pattern is matched, the shift table is structured in a sequential manner on CPU according to algorithm being selected. Moreover, the computing devices are also prepared.

```

CHECK algorithm being used AND encoding of the matching text:
THEN structure the shift table;

Initialize OpenCL framework;
CHECK processing mode being selected;
IF selected computing device is available:
    submit kernel instances and move to the matching stage;
ELSE
    report error to opt_check();

```

Listing 5.1: The preprocessing stage of Clgrep is implemented as a part of the host program.

On the other hand, the programs of the matching stage reside in the kernel code. They are more important but difficult to implement. There are a number of limitations in OpenCL, and everything must be controlled at such a low level, when programming the kernel code. Furthermore, whenever an issue is raised in kernel code, no debugging information can be reported, and the system display may be frozen in many cases.

```

CHECK the ID of work-item;
THEN partitioning data;

FOR each work-item:
    WHILE(a certain range):
        search patterns;
        IF a match is found:
            save result in the Local Memory;
        save result in the Global Memory of computing device;

RETURN all the results to the System Memory;

```

Listing 5.2: The matching stage of Clgrep is implemented as kernel code in OpenCL.

In Clgrep, `res_report()` is used for printing results via the standard output. It is merely a handful of lines code. Only the post-processing stage is implemented in the `res_report()` function not the result reporting hierarchy used in the matching stage.

5.5 Summary

In this chapter the designs and implementation of Clgrep were presented in detail. The algorithms of Clgrep were introduced, and their strengths and weaknesses discussed. Then, the mechanism and matching procedure of Clgrep were described, and the Parallel Computing model and extra processes in such an approach were emphasized. A number of optimizations were discussed, the experiment results of optimizations were presented, and several concepts rarely mentioned in other studies were introduced. Finally, the function-oriented architecture of Clgrep was described.

In short, Clgrep as a string matching command utility is very different from any other alternatives. It is the only one based on OpenCL, features different processing modes, and behaves in a SIMD Parallel Computing manner.

In addition, as an application specially developed for the study, the features of Clgrep are designed with our research questions in mind. As a result, Clgrep is fully capable of supporting our following research.

Chapter 6

Experiments

In the experimental stage of our study, Clgrep is examined through a variety of matching cases. The performance of each algorithm and processing mode is carefully measured, profiled, and further compared with the sequential implementations correspondingly. The use of QS and Horspool algorithms are investigated in the context of Heterogeneous Parallel Computing.

This chapter introduces the methodology designed for the experiment, and presents the experiment results with discussions. The rest of the chapter is organized as follows:

1. First of all, the aims of the experiment are described.
2. Secondly, the methodology of experiment is introduced in detail, the matching cases used in the experiment are described, then the experiment procedure and time measurement are presented. In addition, the experiment setup is introduced.
3. Thirdly, the results of the experiment are visualized and discussed, and important observations of the experiment are explained.
4. Finally, this chapter ends with a brief summary.

6.1 The Aims of Experiment

There are two aims of the experiment:

1. We aim to validate the new approach proposed in the study – Clgrep, a grep-like Heterogeneous Parallel Computing exact matching utility, and investigate the strengths and weakness of such an approach in practice.
2. We aim to profile the performance of string matching algorithms in different processing modes and matching cases, and further investigate the use of heuristics-based algorithm, specifically QS and Horspool algorithms in the context of Heterogeneous Parallel Computing,

6.2 Experiment Methodology

According to the aims of the experiment and a number of previous studies [32, 33, 34, 53, 54, 55, 54], the experiment methodology of the study is designed.

In the experiment, both single pattern matching and multi-pattern matching are considered. A variety of matching cases is designed to examine the matching performance of each algorithm and processing mode. In addition, several Bash scripts were also developed to control and automate the procedure of the experiments. Bash is a popular Unix-shell scripting language. The scripts were also used to record the results of the experimental runs.

6.2.1 The Matching Cases

Matching Type	Text Size	Length of Patterns	Number of Patterns
English	1, 10, 100 MB	Ranged from 1 to 10	10, 20
DNA	100 MB	Ranged from 1 to 20	10, 20

Table 6.1: The matching cases of the experiment.

As listed in the table above, the matching cases are composed of a number of combinations including different types of text, different sizes of text, different lengths of patterns, and different numbers of patterns that are concurrently searched in multi-pattern matching. In addition to the processing modes and algorithms we are focused on in the study, there are overall 100 combinations in the experiment. To the best of our knowledge it may be one of the most comprehensive experiments on this specific topic.

Matching Text and Patterns

The matching texts and patterns used in the experiment are randomly selected then converted from the Gutenberg Project, a project that offers over 40,000 book and documents in plain-text or other formats for free.

An interesting Bash shell script is developed by us for filtering and converting the text data downloaded from the FTP server of the Gutenberg Project.

The matching texts are all saved as plain-text and encoded in ASCII, including three English texts (size at 1 MB, 10 MB, 100 MB respectively) and one complete genome segment (the number 13 DNA human genome about 100 MB large in size). In addition, the patterns used in the experiment are also specially selected. They are either meaningful vocabularies or real genome sub-strings.

6.2.2 The Sequential Implementation

The sequential string matching program used in the experiment is carefully designed. It employ the same algorithms as Clgrep without parallelism. It can be viewed as a lightweight grep though they are not identical.

- They employ different variants of BM algorithm when doing exact pattern matching.
- The matching process of grep is based on the basis of line but the sequential program of this study is designed for search every single matched character. In other word, the sequential implementation used in the study does not skip searching the entire line after matched a single pattern.

Besides, the sequential implementation is fully optimized by gcc compiler just like most grep distribution available on Unix-like OS. However, unlike grep it does not include any control options or other functionalities.

6.2.3 Experiment Procedure

Experiments on string matching are cumbersome and time-consuming [32].

In this study, a single round of the experiment may take hours on a fairly powerful computer. Therefore, in order to accurately control the execution of the experiment, and precisely record the results of every single case, a number of Bash shell scripts¹ are developed to perform the experiment procedure automatically.

A demonstration of the experiment procedure is shown in Figure 6.1 in which:

1. Firstly, by using native functions of OpenCL framework, experiment settings are profiled: the important details of computing devices and platform used in the experiment are recorded.
2. Then, a variety of matching scenarios consisting of many cases are applied for evaluating the matching performance: in every single matching case, for each algorithm and processing mode the execution time is precisely measured, tagged, and saved in files.

¹All the scripts developed can be found at <https://github.com/xuwupeng2000/CLgrep>

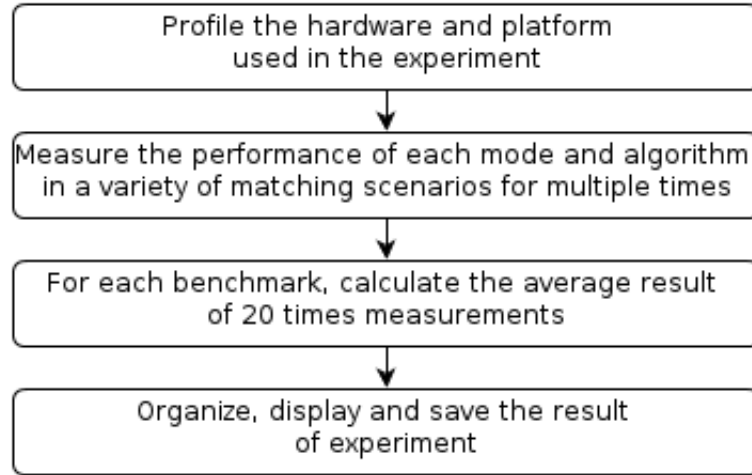


Figure 6.1: Experiment procedure.

3. Then, the same matching tasks are executed 20 times by program and the average results are calculated.
4. Finally, the experiment results are organized, displayed, and saved in files as reports.²

```

Wed Aug  8 09:51:01 NZST 2012
=&= Horspool Algorithm
=&= Needles Length = 1-20
=&= 100 MB DNA Haystack
*****
Sequential Implementation  Multi-core CPU mode  Multi-core GPU mode  CG mode
1:17.05 real time          1:24.55 real time    1:37.35 real time    1:45.26 real time
*****

```

Figure 6.2: The raw data collected from a single experiment.

²All the raw data we collected in the experiment can be found at <https://github.com/xuwupeng2000/CLgrep>

6.2.4 Time Measurement

Unlike most NIDS studies we do not disregard the costs of preprocessing stage and post-processing stage in the experiment. To be more objective, time is counted from when the commands of Clgrep are entered until the results are sent back and displayed to users, including not only computation costs but also I/O costs. Besides, to further control the experimental errors the final values reported are the average of 20 measurements.



Figure 6.3: The entire process is carefully measured in the experiment.

The Costs of The Non-Matching Stages

Despite that we focus on the “real-world” performance rather than theoretical analysis in the study. The costs of each stage are summarized and profiled by experiment. However, the results need to be taken with care. The exact values may be affected by a number of factors and thus may only be applied in the experiment of this study³.

Stage	Cost (Time in Seconds)
Preprocessing	About 0.4s in GPU and CPU mode but 0.5s in CG mode
I/O	100 MB data costs about 0.4s in GPU and CG modes but only 0.1s in CPU mode
Post-processing	Less than 0.02s in any mode

Table 6.2: The non-matching costs of every single search.

³More detail can be found in the appendix of the thesis.

Preprocessing Costs

The costs of preprocessing stage consist of these:

OpenCL framework initialization

Although the initialization cost of OpenCL framework is a constant in the experiment it is depending on the hardware capability, the implementation of OpenCL framework and the graphic card driver being used. Noticeably, the initialization cost of CG mode is slightly more expensive than other modes as more than one hardware pipeline needs to be set up.

Shift table precomputation

The cost of constructing the shift table depends on the algorithm used and the number of search patterns. It is insignificant and can be effectively done on the host.

I/O Costs

The I/O cost of transferring data between System Memory and GPU Global Memory is a critical part of the entire process. It highly depends on the size of data and the capability of System I/O bus and hardly be controlled by software program in most cases.

Post-processing Costs

Finally, the cost of post-processing stage includes 1) Reading then display the matching results, 2) OpenCL framework finalization. It is a not a major concern as only few milliseconds are spent at this stage.

Measurement Results Analysis

Once the search patterns and text are specified, the quantity of calculation is constant. For the same experiment, the measurement result of each run is extremely close. In consequence, the standard deviation is small. Assuming that the difference between each run should be less than 1% of the average result, the confidence level of the results is over 95%.

6.2.5 Experiment Setup

When benchmarking the performance of string matching on heterogeneous computer systems, the results may vary with a number of experiments settings. Therefore, to ensure the repeatability of the study, the details of the experiment set-ups are described in the following.

As listed in Table 6.3, Ubuntu 12.04 GNU/Linux Operating System was used in the experiment. The Linux Kernel version of the system is 3.2. In addition, AMD Catalyst proprietary display driver (version 12.7) was deployed with AMD APP SDK (version 2.6). As an implementation of OpenCL, AMD APP SDK (2.6) supports the OpenCL 1.1 standard.

During the experiment only the typical background processes ran, and the networking was disconnected. The entire experiment procedures were controlled by program, and there are no user interruptions at all.

Operating System	Ubuntu 12.04, Kernel version is 3.2
OpenCL Implementation	AMD APP SDK 2.6 supports OpenCL 1.1
Graphic Card Driver	AMD Catalyst Driver version 12.7

Table 6.3: The experiment settings and software versions.

Apart from the software settings, an Intel Core i7 CPU and a Radeon HD 6990 GPU (PCI Express 2.1) were used in our experiment, both of them fairly high-end. Besides, a DDR3 (double data rate synchronous dynamic random-access memory) 1600 MHz 4 GB memory chip, and a 7200 RPM (revolutions per minute) 500 GB hard disk from Western Digital were employed. The hard disk was connected with SATA 3 to the motherboard of the computer.

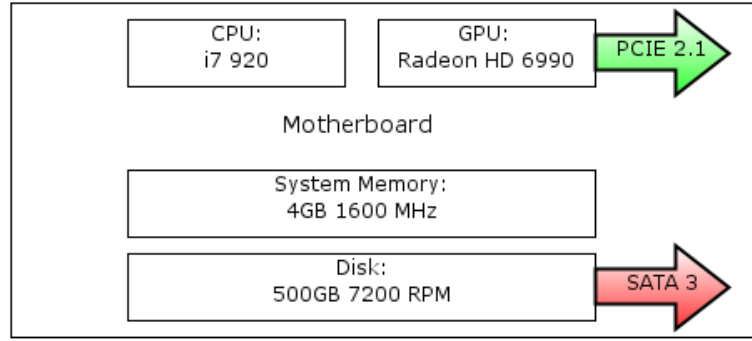


Figure 6.4: An overview of the hardware experiment settings.

Considering that the specific capabilities of the hardware are very critical to performance, the details of CPU and GPU are presented in the table below. The ‘CL-LOCAL’ and ‘CL-GLOBAL’ types are used by OpenCL framework to indicate very fast memory and normal memory, respectively.

Model	Radeon HD 6990 GPU	Intel Core i7 920 CPU
Vendor	AMD	Intel
Computing Units	24	8
Clock Frequency (MHz)	830	2670
Global Memory Size (MB)	1024	3200
Max Allocatable Memory (MB)	256	1024
Local Memory Type	CL-LOCAL	CL-GLOBAL
Local Memory Size (KB)	32	32
Global Memory Cache Type	N/A	Read and Write Cache
Global Memory Cache Size (K)	N/A	32
Max Group Size	256	1024

Table 6.4: The hardware settings of experiment.

6.3 Results and Discussion

6.3.1 Single Pattern Matching

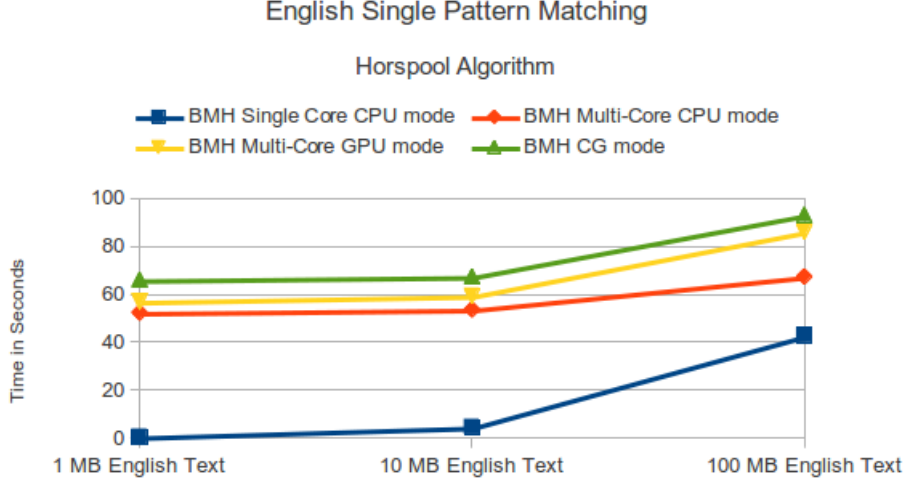


Figure 6.5: The results of Horspool algorithm.

In this section the experiment results are presented with discussions in which the strengths and weaknesses of Clgrep are shown. More importantly, the use of Horspool and Quick Search algorithms in the context of Heterogeneous Parallel Computing is explored.

In Figure 6.5 the performances of Horspool algorithm in English single pattern matching are demonstrated. Apparently, single core CPU mode is much more efficient than others, especially when the size of text is small.

In the experiment, matching a single pattern for 100 times in 1 MB English text merely took about 2 seconds when processing by the sequential implementation of Horspool (single core CPU mode). However, for exactly the same task, it took about 50 seconds to 60 seconds in the rest of the modes that were implemented with OpenCL framework and responding in SIMD Parallel Computing manner.

When the size of text is increasing from 1 MB to 10 MB and further to 100 MB, despite the gaps between the sequential implementation and other modes being clearly shortened, the results still keep the same ranks. As presented in the table below, for 100 MB text, the performance of CG mode still is more than 2 times less efficient than processing on a single core of CPU.

Mode	Time (Seconds)
BMH Single Core CPU Mode	42.84
BMH CG Mode	92.25

Table 6.5: English single pattern matching: The sequential implementation vs. BMH CG mode.

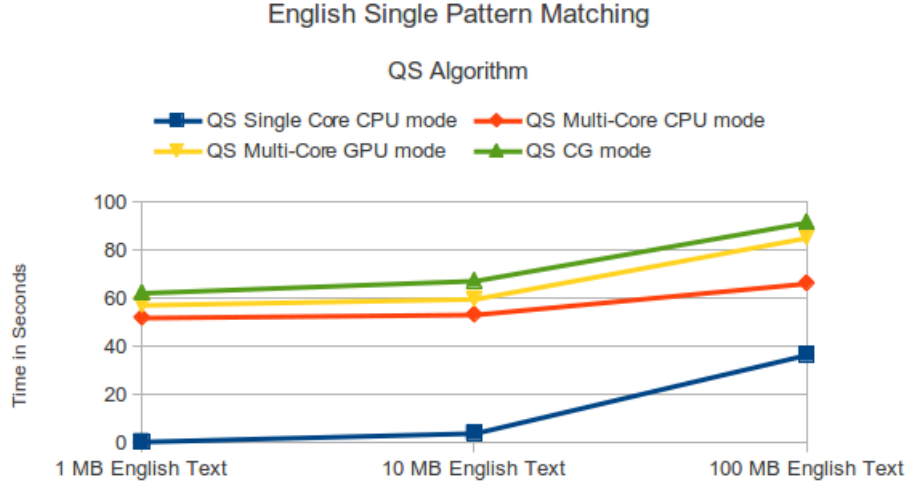


Figure 6.6: The results of QS algorithm.

As demonstrated in Figure 6.6, a very similar result was achieved while applying QS algorithm for English single pattern matching: among these modes, processing on a single core of CPU without OpenCL framework is far more efficient.

The results of the experiment suggest that compared to the sequential implementations, Clgrep is not competitive in single English text matching. The initialization latency of OpenCL framework exceeds the gain that can be obtained, in most cases, especially in CG mode where both CPU and GPU are managed and worked concurrently.

Besides, the expensive I/O cost of GPU mode also negatively affects the results, causing the GPU mode and CG mode to be particularly inefficient. While matchings are processed by the GPU, instead of directly mapping from host memory region, OpenCL objects are actually read and written from host to the Global Memory of GPU. This operation consists of a number of instructions, is limited by the bandwidth of system I/O, and is very costly.

In addition, regardless of processing modes, the performance of QS algorithm is slightly better than Horspool algorithm in the experiment. As indicated in Table 6.6, the best

result (0.45 seconds) was achieved by the sequential implementation of QS algorithm, despite the difference between QS and Horspool being not significant.

Mode	Time (Seconds)
BMH Single Core CPU Mode	0.52
BMH Multi-core CPU Mode	52.42
BMH GPU Mode	57.02
BMH CG Mode	66.01
QS Single Core CPU Mode	0.45
QS Multi-core CPU Mode	52.14
QS GPU Mode	56.66
QS CG Mode	62.51

Table 6.6: English single pattern matching: BMH vs. QS.

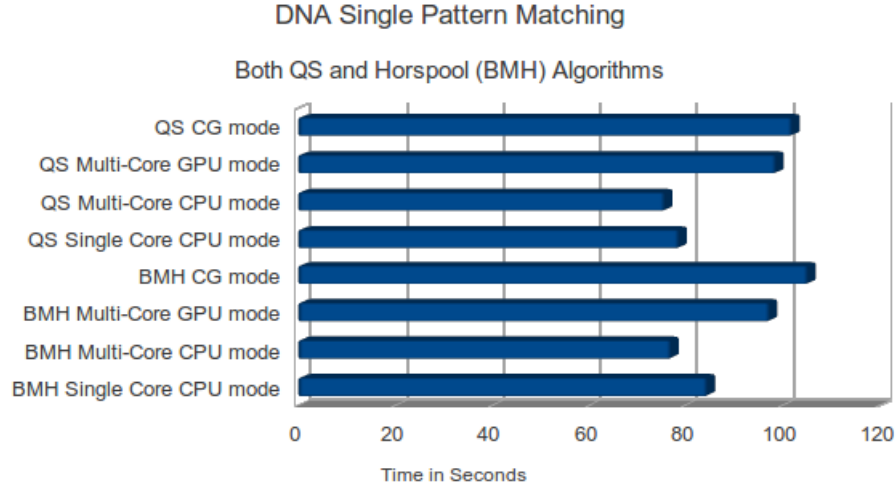


Figure 6.7: The result of DNA single pattern matching.

DNA sequence matching is different from English text matching. It consists of only four bases (A, C, T, G). Comparing to natural language such as English, in DNA sequence, characters are more likely to be repeated. In consequence, the potential shift distances of Horspool and QS algorithms are both decreased. In other words, more comparisons are required to determine an exact match in the DNA sequence rather than English text. As the result, DNA sequence matching is more computationally intensive and time-consuming than English text matching in general.

In Figure 6.7, the matching performances of QS and BMH algorithms are both presented. The results suggest that the efficiency of multi-core CPU mode is better than the other processing modes in DNA sequence single pattern matching. Because of the strong performance of multi-core CPU mode, Clgrep firstly exceeds the traditional sequential implementation in the single pattern matching experiment.

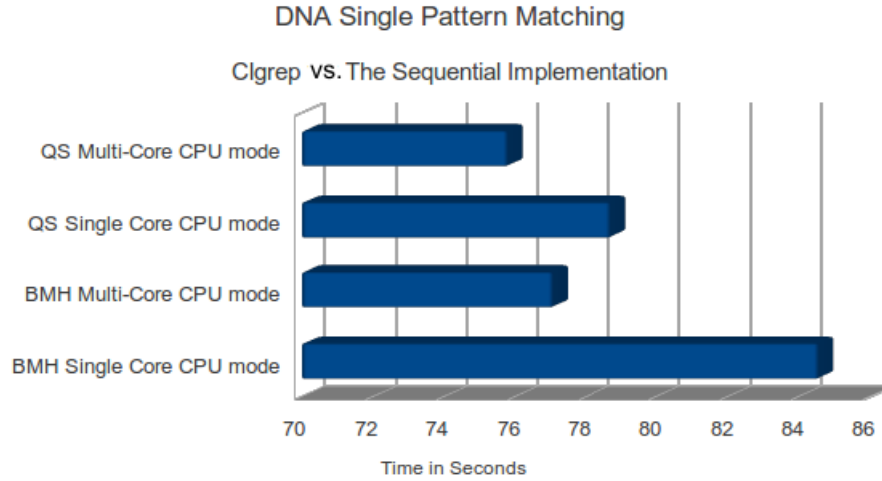


Figure 6.8: A more precise comparison between the multi-core mode of Clgrep and the sequential implementation.

A more precise comparison between the multi-core mode of Clgrep and the sequential implementation is presented above, in which the best result of 75.78 seconds is achieved by Clgrep in multi-core CPU mode.

Besides, as can be seen in Figure 6.8, QS algorithm, again, achieves a better performance on average compared with Horspool algorithm (BMH).

In contrast to English text matching, there is an obvious improvement of Clgrep in DNA sequence matching. Unlike processing on a single core of CPU, the multi-core processing modes of Clgrep are not very sensitive to the increasing computational requirement. They demonstrated stable behaviour during the experiment. A sophisticated trade-off between I/O costs and computational capability was found in the multi-core CPU mode of Clgrep, and makes it particularly efficient in DNA sequence single pattern matching.

6.3.2 Multi-pattern Matching

In multi-pattern matching, a set of patterns is searched simultaneously. Compared with single pattern matching, it is much more costly and time-consuming. The computational quantity and intensity of it not only depend on the type of text and the length of pattern, but also the number of patterns that can be searched concurrently in matching.

In the experiment, two types of text were searched, both English text and DNA sequence. In addition, different numbers of patterns concurrently searched were also examined. Two groups of patterns were employed, sized at 10 and 20 respectively.

The results of the multi-pattern matching experiment are presented in this subsection.

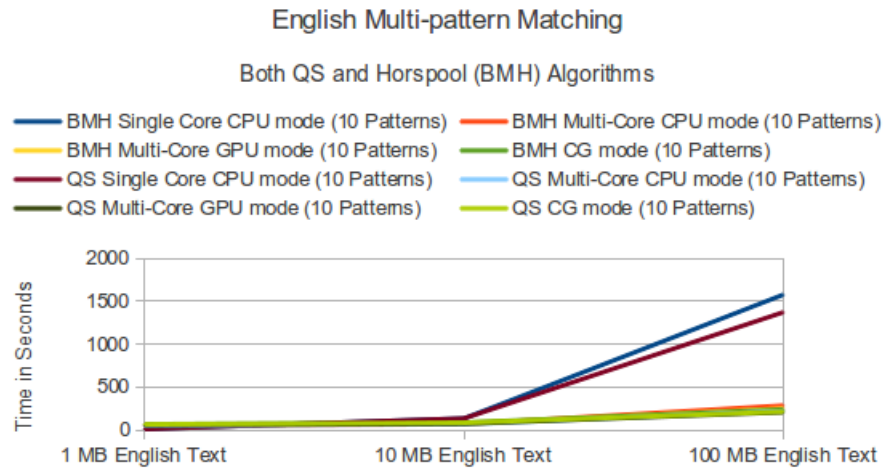


Figure 6.9: The results of matching 10 patterns concurrently in English text.

In Figure 6.9, the overall performance of matching 10 patterns in English text is demonstrated. Although the performance of each mode inside Clgrep is close, the difference between Clgrep and the sequential implementations is obvious. Contrary to the sequential implementations of each algorithm, Clgrep presented a very strong performance during the experiment, especially when the size of text was relatively large.

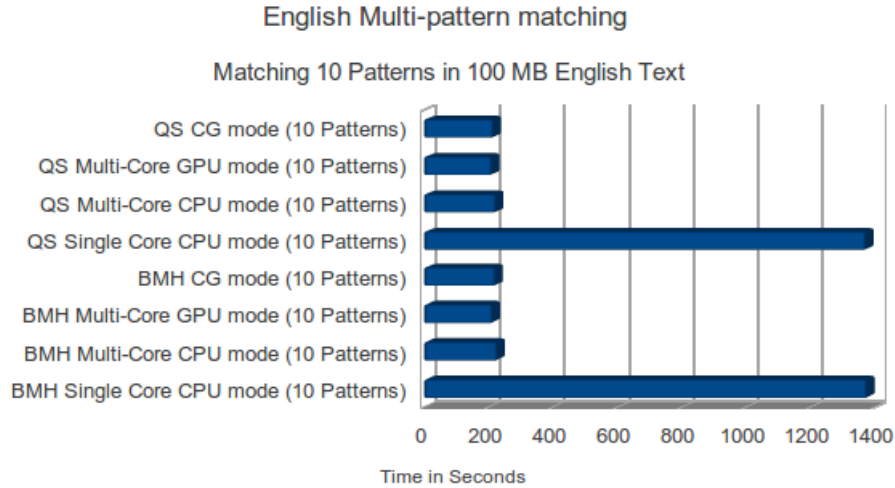


Figure 6.10: Matching 10 patterns in 100 MB English text.

As presented in Figure 6.10, when multiple patterns are concurrently searched in 100 MB English text, the performances of the sequential implementation are apparently limited. During the experiment, locating 10 patterns for 100 times took the sequential implementation about 23 minutes, but merely 3 minutes in GPU mode of Clgrep. Arguably, the performance gaps are beyond the differences in algorithm.

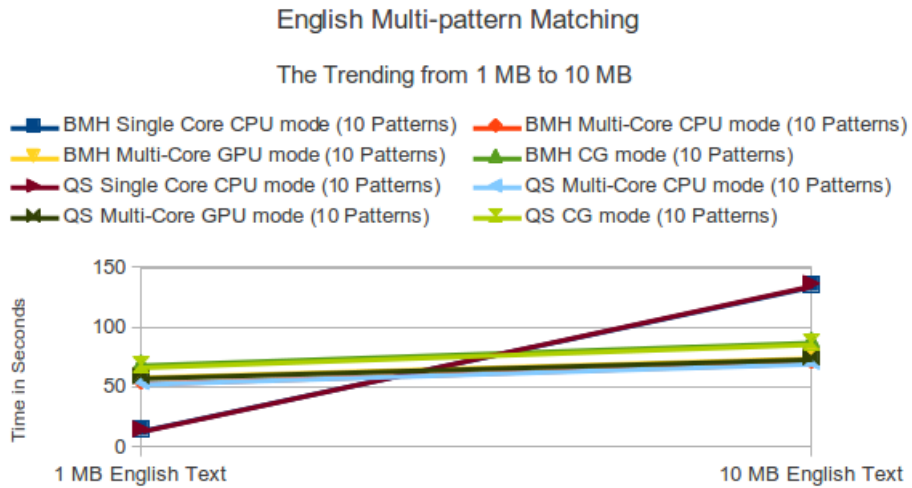


Figure 6.11: The performance trend from 1 MB to 10 MB.

Like single pattern matching, Clgrep does not perform well in terms of small size of text, though the difference is less significant when the patterns number is slowly increasing. As can be observed in Figure 6.11, in this specific case (matching 10 patterns concurrently), the margin of text size for Clgrep to overtake the sequential implementation is about 5 MB or 6 MB.

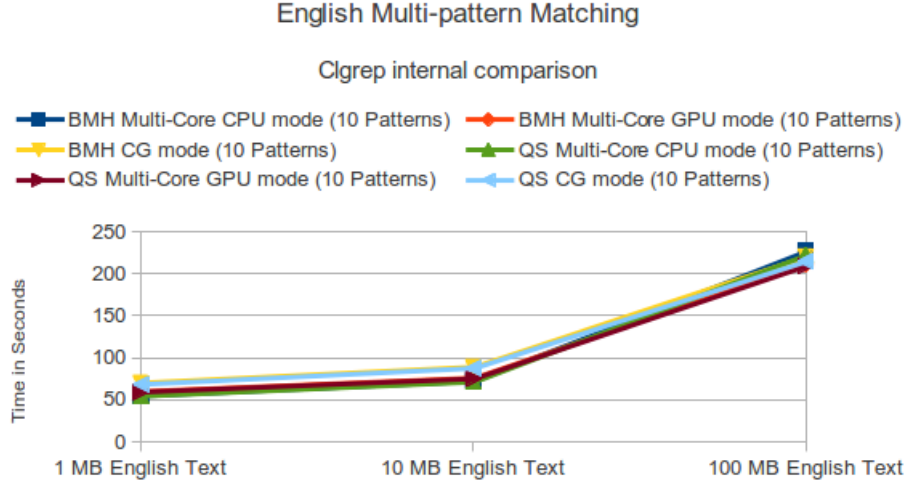


Figure 6.12: The internal comparison of Clgrep when matching 10 patterns concurrently in English text.

Apart from the comparison between Clgrep and the sequential implementation of corresponding algorithm, let us look at the differences between each mode of Clgrep. As presented in Figure 6.12, within Clgrep, the performance of each mode is very close in English text matching. In the case of matching 10 patterns concurrently, only the green curve can be identified as under others at the point of 10 MB, and indicates where the performance of QS algorithm in multi-core CPU mode is slightly better.

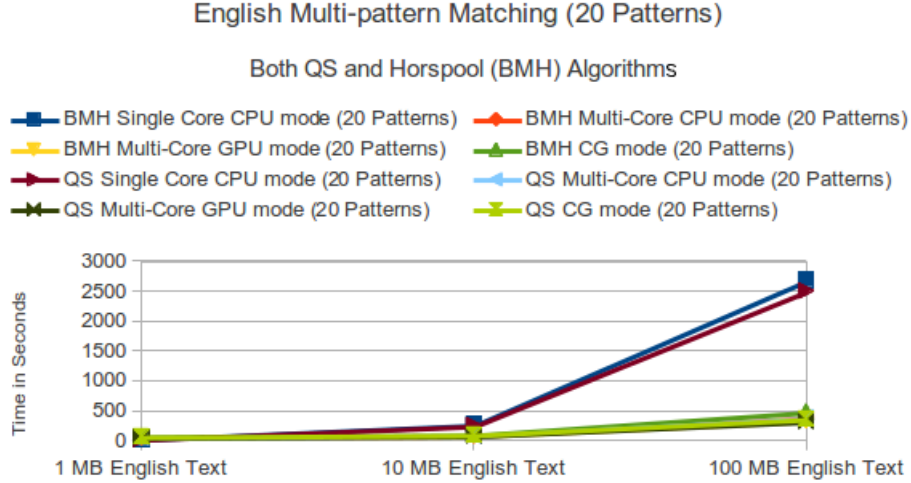


Figure 6.13: The results of English multi-pattern matching (20 patterns).

When matching 20 patterns concurrently in English text, the quantity and intensity of computation is further increased. In consequence, the gaps between the sequential implementation (Single Core CPU mode) and Clgrep become larger as well.

As can be seen in the table below, while matching 20 patterns concurrently in 100 MB English text, the GPU mode of Clgrep is about 5 times faster than processing on a single core of CPU, regardless of the algorithms applied.

Mode	Time (Approximate Minutes)
BMH Single Core CPU Mode	44.8
BMH Multi-core CPU Mode	6.4
BMH GPU Mode	5.6
BMH CG Mode	8.0
QS Single Core CPU Mode	41.8
QS Multi-core CPU Mode	6.3
QS GPU Mode	5.4
QS CG Mode	6.0

Table 6.7: The results of matching 20 patterns concurrently in 100 MB English text.

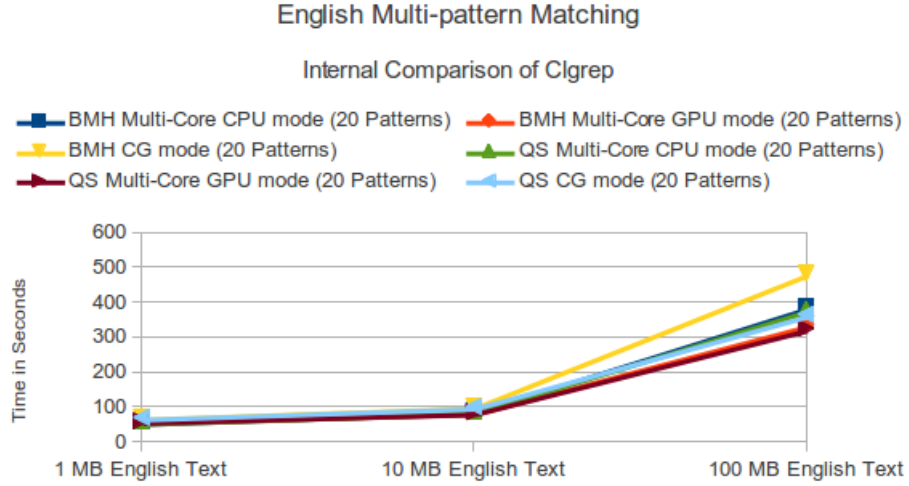


Figure 6.14: The differences between each mode of Clgrep when matching 20 patterns concurrently in English text.

While searching for 20 patterns concurrently, the internal differences of Clgrep also become more obvious.

As can be seen in Figure 6.14, the GPU modes of Clgrep are more efficient than others, regardless of the algorithms used. In addition, due to the hardware management costs of OpenCL framework, CG mode is still clearly behind others.

Apart from the performance differences between each processing mode, Figure 6.14 also demonstrates that QS algorithm is more efficient than Horspool (BMH) algorithm overall.

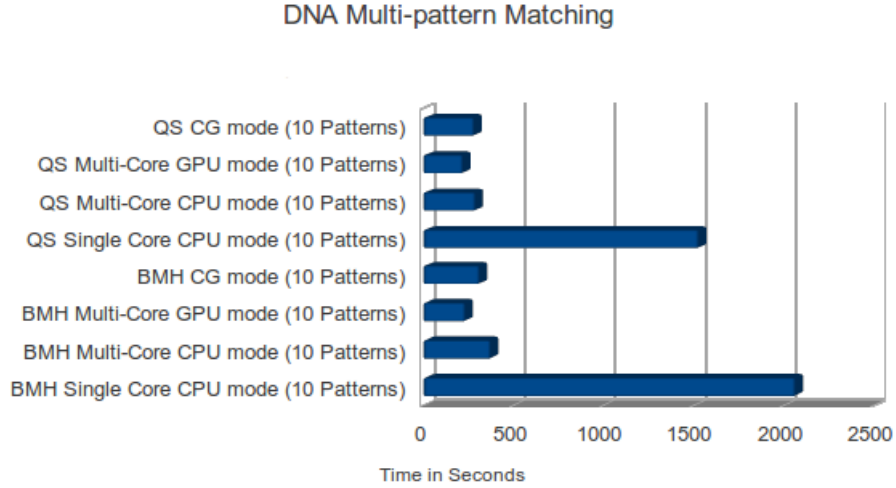


Figure 6.15: Matching 10 patterns concurrently in DNA sequence.

Having discussed the multi-pattern matching of English text, let us look at the experiment results of DNA sequence matching. Certainly, it is the most time consuming task in the experiment of the study.

In Figure 6.15, from the results of searching 10 patterns in DNA sequence for 100 times, the computational limitation of processing data on a single core of CPU is clearly explored. It took over 30 minutes for the sequential implementations of each algorithm to address every match. On the contrary, the same task took only 5 minutes in the GPU mode of Clgrep.

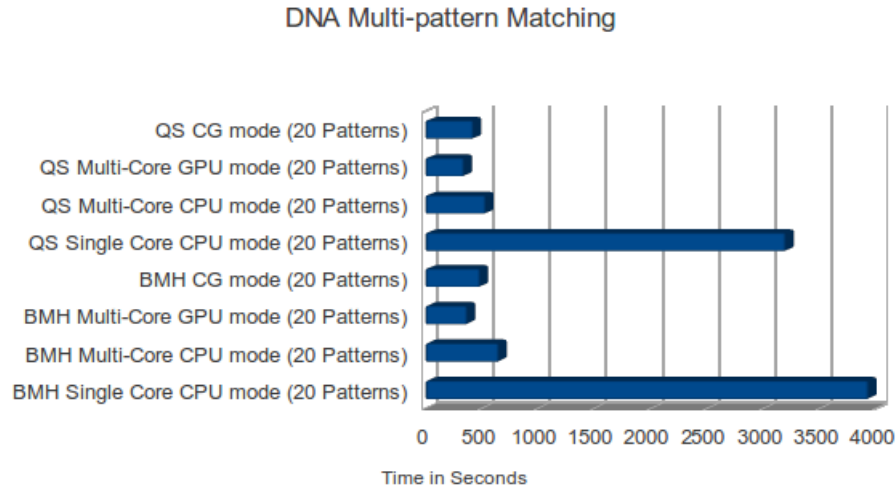


Figure 6.16: Matching 20 patterns concurrently in DNA sequence.

As presented in Figure 6.16 a similar result is achieved in the case of searching 20 patterns concurrently. The computational capability of CPU is overwhelmed by the onerous comparisons of matching, and thus the efficiency of processing on a single core of CPU becomes unacceptable. It spends almost one hour finding all the matches during the experiment. Contrary to that, because of the massive computational capability and parallel nature of GPU, the same task, takes less than 10 minutes in the GPU mode of Clgrep.

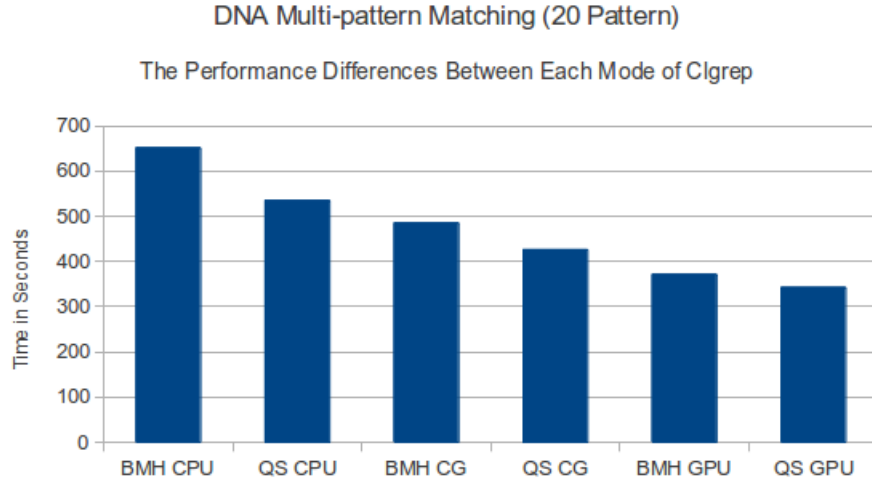


Figure 6.17: The performance differences between each mode of Clgrep when matching 20 DNA patterns concurrently.

In terms of DNA sequence matching, particularly when searching for 20 patterns concurrently, the performance gap between each mode of Clgrep becomes clearer.

As can be seen in Figure 6.17, instead of processing in the traditional sequential manner and benefiting from avoiding the extra costs of OpenCL framework, Parallel Computing processing mode with CPU and GPU has become a much better choice by being efficient.

In the case of searching 20 patterns concurrently, GPU presents a very strong performance. It is at least 2 times faster compared with multi-core CPU mode during the experiment. Besides, CG mode benefits from the computational capability of GPU as well and takes second place, followed by the multi-core CPU mode.

Regarding the processing mode, the experiment results also suggest that QS algorithm is more efficient than Horspool. The performance gap between these two algorithms is more obvious in DNA sequence matching rather than English text matching. Clearly, the advantages of QS algorithm start to present while shorter patterns are searched for.

6.3.3 The Scalability of Clgrep

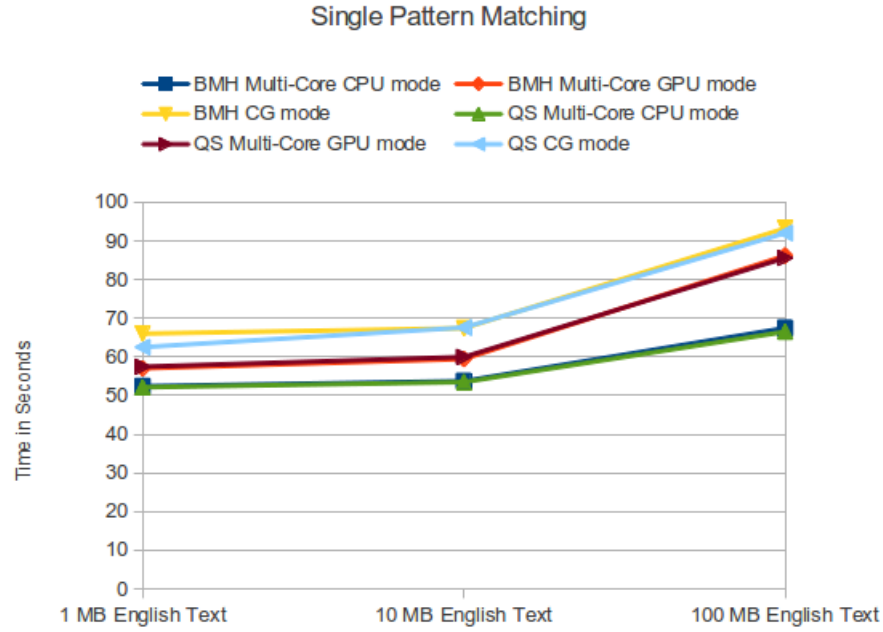


Figure 6.18: English single pattern matching. The comparison between each mode of Clgrep.

All the experiment results of each case having been presented. The scalability of Clgrep is discussed in this subsection.

As can be seen in Figure 6.18, while the size of text is increasing, all of the lines are becoming tilted, in which the similar scalability of each mode inside Clgrep is indicated by the close gradients.

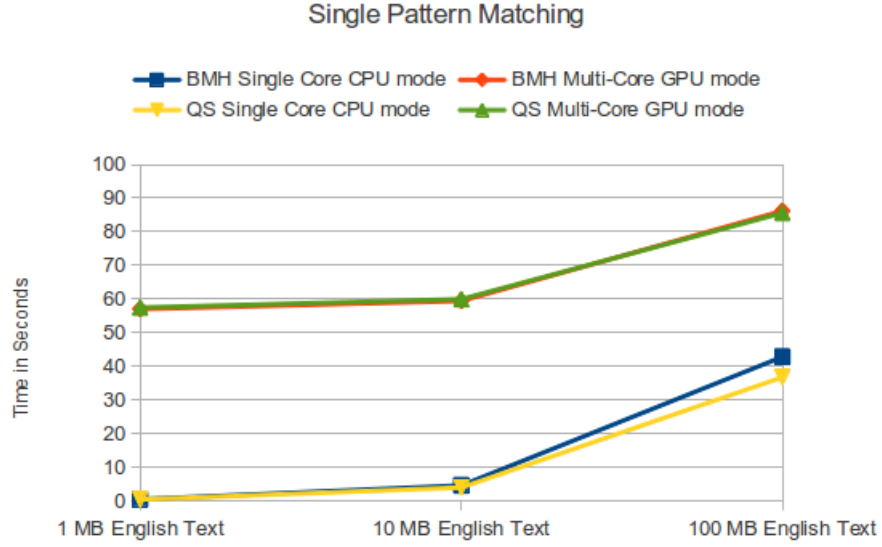


Figure 6.19: English single pattern matching. Clgrep vs. The sequential implementation.

The scalability comparison between the sequential implementation is shown in Figure 6.19. In the case of single pattern matching, although the scalability of Clgrep (represented by the GPU mode of Clgrep) is slightly better than the traditional sequential implementation, the difference between them is not obvious. There is no clear evidence to suggest that the Parallel Computing model of Clgrep is going to be significantly more efficient than the sequential implementation, if the text size is further increasing in the experiment. The advantages of Clgrep are limited by these factors⁴ listed below:

- The initialization cost of OpenCL framework.
- The extra cost of employing the Parallel Computing model and managing multiple hardware.
- The expensive I/O operations of GPU, more importantly.

⁴More detail can be found in the appendix of the thesis.

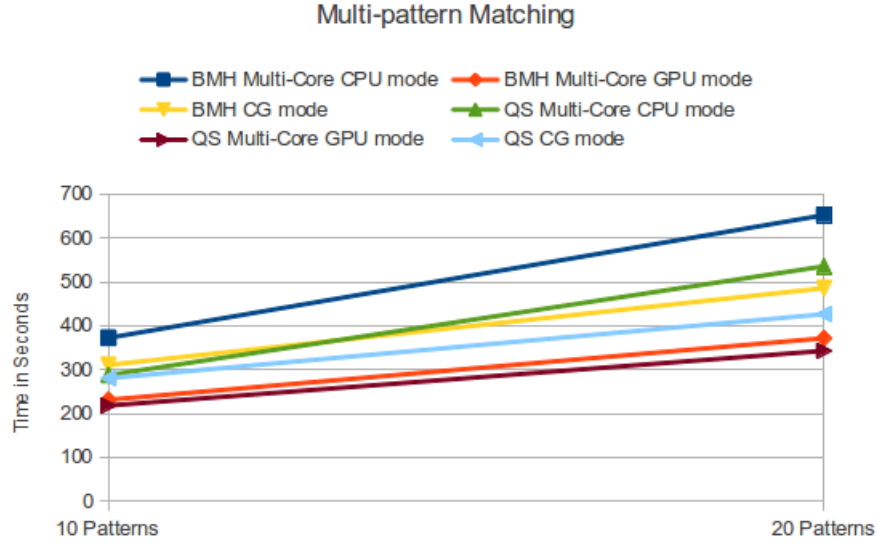


Figure 6.20: DNA multi-pattern matching. The internal comparison of Clgrep.

The scalability of each mode inside Clgrep is demonstrated in Figure 6.20. Again, they are very close. Although the scalability of GPU mode is slightly better it is not significant. Besides, the differences between algorithms are not obvious.

Nevertheless, the situation is changed when comparing any mode of Clgrep with the sequential implementation. In terms of multi-pattern matching, especially DNA sequence multi-pattern matching, an outstanding performance is presented by Clgrep. The scalability of it is definitely better than the sequential implementation, in most cases.

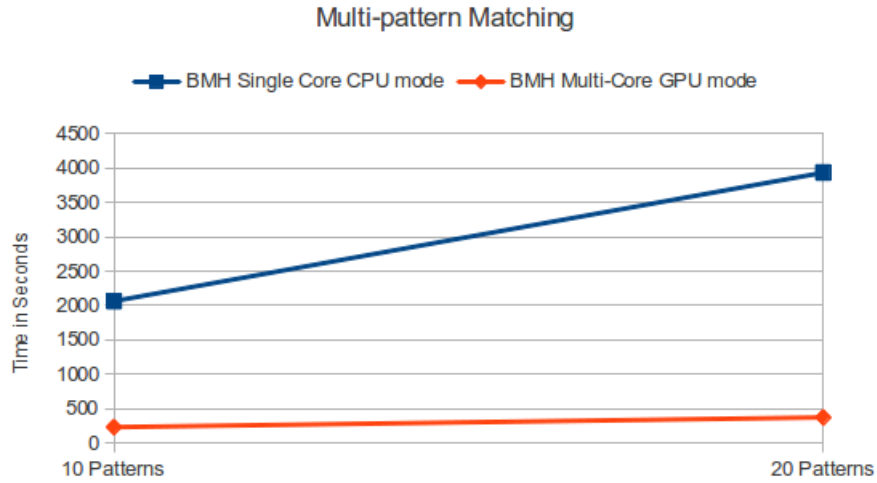


Figure 6.21: DNA multi-pattern matching. Clgrep (BMH algorithm) vs. The sequential implementation.

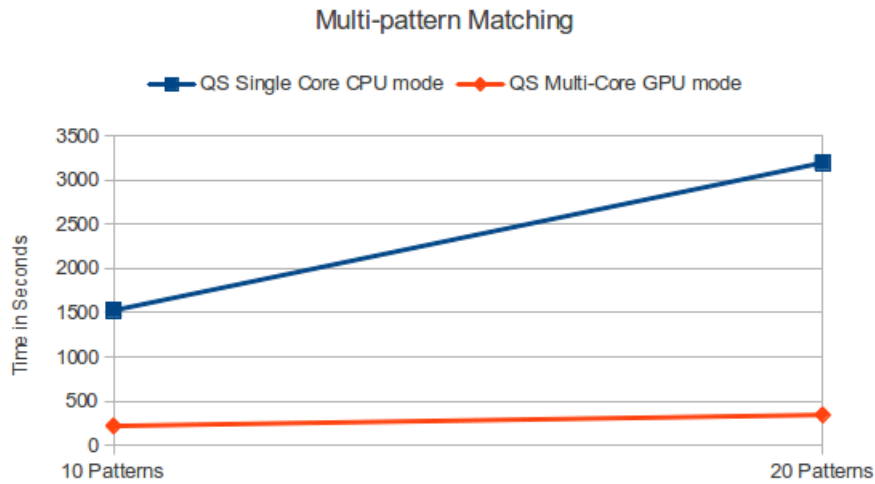


Figure 6.22: DNA multi-pattern matching. Clgrep (QS algorithm) vs. The sequential implementation.

As demonstrated in both Figure 6.21 and Figure 6.22, the orange line is obviously smoother than the blue and confirms that the Parallel Computing execution model of Clgrep is more scalable than the traditional sequential implementations. By fully utilizing the multiple cores of the modern processor, the performance of Clgrep scales in a linear behavior when the number of searching patterns is increasing.

6.4 Summary

In this section, we presented the experiment methodology and results of the study. First, the matching cases used in the experiment were introduced. Then, the experiment settings as well as procedures were described in detail. Finally, the data we collected in the experiment were presented and discussed.

As an alternative to grep, Clgrep was examined in the experiment through a variety of matching cases. Moreover, the use of QS and Horspool algorithms were investigated in the context of Heterogeneous Parallel Computing. The findings will be presented in the following section.

Chapter 7

Future Work

7.0.1 Other Forms of String Matching Problem

There are a number of interesting research paths that could extend the scope of the study. First of all, as we previously introduced, the study only focuses on exact string matching. There are several different forms of string matching problems that have not been considered in the context of Heterogeneous Parallel Computing. It would, therefore, be interesting to explore the use of heterogeneous processing techniques in these problems.

7.0.2 The Development of grep-like String Matching Utilities

Since the amount of data has been increasing, the development of modern grep-like string matching utilities has also become a more important issue. Clgrep only offers basic matching functions despite two algorithms being implemented.

We plan to further develop the Bittap algorithm in Clgrep for both regular expression matching and approximate matching. By continually developing the algorithm, we expect one day Clgrep can be extended to a fully functional string matching utility.

7.0.3 The Evaluation of Appropriate Algorithms

In the previous studies, a variety of string algorithms are proposed, implemented and evaluated on CPU in sequential execution models. On the contrary, only few studies

aim to evaluate the performance of string matching algorithm on heterogeneous computing devices, despite the fact that Heterogeneous Parallel Computing may be the major form of computer in future.

Therefore, it would be prudent to explore the use of other algorithms in the context of Heterogeneous Parallel Computing, especially the algorithms that perfectly suit the Parallel Computing model.

We are interested in hashing-based algorithms, because depending on the special memory accessing patterns, they may potentially achieve outstanding performance on heterogeneous computing devices such as GPU.

7.0.4 Clgrep vs. grep

As we know, grep is based on the idea of line. It is designed for printing the lines in which search patterns are found. Comparing with grep, Clgrep adapts a very different approach. Clgrep is developed for matching every single character. When one search pattern is found in a line, grep will stop searching further in this particular line but Clgrep would not.

Besides, grep is not only an exact string matching utility. It supports both approximate matching and regular expression, in addition, has a number of control options. On the other hand, Clgrep is much lighter weight in terms of control functions.

After carefully examined the source code of grep, we found that in order to fairly compare the matching performance between grep and Clgrep, a significant modification is required. A comparison between grep, one of the most famous string matching tool, and Clgrep would be a very interesting topic; it should be done in the near future, ideally after more control functions and algorithms are implemented in Clgrep.

Chapter 8

Conclusions

String matching is one of the most important subjects in information science. Its efficiency is critical to applications in many cases. To enhance the matching performance, previously, many research efforts have contributed to these areas: both algorithms and processing techniques have been investigated. However, because the issue of string matching exists in many forms crossing multiple areas, different terminologies are used, the boundary between each form of the problem is poorly defined, and some concepts are not even distinguished in certain points of view [23].

In this study, we have widely investigated the problem of string matching. In Chapter 2 the different forms of string matching were clearly distinguished, and the exact on-line string matching problem was defined. Moreover, an overview of string matching algorithm was made in which a number of widely used algorithms were introduced, and the classifications of string matching algorithm were reviewed and discussed.

Furthermore, a review of the current stage of traditional grep-like tools and GPGPU string matching research was made. In Chapter 4 we summarized the algorithms, optimizations and evaluation methodologies used in the previous studies. A number of research gaps were identified, and the research questions of the study were addressed.

A parallel exact matching utility Clgrep was designed, developed and evaluated. Clgrep was developed as an alternative to grep for constitutionally intensive string matching, in addition to support the research of the study. Clgrep features two efficient algorithms, a number of optimizations, and three flexible processing modes. Implemented without vendor-specific extensions, it is able to leverage the matching performance by any OpenCL-capable devices. In light of the experiment results, compared to traditional

sequential implementations of corresponding algorithms, Clgrep can be several times more efficient in computationally intensive matching. To the best of our knowledge Clgrep is the first alternative of grep based on OpenCL and is able to matching on CPU and GPU currently.

In Chapter 5 the algorithms used in Clgrep were introduced. The SIMD Parallel Computing mechanism of Clgrep was described. Moreover, a number of optimizations that considered and examined during developing were discussed.

In addition, a series of well-designed experiments were accomplished in the study. The performance of Clgrep was examined more importantly the use of typical heuristics-based algorithm was investigated in the context of Heterogeneous Parallel Computing.

In Chapter 6 the details of the experiment were described. The procedure, settings and matching scenarios used in the experiment were introduced. The matching cases of the experiment consist of more than 100 combinations overall, including different size of text, different length of pattern, etc. To the best of our knowledge is may be one of the most comprehensive experiment on this specific topic.

For the questions were raised in Chapter 4, the answers have been found in the experiment of the study. Several conclusions can be drawn from the outcomes of experiment.

First of all, the results suggest that single pattern matching may not scale well in a Parallel Computing manner, either on multi-core CPU or GPU. The performance is limited by several factors:

- The initialization cost of OpenCL framework
- The extra cost of employing the Parallel Computing model and managing multiple hardware
- More importantly, the expensive I/O operations of GPU (The detail can be found in the appendix of the thesis.)

On the other hand, multi-pattern matching exhibits parallelism patterns that can easily be exploited by GPGPU Computing regardless of the algorithms used. This has been especially visible in the case of DNA sequence matching. The experiment results suggest that Heterogeneous Parallel Computing can be successfully utilized to enhance the matching performance when more computational capability is required rather than I/O operations.

We also found that multi-core CPU mode also has its special uses. In the case of matching a single pattern in DNA sequences or a set of patterns in a small size text, a sophisticated trade-off between I/O costs and computational capability can be found in the multi-core CPU mode of Clgrep. It may potentially overtake others and achieve the best performance.

The CG mode of Clgrep, on the other hand, has not ever achieved the best performance during the experiment. Despite, in many cases, the performance of CG mode being not poor, its efficiency is always in the middle, compared with other modes. The performance of CG mode is clearly limited by the cost of managing multiple computing devices simultaneously.

In brief, the results suggest that the performances of Heterogeneous Parallel Computing matching, either on multi-core CPU or GPU, are highly related to the computational intensity of certain cases. The fact is that when computational power is intensively required, several times speedup can be achieved. Otherwise, there is no benefit to initialize OpenCL framework then block data to multiple computing devices.

8.1 The Limitation of The Study

Clgrep as a grep-like exact string matching utility offers great flexibility. It can leverage the matching performance by either CPU or GPU. However, as research, the study has a number of limitations, especially in the experiment. In this section these limitations are discussed. Moreover the scope of the study is re-emphasized.

The limitations of the study:

1. Any such experiment is affected by the hardware, OS, the compilers used.
2. Despite all the BM-like heuristics-based algorithms being similar, they are not identical. The algorithms of Clgrep are representative in general, but generalizing our evaluation conclusions to other heuristics-based algorithms may not be appropriate in some cases.
3. The length ranges and numbers of patterns in our experiment are limited. They may represent the majority of use cases but not every case.
4. This study focuses on the comparison between SIMD and SISD. Although MIMD may be another scalable solution for computational intensive string matching it is not included in this research. Instead, a SIMD vs. MIMD study is considered in the future.

In summary, any exhaustive examination is impractical. Any such study is limited one way or another.

The scope of the study:

1. We aim to develop an exact string matching utility from the idea of Heterogeneous Parallel Computing that has its unique features, and can be very efficient in specific cases.
2. We concentrate on investigating the use of QS and Horspool algorithms in the context of Heterogeneous Parallel Computing.

Appendix A

The Source Codes of Clgrep and The Bash Scripts Used in The Experiment

The source code of Clgrep and the Bash scripts used in the experiment have been updated to the github repository of the study, and can thus be easily accessed on-line. It is, therefore, neither listed in the appendix, nor printed in the hard copy of the thesis.

The repository of the study: <https://github.com/xuwupeng2000/CLgrep>

Appendix B

The Experiment Results

87

B.0.1 Single pattern matching

1 MB English Text		10 MB English Text	
Mode	Total Cost	Init and I/O Costs	Total Cost
BMH Multi-Core CPU mode	52.42	39.51	53.73
BMH Multi-Core GPU mode	57.02	40.07	59.43
BMH CG mode	66.01	51.2	67.4
QS Multi-Core CPU mode	52.14	39.24	53.47
QS Multi-Core GPU mode	57.46	40.11	59.97
QS CG mode	62.51	50.86	67.57
100 MB English Text		100 MB DNA	
Mode	Total Cost	Init and I/O Costs	Total Cost
BMH Multi-Core CPU mode	67.45	51.02	77.05
BMH Multi-Core GPU mode	86.2	79.04	97.35
BMH CG mode	93.25	90.72	105.26
QS Multi-Core CPU mode	66.5	50.95	75.78
QS Multi-Core GPU mode	85.59	79.23	98.68
QS CG mode	92.01	87.11	101.98

Figure B.1: The results of single pattern matching.

B.0.2 Multi-pattern matching

Mode	1 MB English Text		10 MB English Text	
	Total Cost	Init and I/O Costs	Total Cost	Init and I/O Costs
BMH Multi-Core CPU mode (10 Patterns)	54.59	40.23	72.03	42.11
BMH Multi-Core GPU mode (10 Patterns)	59.92	40.29	75.95	41.32
BMH CG mode (10 Patterns)	70.14	50.12	89.17	51.35
QS Multi-Core CPU mode (10 Patterns)	54.3	39.78	71.28	41.21
QS Multi-Core GPU mode (10 Patterns)	59.24	40.08	75.16	41.05
QS CG mode (10 Patterns)	68.39	50.32	87.63	52.01
BMH Multi-Core CPU mode (20 Patterns)	57.19	39.29	87.95	41.41
BMH Multi-Core GPU mode (20 Patterns)	60.06	40.17	85.61	41.22
BMH CG mode (20 Patterns)	70.14	49.89	102.5	51.08
QS Multi-Core CPU mode (20 Patterns)	57.05	41.23	86.95	41.44
QS Multi-Core GPU mode (20 Patterns)	60.09	40.35	84.8	41.26
QS CG mode (20 Patterns)	69.39	50.29	100.7	50.99
Mode	100 MB English Text		100 MB DNA	
	Total Cost	Init and I/O Costs	Total Cost	Init and I/O Costs
BMH Multi-Core CPU mode (10 Patterns)	287.06	51.17	372.37	51.28
BMH Multi-Core GPU mode (10 Patterns)	251.81	79.24	231.34	80.07
BMH CG mode (10 Patterns)	240.72	89.13	310.79	90.45
QS Multi-Core CPU mode (10 Patterns)	222.5	51.21	287.6	51.39
QS Multi-Core GPU mode (10 Patterns)	209.63	78.97	218.14	80.08
QS CG mode (10 Patterns)	215.05	90.1	280.24	88.99
BMH Multi-Core CPU mode (20 Patterns)	386.97	51.77	651.96	51.23
BMH Multi-Core GPU mode (20 Patterns)	336.2	80.06	371.61	79.45
BMH CG mode (20 Patterns)	483.06	90.44	485.62	89.85
QS Multi-Core CPU mode (20 Patterns)	379.33	51.82	535.39	52.01
QS Multi-Core GPU mode (20 Patterns)	325.08	79.33	342.87	80.32
QS CG mode (20 Patterns)	365	89.94	426.32	91.33

Figure B.2: The results of multi-pattern matching.

References

- [1] PD Michailidis and KG Margaritis. String matching algorithms: Survey and experimental results. In *International Journal of Computer Mathematics*, volume 76, pages 411–434, 2000.
- [2] M. Crochemore and W. Rytter. *Text algorithms*. Oxford University Press, USA, 1994.
- [3] J.D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A.E. Lefohn, and T.J. Purcell. A survey of general-purpose computation on graphics hardware. In *Computer graphics forum*, volume 26, pages 80–113. Wiley Online Library, 2007.
- [4] Tony Abou-assaleh and Wei Ai. Survey of global regular expression print (grep) tools. Technical report, Dalhousie University, 2004.
- [5] 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*, pages 116–134. Springer, 2008.
- [6] 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*, pages 87–88. ACM, 2010.
- [7] N.F. Huang, Y.M. Chu, and H.W. Hsu. Graphics processor-based high performance pattern matching mechanism for network intrusion detection. 2011.
- [8] G. Vasiliadis, M. Polychronakis, and S. Ioannidis. Parallelization and characterization of pattern matching using GPUs. In *Workload Characterization (IISWC), 2011 IEEE International Symposium*, pages 216–225. IEEE, 2011.
- [9] A.V. Aho and M.J. Corasick. Efficient string matching: an aid to bibliographic search. *Communications of the ACM*, 18(6):333–340, 1975.

- [10] D.M. Sunday. A very fast substring search algorithm. *Communications of the ACM*, 33(8):132–142, 1990.
- [11] R.N. Horspool. Practical fast searching in strings. *Software: Practice and Experience*, 10(6):501–506, 1980.
- [12] J.D. Owens, M. Houston, D. Luebke, S. Green, J.E. Stone, and J.C. Phillips. GPU computing. *Proceedings of the IEEE*, 96(5):879–899, 2008.
- [13] M. Scarpino. *OpenCL in Action*. Manning Publications, New York, 2012.
- [14] C. Charras and T. Lecroq. *Handbook of exact string matching algorithms*. King’s College Publications, 2004.
- [15] M. Crochemore, C. Hancart, and T. Lecroq. *Algorithms on strings*. Cambridge University Press, 2007.
- [16] U. Manber, S. Wu, et al. Glimpse: A tool to search through entire file systems. In *Usenix Winter 1994 Technical Conference*, pages 23–32, 1994.
- [17] M.C. Schatz, C. Trapnell, A.L. Delcher, and A. Varshney. High-throughput sequence alignment using graphics processing units. *BMC Bioinformatics*, 8(1):474, 2007.
- [18] G. Vasiliadis and S. Ioannidis. Gravity: a massively parallel antivirus engine. In *Recent Advances in Intrusion Detection*, pages 79–96. Springer, 2010.
- [19] G. Navarro and M. Raffinot. *Flexible pattern matching in strings: practical on-line search algorithms for texts and biological sequences*. Cambridge University Press, 2002.
- [20] P. Weiner. Linear pattern matching algorithms. In *Switching and Automata Theory, 1973. SWAT’08. IEEE Conference Record of 14th Annual Symposium*, pages 1–11. IEEE, 1973.
- [21] D.E. Knuth, J.H. Morris Jr, and V.R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2):323–350, 1977.
- [22] R.S. Boyer and J.S. Moore. A fast string searching algorithm. *Communications of the ACM*, 20(10):762–772, 1977.

- [23] P. Lin, Z. Li, Y. Lin, Y. Lai, and FC Lin. Profiling and accelerating string matching algorithms in three network content security applications. *IEEE Communications Surveys & Tutorials*, 8(2):24–37, 2006.
- [24] G. Navarro. Nr-grep: a fast and flexible pattern-matching tool. *Software: Practice and Experience*, 31(13):1265–1312, 2001.
- [25] M. Crochemore and D. Perrin. Two-way string-matching. *Journal of the ACM (JACM)*, 38(3):650–674, 1991.
- [26] R.M. Karp and M.O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, 1987.
- [27] G. Navarro and M. Raffinot. A bit-parallel approach to suffix automata: Fast extended string matching. In *Combinatorial Pattern Matching*, pages 14–33. Springer, 1998.
- [28] S. Wu, U. Manber, et al. A fast algorithm for multi-pattern searching. Technical report, TR-94-17, University of Arizona, 1994.
- [29] A. Hume and D. Sunday. Fast string searching. *Software: Practice and Experience*, 21(11):1221–1248, 1991.
- [30] M. Crochemore, A. Czumaj, L. Gasieniec, S. Jarominek, T. Lecroq, W. Plandowski, and W. Rytter. Speeding up two string-matching algorithms. *Algorithmica*, 12(4):247–267, 1994.
- [31] A. Apostolico and R. Giancarlo. The boyer-moore-galil string searching strategies revisited. *SIAM Journal on Computing*, 15(1):98–105, 1986.
- [32] T. Berry and S. Ravindran. A fast string matching algorithm and experimental results. In *Proceedings of the Prague Stringology Club Workshop*, volume 99, pages 16–28, 1999.
- [33] T. Lecroq. Experimental results on string matching algorithms. *Software: Practice and Experience*, 25(7):727–765, 1995.
- [34] T. Lecroq. New experimental results on exact string-matching. *Rapport LIFAR*, 22(11):45–69, 2000.

- [35] J.E. Stone, D. Gohara, and G. Shi. OpenCL: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering*, 12(3):66, 2010.
- [36] D.B. Kirk and W.H. Wen-mei. *Programming massively parallel processors: a hands-on approach*. Morgan Kaufmann, 2010.
- [37] B. Gaster, L. Howes, D.R. Kaeli, P. Mistry, and D. Schaa. *Heterogeneous Computing with OpenCL*. Morgan Kaufmann, 2011.
- [38] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to Parallel Computing*, volume 110. Benjamin/Cummings USA, 1994.
- [39] C. AMD. Accelerated parallel processing: OpenCL programming guide. <http://developer.amd.com/sdks/AMDAPPSDK/documentation>, 2011. [Online; accessed 12-March-2013].
- [40] C. Nvidia. OpenCL programming guide for the CUDA architecture, version 3.2. http://www.nvidia.com/content/cudazone/download/OpenCL/NVIDIA_OpenCL_ProgrammingOverview.pdf, 2009. [Online; accessed 12-March-2013].
- [41] T.G. Mattson, B.A. Sanders, and B. Massingill. *Patterns for Parallel Programming*. Addison-Wesley Professional, 2005.
- [42] R. Smith, N. Goyal, J. Ormont, K. Sankaralingam, and C. Estan. Evaluating GPUs for network packet signature matching. In *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium*, pages 175–184. IEEE, 2009.
- [43] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: stream computing on graphics hardware. In *ACM Transactions on Graphics (TOG)*, volume 23, pages 777–786. ACM, 2004.
- [44] W.R. Mark, R.S. Glanville, K. Akeley, and M.J. Kilgard. Cg: A system for programming graphics hardware in a c-like language. In *ACM Transactions on Graphics (TOG)*, volume 22, pages 896–907. ACM, 2003.
- [45] Michael D. McCool, Zheng Qin, and Tiberiu S. Popa. Shader metaprogramming. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, HWWS '02, pages 57–68, Aire-la-Ville, Switzerland, Switzerland, 2002. Eurographics Association.

- [46] C. Nvidia. Nvidia CUDA C Programming Guide. http://www.math.vt.edu/technology/facilities/cuda/NVIDIA_CUDA_C_ProgrammingGuide_3.1.pdf, 2010. [Online; accessed 12-March-2013].
- [47] Khronos OpenCL Working Group. The OpenCL Specification Version 1.1, 2010. <http://www.khronos.org/registry/cl/specs/>, 2010. [Online; accessed 13-March-2013].
- [48] A. Hume. A tale of two greps. *Software: Practice and Experience*, 18(11):1063–1072, 1988.
- [49] Z. Ahmad, M.U. Sarwar, and M.Y. Javed. Agrep- a fast approximate pattern-matching tool. *Journal of Applied Sciences Research*, 2(10):741–745, 2006.
- [50] S. Antonatos, K.G. Anagnostakis, and E.P. Markatos. Generating realistic workloads for network intrusion detection systems. In *ACM SIGSOFT Software Engineering Notes*, volume 29, pages 207–215. ACM, 2004.
- [51] J.B.D. Cabrera, J. Gosar, W. Lee, and R.K. Mehra. On the statistical distribution of processing times in network intrusion detection. In *Decision and Control, 2004. CDC. 43rd IEEE Conference*, volume 1, pages 75–80. IEEE, 2004.
- [52] N. Jacob and C. Brodley. Offloading ids computation to the GPU. In *Computer Security Applications Conference, 2006. ACSAC’06. 22nd Annual*, pages 371–380. IEEE, 2006.
- [53] S. Faro, T. Lecroq, C. Barton, M. Giraud, C. Iliopoulos, T. Lecroq, L. Mouchard, SP Pissis, H. Bouhamed, T. Lecroq, et al. The exact online string matching problem: a review of the most recent results. *ACM Computing Surveys*, 11:41–68.
- [54] M. Onsjö and Y. Aono. Online approximate string matching with CUDA. <http://odinlake.net/wordpress/wp-content/uploads/2009/03/pattmatch-report.pdf>, 2009. [Online; accessed 19-July-2008].
- [55] C.S. Kouzinopoulos and K.G. Margaritis. String matching on a multicore GPU using CUDA. In *Informatics, 2009. PCI’09. 13th Panhellenic Conference*, pages 14–18. IEEE, 2009.
- [56] M. Fisk and G. Varghese. Applying fast string matching to intrusion detection. Technical report, Los ALAMOS National Lab NM, 2002.

- [57] X. Zha and S. Sahni. Multipattern string matching on a GPU. In *Computers and Communications (ISCC), 2011 IEEE Symposium*, pages 277–282. IEEE, 2011.
- [58] M. McCool, S. Du Toit, T. Popa, B. Chan, and K. Moule. Shader algebra. In *ACM Transactions on Graphics (TOG)*, volume 23, pages 787–795. ACM, 2004.
- [59] E. Wu and Y. Liu. Emerging technology about GPGPU. In *Circuits and Systems, 2008. APCCAS 2008. IEEE Asia Pacific Conference*, pages 618–622. IEEE, 2008.
- [60] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. Nvidia tesla: A unified graphics and computing architecture. *Micro, IEEE*, 28(2):39–55, 2008.
- [61] A. Munshi, B. Gaster, and T.G. Mattson. *OpenCL Programming Guide*. Addison-Wesley Professional, 2011.
- [62] A.T.I.S. Computing. OpenCL programming guide. *AMD Corporation, August*, 2010.
- [63] M.J. Flynn and K.W. Rudd. Parallel architectures. *ACM Computing Surveys (CSUR)*, 28(1):67–70, 1996.
- [64] A. Gee. Research into GPU accelerated pattern matching for applications in computer security. 2008. [Online; accessed 19-July-2008].