Hardware Implementation of a String Matching Algorithm Based on the FM-Index

A Dissertation submitted in partial satisfaction of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Edward Bryann Cabanayan Fernandez

March 2013

Dissertation Committee:

Professor Walid Najjar, Chairperson
Professor Stefano Lonardi
Professor Sheldon Tan
Professor Frank Vahid
The Dissertation of Edward Bryann Cabanayan Fernandez is approved:

Committee Chairperson

University of California, Riverside
Acknowledgments

I am very grateful to my advisers, Walid Najjar and Stefano Lonardi, in guiding this research resulting to an outcome far more than my expectations. I am very thankful to my colleague, Jason Villareal, which I regard as my team mate in successfully completing this research. I would like to thank Roger Mousalli, Robert Halstead, Steve Cole, Casey Czechowski, Monik Khare, and Nathan Diep, my co-graduate students whom I share my struggles and hardships as part of graduate student life. I would like to thank Terry Phonharath, Amy Ricks and Madie Heersink who are always ready to help and assist all graduate students the best way possible.
To my family and friends,

advisers and colleagues,

and our Creator.
ABSTRACT OF THE DISSERTATION

Hardware Implementation of a String Matching Algorithm Based on the FM-Index

by

Edward Bryann Cabanayan Fernandez

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, March 2013
Professor Walid Najjar, Chairperson

String matching is the searching of patterns in a very long string called text. It is involved in DNA sequence mapping that matches millions of short patterns, called reads, on a reference genome. The length of the reads is in the range of 36 to 150 characters and a typical genome length is billions of characters. The processing massive amount of data led to the development of advanced algorithms. The FM-index, based on the Burrows-Wheeler Transform, is a recently developed data structure utilized by the fastest software tool to map millions of reads on a reference genome. Although the FM-index is a very sophisticated tool used for mapping, current software tools still need faster execution due to rapidly increasing data because of improved sequencing technologies.

The focus of this research is improving the execution time of existing string matching algorithm based on the FM-index through hardware acceleration using FPGAs. We introduce FHAST (FPGA Hardware Accelerated Sequence-matching Tool) as an accelerator acting as a drop-in replacement for Bowtie, an industry-accepted software mapping tool. FHAST uses a multi-threaded architecture masking external memory latency by executing
concurrent hardware threads. FHAST is implemented on a Convey HC-1 supercomputing system to take advantage of high memory bandwidth and shared memory space of hardware and software. We observe an actual speed up as high as 70x compared to Bowtie, which reduces execution runs from hours to minutes.
Contents

List of Figures x

List of Tables xiv

1 Introduction 1

2 Background 7
  2.1 Introduction .................................................. 7
  2.2 String Matching Algorithms ................................. 8
    2.2.1 Naive approach ......................................... 8
    2.2.2 Finite State Machines ................................. 9
    2.2.3 Dynamic Programming ................................. 10
    2.2.4 Index-Based searching ............................... 12
    2.2.5 Summary of Algorithms ............................ 15
  2.3 String Matching Hardware Approaches .................. 17
    2.3.1 Naive Approach .................................. 17
    2.3.2 Finite State Machines .......................... 20
    2.3.3 Dynamic Programming .......................... 24
    2.3.4 Index-Based Searching .......................... 26
  2.4 String Matching Approaches on GPU ................... 29
  2.5 Conclusion .............................................. 31

3 Exact String Matching 33
  3.1 Introduction ............................................... 33
  3.2 Finite State Machines .................................... 35
  3.3 Encoded Patterns ......................................... 41
  3.4 Hybrid Approach ......................................... 46
  3.5 Patterns on Content Addressable Memory ............. 51
  3.6 Evaluations ................................................ 53
  3.7 Conclusion ................................................ 55
4 Approximate String Matching
4.1 Introduction .............................................. 57
4.2 Naive Approach ........................................... 59
4.3 Convolution Based Approach .............................. 67
4.4 Evaluations ................................................ 71
4.5 Conclusion .................................................. 75

5 Hardware Implementation of the FM Index .................. 77
5.1 Introduction .................................................. 77
5.2 Data Structures of the Burrows-Wheeler Transform .......... 79
5.3 Pattern Searching Using the FM-Index in Hardware ............. 81
5.3.1 Searching and Locating a Pattern ....................... 82
5.4 Hardware FM-index using Block RAMs ....................... 87
5.4.1 Architecture ........................................... 87
5.4.2 Evaluation .............................................. 90
5.5 Hardware FM-index using External Memory ................... 96
5.5.1 Exact Matching Architecture ............................ 96
5.5.2 Approximate Matching Architecture ........................ 98
5.5.3 Evaluations ............................................ 102
5.6 Conclusion ................................................. 106

6 Summary and Conclusion ..................................... 107

Bibliography ..................................................... 110
List of Figures

2.1 (a) Cell Z depends on Cell A, B, C of a similarity score matrix. (b) Directions of how cells of the similarity score matrix are computed. 11
2.2 Each hash function corresponds to a value which points to a location of the seed on the text. ................................................. 13
2.3 (a) Example of a suffix tree (b) Example of a suffix array .... 14
2.4 Architecture of an intrusion detection system. Packets enter all rule blocks in parallel where each rule block flags a signal of any malicious content. ........................... 18
2.5 Architecture of an intrusion detection system. Packets are fed through a fan-out tree that distributes the payload to matching engines. Matched rules are reported by the output encoder. ... 19
2.6 Aho-corasick bit-split implementation of peptide sequence matcher. Each symbol is broken down to five bits where each bit is used as input to simple FSMs. All FSMs must agree before a match is reported. .......................................................... 20
2.7 Mapping of logic structures to create a NFA of regular expression ((a—b)*(cd)). ................................................................. 22
2.8 Example of character decoder showing reduced hardware resources and routing compared to comparators. ......................... 23
2.9 Off-diagonals of the similarity score matrix are processed in parallel creating the wavefront algorithm. ................................. 24
2.10 Diagonal paths on the similarity score matrix represent seeds where the BLAST heuristic starts. ................................. 26
2.11 Architecture of hardware string matching using Bloom filters. Symbols are streamed and several portions of streaming window are fed in parallel to N Bloom filters. ................................. 28
2.12 Pattern matching unit of the trie. Match input is the signal that a character matches from a previous trie level. Sum of the 1’s is used to get the address of the next trie level from memory. The contents are compared to the next incoming symbols. .... 29
3.1 Block diagram of architecture implementing exact string matching implementations. ........................................ 34
3.2 Throughput versus number of strings for groups of size 4 to 16 with a pattern length of 36 and groups of size 8 to 32 for a pattern length of 16. .......................................................... 37
3.3 FPGA area versus number of strings for groups of size 4 to 16 with a pattern length of 36 and groups of size 8 to 32 for a pattern length of 16. .......................................................... 38
3.4 Throughput/slice versus number of strings for groups of size 4 to 16 with a pattern length of 36 and groups of size 8 to 32 for a pattern length of 16. .......................................................... 39
3.5 Number of FSM states versus pattern length for group sizes of 4, 8, 16, and 32 using the Aho-Corasick implementation. ............ 40
3.6 A match occurs when both the common prefix and individual suffix of a pattern match. .................................................. 41
3.7 Throughput versus number of strings for groups of size 1 to 16 with a pattern length of 36 and 16. ........................................ 42
3.8 Area versus number of strings for groups of size 1 to 16 with a pattern length of 36 and 16. ........................................ 43
3.9 Throughput/slice versus number of strings for groups of size 1 to 16 with a pattern length of 36 and 16. ........................................ 44
3.10 Logic units versus group size for 4096 patterns of 36 characters in length. .................................................. 45
3.11 A match on the prefix enables a latch that enables the suffix state machine. .................................................. 46
3.12 Throughput versus number of strings for groups of size 1 to 16 with a pattern length of 36 and 16. ........................................ 48
3.13 Area versus number of strings for groups of size 1 to 16 with a pattern length of 36 and 16. ........................................ 49
3.14 Throughput/slice versus number of strings for groups of size 1 to 16 with a pattern length of 36 and 16. ........................................ 50
3.15 The text streams in a shift register. The contents of the shift register are used as the word input of the CAM. ............ 51
3.16 Frequency vs CAM word count for two word lengths with 67% of the total number of slices of the FPGA. ..................... 52
3.17 Throughput/slice of each exact string matching implementation for two pattern lengths. ........................................ 54
4.1 A read is held in a register (here shown with four characters) for the duration of the current stream. The stream, of 2-bits per character, is run through a shift register. .......................... 59
4.2 Diagram of a four-bit pipelined tree adder. ...................... 60
4.3 Chaining of stream shift registers. ................................... 60
4.4 A block of four streams matched against one register. ........ 61
4.5 A portion of the entire architecture showing three chained blocks of eight shift registers each. ................................. 61
4.6 Number of string comparisons versus read length for 1 to 8 streams with 90% of the total number of slices of the FPGA (60% register slices) using the naive implementation allowing one mismatch and three mismatches. .................................................................................. 63
4.7 Operating frequency versus read length for 1 to 8 streams with 90% of the total number of slices of the FPGA (60% register slices) using the naive implementation allowing one mismatch and three mismatches. .................................................................................. 64
4.8 Throughput versus read length for 1 to 8 streams with 90% of the total number of slices of the FPGA (60% register slices) using the naive implementation allowing one mismatch and three mismatches. 65
4.9 An example of convolution-based string matching between read P and genome T. The output is a vector of integers representing the total number of matches between P and T, for all possible locations in T. ............................................................................................................ 68
4.10 Frequency versus genome length for read lengths of 16 and 36. 69
4.11 Area versus genome length for read lengths of 16 and 36. ..... 70
4.12 Throughput versus genome length for read lengths of 16 and 36. 71
4.13 Throughput/slice of each approximate string matching implementation for two read lengths. .............................................. 73

5.1 Example of sorting the BWT(Q). .................................................. 81
5.2 Example of searching the read ”TAGG” on the string ”GCTAATTAGGTACC” using the FM-index. After the 4th iteration the read is found because the index of the top pointer is less than the bottom pointer. .................................................................................................................. 83
5.3 Example of locating the read ”TAGG” on the string ”GCTAATTAGGTACC” using the third approach. After the 2nd iteration the read is located on position 6 of the genome. ........................................... 85
5.4 Example of searching the read ”CCGA” on the string ”GCTAATTAGGTACC” using the FM-index. After the 2nd iteration the read is not found because the top and bottom pointers index the same suffix array location. ...................................................... 86
5.5 Block diagram for searching a read on a genome using the FM-index. 88
5.6 Block diagram for searching and locating a read on a genome using the FM-index. ................................................................. 89
5.7 Controller used for architecture. .................................................... 90
5.8 Operating frequency and area versus genome section length. Note that the frequency decreases by only 20% as the genome section size increases from 1K to 16K. Logic area decreases exponentially as we increase the genome subsection length. ........................................ 92
5.9 Block diagram of exact string matching algorithm using external memory. ................................................. 97
5.10 Update block including a block RAM for precalculated addresses. ....................................................... 99
5.11 Block diagram of approximate matching architecture using n exact string matching architectures. ......................................................... 99
5.12 A failing character of the read creates three copies of the read with the failing character replaced. .......................................................... 100
5.13 Update block including the replace block for approximate matching. .................................................. 101
5.14 Locate block have send and receive queues similar to send and receive blocks for finding the location of a read from the suffix array residing in main memory. .................................................. 102
5.15 Role of software is mainly on memory allocation and reporting. Hardware performs the search algorithm. ................................. 103
5.16 Speed up of FHAST and Bowtie for exact matches, one and two mismatches. .................................................. 105
# List of Tables

2.1 Summary of String Matching Algorithms .......................... 16
3.1 Throughput and area comparison of the five exact string matching implementations. .................................................. 53
3.2 Qualitative comparison of four exact string matching implementations. ................................................................. 55
4.1 Throughput and area comparison of naive and convolution-based string matching implementations. ......................... 71
4.2 Qualitative comparison of naive and convolution-based string matching implementations. ................................. 72
4.3 Execution times of software tools for read lengths of 16 and 36 considering zero and three mismatches. ....................... 74
4.4 Comparison of software execution time to FPGA execution time. ................................................................. 74
5.1 Example of deriving the Burrow-Wheeler Transform of a genome. The genome is terminated by the "$" symbol. ................. 80
5.2 Example of deriving the suffix array of a genome. ..................... 81
5.3 I-Table stores the first occurrence of each character on the sorted BWT(Q). ........................................................... 81
5.4 The C-table stores the count of each character on a previous location. The leftmost column of the table shows the indices of the suffix array. ................................................................. 82
5.5 Resource utilization and Frequency of FM-index ..................... 91
5.6 Overhead due to genome overlap for search string length 36. .... 93
5.7 Character comparisons (CC) per second for FM-index and naive implementations. .................................................. 93
5.8 Expected length of shared suffixes between two substrings for a specific genome length. ........................................ 95
5.9 Execution times of Bowtie and FPGA Implementation with the number of matching DNA sequences on the E-coli genome in percentages. ........................................... 95
5.10 Specifications of CPU running Bowtie. .......................... 104
5.11 Execution times of Bowtie and FPGA Implementation with the number of matching DNA sequences on the E-coli genome in percentages. .............................................. 105
Chapter 1

Introduction

String matching is a fundamental problem in computer science where short strings of specific length referred to as patterns are searched on a very long string referred to as text. The problem is found in a wide variety of applications. These applications can be classified in two types: exact and approximate string matching. Exact string matching only considers perfect matches whereas approximate string matching allows a maximum number of mismatches. The number of allowed mismatches is application dependent.

Examples of applications that use exact string matching are network intrusion detection, network routing and database querying. In network intrusion detection, a database containing patterns of known network attacks (e.g. SNORT) is searched on a stream of packets. Exact matching is utilized to screen packets containing signatures of any known network threats. In network routing, the headers of packets are matched in a routing table to determine the next destination of a packet. In database systems, exact matching is performed to retrieve records when processing a query. Examples of applications that
use approximate string matching are spell-checkers and DNA sequence mapping. A spell-
checker matches mispelled words to the closest correct word in a dictionary with an allowed
number of character mismatches. In DNA sequence mapping, approximate matching is used
to compensate errors introduced by sequencing machines or to detect slight differences in
reference genomes that could be regarded as mutations.

The focus of this thesis is on DNA sequence mapping. DNA sequence mapping re-
quires one to find the locations of a set of short DNA sequences in a reference genome. Short
DNA fragments, also called as reads, are generated by sequencing instruments. Listed be-
low are some of the companies and organizations that developed next generation sequencing
technologies:

- **Illumina/Solexa sequencing technology** is based on dye-terminators. Specialized dyes
  behave differently to each nucleotide. The nucleotide is exposed to these dyes and the
  sequencing machine with a specialized camera identifies the nucleotide based on the
  color reaction.

- **ABI SOLiD technology** is based on ligation. Sequencing by ligation uses the enzyme
  that enables the bonding of DNA strands to determine the base pair at a position of
  a DNA sequence.

- **Ion Torrent technology** is based on hydrogen ion detection. A DNA strand is exposed
to one type of a nucleotide which triggers the release of hydrogen ions. A nucleotide
  of the DNA strand is identified because the amount of hydrogen ions release serves as
  signature to the nucleotides present.
These sequencing instruments, collectively called as ”next generation”, generate hundreds of millions of reads from a single three-four days run. The length of the reads range from 36 to 150 nucleotides. The length is progressively increasing as sequencing technologies improve. The reference genome can have sizes ranging from $10^6$ to $10^9$ base pairs. A maximum number of character mismatches is allowed in the search due to possible sequencing errors on the reads or mutations compared to the reference genome.

Given the massive amount of data that needs processing, DNA sequence mapping demand efficient tools and algorithms to speed up the search process. Various algorithms were developed to address DNA sequence mapping. The FM-index is a recently developed data structure utilized by the fastest software tools to map millions of reads to reference genomes [25]. The FM-index is based on Burrows-Wheeler transform [10]. The search is performed on the index constructed on the reference genome. Although the FM-index is a very sophisticated algorithm used for mapping, current software tools using the algorithm still need faster execution due to increasing data from improving sequencing technologies.

A solution to improve the speed of existing string matching software algorithms is to implement the algorithm directly in hardware, which is the focus of this research. One of the platforms that enables this approach are Field Programmable Gate Arrays (FPGA), which are programmable integrated circuits typically used as prototyping or debug boards before implementing components as Application Specific Integrated Circuits (ASIC) on silicon. Recent technologies and studies have used FPGAs as hardware accelerators in contrast to ASICs because of programmability and reduced design time. Numerous studies have shown qualitatively and quantitatively the possibility of using FPGAs as hardware
accelerators across different applications. FPGAs are used to accelerate processing of XML queries that exploits inherent parallelism existent in XPath systems [44]. FPGAs are also used to accelerate a pattern-based decomposition machine learning algorithm [45]. In the field of accounting, FPGAs are used to speed up Monte Carlo, a market estimation application [28]. In the field of image processing, FPGAs are used to accelerate graphics and imaging algorithms [68]. Hardware acceleration based on FPGAs is possible because of these main factors [29]:

- More parallelism thru creation of multiple copies of processing units which executes unrolled loops of an assembly code.

- Elimination of instructions fetched from memory resulting to memory access totally allocated for data.

- Basic blocks of an assembly code are implemented as dataflows on the hardware resulting to a straight-forward execution.

- Datapaths and execution units are directly implemented to the FPGA eliminating latency of support instructions such as branches and jumps.

Here the main application is DNA sequence mapping. The novel contributions of this thesis are hardware implementations of string matching algorithms, specifically on FPGAs:

- Implementation and design space exploration of multiple pattern matching using finite state machines, shift registers and constants specific for exact string matching.
• Implementation and design space exploration of a multiple stream multiple pattern matching approach using the naive string matching approach specific for approximate string matching.

• Implementation and design space exploration of convolution-based approximate string matching.

• Development, implementation and design space exploration of an exact string matching based on the FM-index.

• Development, implementation and evaluations of approximated string matching based on the FM-index.

Listed are chapters of this dissertation:

• Second chapter: We comprehensively explore the literature on string matching. We start our discussion on general operations of string matching algorithms. We classify the algorithms and discuss strengths and weaknesses of each approach. We then follow with a discussion about hardware implementations of string matching algorithms. We identify advantages and disadvantages of various implementations to help us in developing our own hardware implementation.

• Third chapter: We develop four exact string matching implementations namely: finite state machines, encoded patterns, hybrid implementation and CAM-based implementation. We explore the design space of each approach to identify the best configuration that maximizes throughput per logic area. We then compare, analyze and evaluate strengths and weaknesses of each implementations.
• Fourth chapter: We develop two approximate string matching implementations namely: naive approach and convolution-based approach. We again explore the design space of each approach to maximize the throughput per logic area of each implementation. We again compare, analyze and evaluate strengths and weaknesses of each implementation. We then compare our best implementation to existing software tools and identify the need for a smarter algorithm implemented in hardware to obtain a substantial performance gain versus current software tools.

• Fifth chapter: We introduce, develop and evaluate the first hardware implementation of an exact string matching algorithm based on the FM-index. We explore the design space to minimize execution time and compare to Bowtie [34], a DNA sequence mapping tool based on the FM-index. We expand our implementation to approximate string matching using external memory resulting to the development of FHAST (FPGA Hardware Accelerated Sequencing Tool) as a drop-in replacement for Bowtie. FHAST is a sequencing tool using the Convey super computing system that exploits FPGA coprocessors as hardware accelerators. Preliminary evaluations demonstrates an actual speed up of 70x compared to a single core processor.

• Sixth chapter: We summarize our entire research, draw conclusions and identify future work and areas of improvement to obtain a higher performance gain, more accurate results, and expand capability similar to existing software tools.
Chapter 2

Background

2.1 Introduction

Various algorithms and heuristics were developed to address the problem of string matching. These algorithms have been implemented as software tools for a fast automated execution. As massive data processing is required in current applications, software execution times are becoming unacceptable. This demands direct implementation of algorithms in hardware for high-speed performance. Various hardware implementations of string matching algorithms were developed across different fields of application to obtain a faster and more efficient operation.

In this chapter, we present a broad overview of string matching relevant to implementing the algorithms in hardware. Listed are the major subsections of this chapter:

1. The first section classifies and discusses the general methodology of fundamental string matching algorithms. We identify and compare strengths and weaknesses of each
2. The second section discusses string matching hardware approaches in various applications implemented on FPGAs. We classify and describe the general architecture of the hardware implementation in reference to the algorithm utilized.

3. The third section provides a brief overview of GPUs used as accelerators for string matching. We cite studies focused on string matching utilizing GPUs to complete an overall perspective of string matching operating separately on a coprocessor.

4. The last section summarizes the chapter and identifies learning points useful for developing a better performing string matching approach.

2.2 String Matching Algorithms

Given the vast applications where string matching is relevant and critical, computer scientists developed a wide variety of algorithms to achieve a faster and smarter search process. These algorithms are implemented as software tools that efficiently speed up the matching process. We group the algorithms into four approaches: Naive, Automata, Dynamic Programming, and Index-Based search.

2.2.1 Naive approach

The naive or brute force approach checks the pattern on every position of the text. This approach is very simple and requires no pre-processing of the pattern or the text. A character-to-character comparison of the pattern and the text is evaluated. The pattern is
then shifted by one character location of the text and another set of character comparisons is evaluated. The pattern is checked on all possible locations of the text resulting in a worst case run time of $O(nm)$, where $n$ and $m$ are the lengths of the pattern and the text.

The Knuth-Morris-Pratt (KMP) algorithm was developed to improve the shifting of the pattern position of the Naive approach [33]. Instead of shifting only one character position, the algorithm shifts the pattern by $n$ positions based on a precomputed shift count for each character of the pattern. The shift count is called as the failure function and is computed during preprocessing of the pattern. The worst case run time of the algorithm is $O(n)$, where $n$ is the length of the text.

The Boyer-Moore algorithm further improves the shifting of the pattern position for a better nominal run time [9]. The shifting of the pattern is based on two rules: 1) In the Bad character rule, the pattern is shifted based on failure function almost similar to the KMP algorithm if a character mismatch is encountered. 2) In the Good suffix rule, the pattern is shifted based on matching subsections of the pattern within the pattern itself. These two rules requires preprocessing of the pattern. The overall worst case run time of the algorithm is still $O(nm)$. However, the algorithm improves the nominal run time to $O(n+m)$.

### 2.2.2 Finite State Machines

Automata or finite state machines (FSM) used in string matching is performed in two phases: the first phase is the creation of the finite state machine based on the pattern and the second phase is the search phase where characters of the text are sequentially
inputted to the FSM. The basic approach of illustrating FSM generated from a pattern is through graphs. The nodes of the graph indicate spelled subpatterns and the edges indicate state transitions upon receiving a character. The worst case run time of the algorithm is \( O(n) \), where \( n \) is the length of the text.

The Aho-Corasick (AC) algorithm was developed for multiple pattern searching [2]. A group of patterns is preprocessed to create a FSM for the entire group. Preprocessing outputs three functions: the goto, failure, and output functions. The FSM is constructed from these three functions. The goto function constructs a graph in such a way that each path of the graph spells a member pattern. The failure function represents state transitions of the FSM when a received character fails a pattern path. The output function represents states of the FSM where a pattern matches. Characters of the text are then sequentially inputted to the FSM. The worst case run time is only dependent on the length of the text resulting to \( O(n) \) where \( n \) is the length of the text.

2.2.3 Dynamic Programming

A major and sophisticated approach in string matching is dynamic programming. These algorithms compute the edit distance of two strings. Edit distance is the number of character conversions to transform one string to the other. The two main dynamic programming algorithms for string matching are the Needleman-Wunch and the Smith-Waterman algorithms.

Needleman-Wunch (NW) focuses on global alignment of two strings [47]. Global alignment attempts to match every character of the two strings. This alignment scheme
is appropriate when two strings are almost similar in length. On the other hand, Smith-Waterman (SW) focuses on local alignment [55]. Local alignment attempts to align a short string in specific regions of a much longer string. This alignment scheme is appropriate when two strings have a significant difference in length such as patterns and texts.

Both algorithms involve a scoring mechanism referring to gap and similarity scores. Gap scores are values corresponding to character pairs. The scores are application-dependent constants and independent of the two strings. It is expressed in a matrix in such a way that each matrix element is the gap score of each character pair.

![Figure 2.1: (a) Cell Z depends on Cell A, B, C of a similarity score matrix. (b) Directions of how cells of the similarity score matrix are computed.](image)

Similarity score is a measure of similarity of two strings for every string-to-string length. It is represented as a matrix. A higher similarity score corresponds to greater alignment. Figure 2.1 (a) shows that the cell of the matrix is a function based on three previously computed cells. This function depends on the algorithm and the gap score matrix. Figure 2.1 (b) shows that the first column and row of the matrix is initialized
to zero. Starting from the top-left cell, matrix elements are successively computed until
the matrix is filled. To determine the alignment, we find the element with the highest
value and trace back a path until an element of zero value is reached. The detected path
determines the alignment of the two strings: a diagonal direction implies an alignment,
vertical direction implies a deletion, and a horizontal direction implies an insertion.

The two algorithms, Needleman-Wunch (NW) and Smith-Waterman (SW), differ
in computation of similarity scores. The NW uses negative scores and the SW converts any
negative scores to zeroes. Both algorithms have a worst case run time of $O(nm)$ because of
filling up the similarity score matrix.

### 2.2.4 Index-Based searching

An index-based search preprocesses the text to create an index. The index is
an alternative representation of the text so that searching is performed efficiently. Three
approaches were developed in creating an index: 1) Hash-based index, 2) Suffix-based index,
and the 3) FM-index.

Using hashes in pattern matching originates from the Rabin-Karp algorithm [15].
This algorithm uses hash values from a hash function to classify patterns in groups. Patterns
with equal hash values have a degree of similarity but are not perfect matches. Thus,
verification between patterns in a group is required before considering a match.

To find pattern locations, hash values associated with hash functions are entries in
a hash table. Each hash value corresponds to a *seed*, which is a short sequence with a known
text location. The hash table serves as a list of locations where the *seeds* appear. Figure
2.2 shows that each hash function corresponds to a location on the text. In performing a search, the hash functions are applied to a pattern. The resulting hash value indicates a partial or possible hit on the listed location. Further verification on the location listed is required to confirm a match.

In a suffix-based index, all possible suffixes of the text are preprocessed in creating the index. An example of a suffix-based index is a suffix tree. A suffix tree is a representation of all suffixes of the text where each leaf corresponds to one suffix [62]. All leaves are connected to the root through edges and nodes. An edge represents a substring of the suffix. A node connects edges of leaves with common prefixes. A leaf sharing no common string is connected directly to the root. Searching a pattern starts from the root by comparing edges moving to a node until a leaf is reached. The worst case run time is $O(m)$ where $m$ is the length of the pattern. The problem with the suffix tree is the space required in creating
and storing the tree for an extremely long text.

Another suffix-based index is a suffix array that lists all locations of all possible suffixes of a text in lexicographical order \[42\]. A suffix array is created to address the space requirements of storing suffix trees. A binary search is simply used to locate patterns because of the sorted order of the suffixes. The worst case run time is \(O(m \log n)\) where \(m\) is the length of the pattern and \(n\) is the length of the text. Figure 2.3 shows an example of a suffix tree and a suffix array for a sample text.

**Figure 2.3:** (a) Example of a suffix tree (b) Example of a suffix array

The FM-index is a recent development in index-based searching, which is based on the Burrow-Wheeler transform (BWT) and the suffix array of the text \[24\] \[25\]. The
BWT is introduced in data compression so that characters of a string are rearranged in a manner that is compressible and reversible [10].

Searching using the FM-index involves two pointers: top and bottom pointers. The two pointers are successively computed as each character of the search pattern is processed. These two pointers indicate the range of locations a pattern appears on the BWT. The pattern exists if a valid range is defined upon processing the last character of the pattern. If two pointers indicate an empty range in the processing of a character, then the pattern does not exist and the search is terminated.

2.2.5 Summary of Algorithms

Table 2.1 summarizes the algorithms mentioned in the previous subsections. The first column lists the algorithms and their classification. The second column states the type of matching. The third and fourth columns identify worst case run time and space requirements respectively.

The longest worst case run times are the Naive, Boyer-Moore and Rabin-Karp algorithms. The Boyer-Moore exhibits a $O(nm)$ run time when the pattern exists on most part of the text that triggers a complex execution of the Good Suffix rule. The Rabin-Karp algorithm also exhibits a $O(nm)$ run time due to the verification step for existing patterns. Although both algorithms have the same worst case run time as the Naive, both have a better nominal run time that happens when the occurrence of the pattern is very rare. The two dynamic programming algorithms, Needleman-Wunsch and Smith-Waterman, also exhibit $O(nm)$ worst case run times. However, both can perform approximate string matching
Table 2.1: Summary of String Matching Algorithms

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Type</th>
<th>Worst Case Time</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Naive</td>
<td>Exact</td>
<td>O(mn)</td>
<td>none</td>
</tr>
<tr>
<td>Boyer-Moore</td>
<td>Exact</td>
<td>O(mn)</td>
<td>O(m)</td>
</tr>
<tr>
<td>Automata</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Knutt-Morris-Pratt</td>
<td>Exact</td>
<td>O(n)</td>
<td>O(m)</td>
</tr>
<tr>
<td>Aho-Corasick</td>
<td>Exact</td>
<td>O(n)</td>
<td>O(m)</td>
</tr>
<tr>
<td>Dynamic Programming</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Needleman-Wunsch</td>
<td>Approximate</td>
<td>O(mn)</td>
<td>O(mn)</td>
</tr>
<tr>
<td>Smith-Waterman</td>
<td>Approximate</td>
<td>O(mn)</td>
<td>O(mn)</td>
</tr>
<tr>
<td>Index-Based</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rabin-Karp</td>
<td>Exact</td>
<td>O(mn)</td>
<td>O(k)</td>
</tr>
<tr>
<td>Suffix Trees</td>
<td>Exact</td>
<td>O(m)</td>
<td>O(n^2)</td>
</tr>
<tr>
<td>Suffix Arrays</td>
<td>Exact</td>
<td>O(mlogn)</td>
<td>O(n)</td>
</tr>
<tr>
<td>FM-Index</td>
<td>Exact</td>
<td>O(mlogn)</td>
<td>O(n)</td>
</tr>
</tbody>
</table>

in contrast to the Naive, Boyer-Moore and Rabin-Karp algorithms.

The shortest worst case run times are suffix tree, Knutt-Morris-Pratt (KMP) and Aho-Corasick (AC) algorithms. Although searching using suffix trees is the fastest method listed on Table 2.1, the space requirements in storing the tree is a big disadvantage. Both algorithms (KMP and AC) achieve the best run times as well. However, expanding the algorithm to multiple patterns could result in huge space requirements for storing the state machines.

The next best in worst case run times are suffix arrays and the FM-index. Both execute in linear-logarithmic time and the text is preprocessed instead of the pattern. This is ideal for a variable multiple pattern search on a long text that is similar to DNA sequence mapping where reads are mapped on a reference genome.

Run time execution of the listed algorithms can be different if directly implemented in hardware. This is because component functions of the algorithms can be parallelized re-
resulting in a more efficient execution. In the next section, we discuss various hardware implementations of string matching algorithms in different applications. We will compare each approach by stating strengths and weaknesses that will help develop our own implementations.

2.3 String Matching Hardware Approaches

Implementing a string matching algorithm directly in hardware is a solution for improving the performance of software tools. Contributions of this research are hardware implementations of string matching algorithms on FPGAs. We review various string matching hardware approaches. We classify the approaches into four types similar to our algorithm classification in the previous section. The first class is using the naive approach and its derivatives. The second class is using automata or state machines. The third class is using dynamic programming. The fourth class is using indexed-based searching. We cite examples from each of the four classes.

2.3.1 Naive Approach

The naive approach checks all possible positions of a pattern on the text. The patterns are hardwired or placed as content of registers. The text is streamed using shift registers. Comparators are used to simultaneously match characters of the patterns and contents of the shift register.

An example of a naive approach is shown in Figure 2.4 [13]. This implementation is used for an intrusion detection system. The signatures of known network threats are stored
in rule blocks. Contents of the rule blocks are matched to incoming packets. The signature is divided into segments. The number of segments depend on the length of the signature and the number of characters matched per cycle. Each segment is matched at different clock cycles. A match on the first segment latches a signal enabling the next segment. Succeeding segments are compared until the entire signature is completely matched. The rule block is replicated to contain different signatures from SNORT. The incoming packet stream is concurrently distributed to rule blocks. Multiple-character matching is used to reduce the number of comparators between the threat signatures and packet stream [1]. Typically, pattern of $n$ characters requires $n^2$ comparators to cover all possible matches in an input string. Multiple-character matching reduces the number of comparators to $n$ by shifting alignment of the input stream for cases where a partial match occurs.

Figure 2.5 shows a diagram of how to improve the previous approach using pipelines and a fan-out tree [57]. The pipeline is used on the encoder. The encoder generates a code
identifying the rule blocks detecting an occurrence of a threat. The fan out tree distributes the packet stream to the rule blocks. The optimizations result in a higher operating frequency that yields a higher throughput.

![Architecture of an intrusion detection system](image)

Figure 2.5: Architecture of an intrusion detection system. Packets are fed through a fan-out tree that distributes the payload to matching engines. Matched rules are reported by the output encoder.

Content Addressable Memory (CAM) is also used for string matching in network intrusion detection as a lookup cache [52]. The lookup cache, containing threat signatures, is composed of three parts: the character match array, processing element array, and the control unit. The character match array is composed of CAMs that is matched to incoming packets. The processing element array stores the finite state machine to recognize the signature matches. The control unit manages the two arrays.

A technology known as ternary content addressable memory (TCAM) is used in naive string matching [66]. A TCAM has three possible states: 0, 1 and x (dont care) that is mainly used for considering wild cards. The research mainly used TCAMs to address composite patterns that includes regular expressions and character negations.
2.3.2 Finite State Machines

A Finite state machine (FSM) in string matching accepts an input stream of characters. State transitions are based on the character detected and the current state of the FSM. A pattern is formed as state transitions occur for every character match. State transitions are governed by a transition function. This function can be represented as a table or a state diagram. Automata could be classified into two types based on state transitions: Deterministic and Non-deterministic finite automata. Deterministic Finite Automata (DFA) strictly define only one state transition based on current state and input. On the other hand, Non-deterministic Finite Automata (NFA) define multiple state transitions based on the current state and the input resulting in multiple active states at the same time.

Figure 2.6: Aho-corasick bit-split implementation of peptide sequence matcher. Each symbol is broken down to five bits where each bit is used as input to simple FSMs. All FSMs must agree before a match is reported.

FSMs are created using the Aho-Corasick (AC) algorithm. An example of a hardware implementation is shown in Figure 2.6 [16]. The AC algorithm is used to locate peptide
sequences in a long genome. The FSM is divided into smaller FSMs using a bit-split implementation. The smaller FSMs operate on a specific bit position of a symbol. A peptide symbol is represented using five bits that break down the original FSM into five smaller ones. Tables are used to represent the new FSMs. The tables are stored in block RAMs due to their high storage density as opposed to putting them into configurable logic blocks.

Bit-split implementation of the FSMs generated from the AC algorithm is also applied to network intrusion detection [59]. Optimal splitting of the AC state machines and optimal partitioning of the patterns are investigated to obtain the best throughput. Besides implementing the AC algorithm in hardware, it is also optimized by using bitmap compression and path compression [60] [69]. Bitmap compression reduces state transition representations by using bit maps instead of pointers. Path compression reduces the number of states by simplifying nodes that are sequentially accessed without any branches. These two optimizations reduce the memory requirements of AC algorithm. The AC algorithm is further optimized by creating a smaller state machine based on a subset of rules from packet headers [3]. The less complex smaller state machine is easily accelerated resulting in higher throughput.

NFAs are used in string matching in the form of regular expressions [51]. The regular expression is received in postfix form and uses a stack data structure for processing. The NFA is created from basic logic structures that correspond to basic regular expressions. This mapping of logic structure is finished in $O(n)$ time, where $n$ is the length of the regular expression. Figure 2.7 is an example of an NFA construction from a regular expression.

To minimize hardware requirements, the AC algorithm is modified to generate
Figure 2.7: Mapping of logic structures to create a NFA of regular expression \(((a—b)^*)(cd)\).

NFAs [32]. The modified algorithm creates NFAs with a reduced number of states because of state sharing. TCAMs are also used to implement the NFAs to reduce entries of the look up table [67]. To improve routing and resource utilization, character decoders are used as matching units of the NFA hardware implementation [14]. Character decoders are matching units that save logic resources by eliminating redundant comparisons that are likely to happen. Instead of repeatedly comparing one character, only one of the output lines of the decoder are used. The decoder effectively centralizes the comparisons and sends only one-bit signals as character symbols instead of the original n-bit character. Routing is also easier because only one-bit signals are propagated. Figure 2.8 shows the hardware implementation of a character decoder illustrating its benefits over comparators.
Recent studies recommend splitting the FSM into two sections. One type of split is based on the frequency of state transitions [18]. Frequent state transitions are placed in fast memory such as a RAM. The remaining state transitions are placed in a slow memory such as an external drive. Another split is dividing the FSM into head and body [65]. The head is implemented as a DFA that is stored in a low latency memory such as a cache. The body is implemented as an NFA, allowing multiple active states and parallel pattern matching.

Figure 2.8: Example of character decoder showing reduced hardware resources and routing compared to comparators.
2.3.3 Dynamic Programming

Hardware is used to compute the cells of the similarity score matrix. The basic
hardware implementation uses the block-cyclic wavefront execution [40]. The wavefront
execution computes only the off-diagonal cells. These cells are independent of each other,
that allows parallel processing. Figure 2.9 shows a diagram of the off-diagonals computed
in parallel and their dependence on the previous off-diagonals elements.

![Wavefront Algorithm Diagram](image)

Figure 2.9: Off-diagonals of the similarity score matrix are processed in parallel
creating the wavefront algorithm.

The wavefront execution in FPGAs uses systolic arrays such that each cell of
the array processes an element of the similarity score matrix [12]. Each cell is optimized
by minimizing logic operators resulting in more processing elements and yielding higher
throughput [27]. The systolic array is generated automatically using a compiler [11]. The
compiler assigns elements of the similarity score matrix to cells of the systolic array based
on the available cells and the pattern length.

Both algorithms (NW and SW) are structured in one architecture in such a way
that any of the two algorithms is available for use depending on the user [7]. Each one is used
in different instances of the approximate string matching problem to allow flexibility while maintaining high performance. The implementation relies on the switching and sharing of interchangeable components before hardware compilation.

The Smith-Waterman algorithm is a widely researched dynamic programming algorithm implemented in hardware [43] [71] [6] [36] [46]. Each study offers a different approach to achieve faster execution. One approach modifies the SW algorithm to allow more parallelism [71] [43]. Another approach is performing the Smith-Waterman traceback while filling up the similarity score matrix in hardware [46]. Another approach is simplifying the dataflow of the processing units, saving more logic resources [36]. Another approach is using Handle-C as a platform to automatically generate hardware [6]. Handle-C is similar to C programming language that is specific for reconfigurable architectures.

The worst case run time of both dynamic programming algorithms is improved by verifying long diagonal paths that exist on the similarity score matrix. BLAST (Basic Local Alignment Search Tool) is a tool that uses a heuristic utilizing this type of method [4]. BLAST first identifies a list of words, called *seeds*, with high scores corresponding to long diagonal paths. Figure 2.10 illustrates long diagonals representing *seeds*. Sections of the patterns are matched to the seeds. Characters are appended to a section to extend and cover nearby diagonal paths until a threshold score is reached. The resulting alignment is verified using a regular Smith-Waterman or Needleman-Wunsch trace back to check the gaps where the diagonal path is involved. The resulting worst case run time is $O(m)$, where $m$ is the length of the pattern.

This method used by BLAST is further improved by a hardware implementation
Figure 2.10: **Diagonal paths on the similarity score matrix represent seeds where the BLAST heuristic starts.**

of BLAST on FPGAs. A recent research on BLAST FPGA implementations combines the three phases into one that eliminates the phases of extending the seeds and running the SW or NW trace back for gap checking [30]. The seed generation of BLAST is also optimized by dividing the entire phase into stages and implementing each stage on a pipeline [31]. The overall hardware structure of the BLAST FPGA implementation is also investigated where memory elements of the architecture such as queues and registers are tuned for higher performance [56].

### 2.3.4 Index-Based Searching

Index-Based string matching performed in hardware has not been explored much due to the nature of algorithms involving complex data structures and search functions. Concrete examples are suffix trees and suffix arrays. The complex data structure of a suffix tree is costly to implement in hardware because of too many interconnections. Performing a binary search on suffix arrays is expensive due to recursion that requires saving of previous
computations. Although complicated in its nature, studies have emerged that use hashes in heuristics.

An example of a heuristic that uses hashes is the Bloom filter [8]. A Bloom filter is a data structure that represents a set of patterns using multiple hash functions on each member pattern of a set. The Bloom filter consists of two phases. The first phase is the preprocessing phase where a Bloom filter associates a pattern to \( k \) hash values representing \( k \) hash functions. The hash functions set the bits of an \( m \)-bit vector corresponding to \( k \) hash values. The second phase is the searching phase that verifies if a pattern is a member of a set. The hash functions are applied to the pattern. If only one bit of the \( m \)-bit vector is not set, then the pattern is not a member of the set. The false positives arise because any member can set any of the bits of the \( m \)-bit vector. The hardware implementation is shown in Figure 2.11. The figure shows that several Bloom filters are created that inspect different sections and sizes of the streaming character window.

Another example of utilizing hashes is a group of seeds used as indices [48]. The index of the text is precomputed so that every seed location is identified. Seeds of the pattern are matched on the index to determine a candidate match. The candidate patterns are verified using the Smith-Waterman algorithm. The hardware implementation involves a pipelined and parallel approach of dividing the workload. The index is stored in external memory and the search patterns are streamed to the FPGA through PCIe. Multiple engines perform the Smith-Waterman to avoid bottlenecks, which results in better throughput.

Another example of index-based searching uses the trie data structure [64]. Characters of patterns are represented as nodes of a trie. Common prefixes of patterns share
Figure 2.11: Architecture of hardware string matching using Bloom filters. Symbols are streamed and several portions of streaming window are fed in parallel to N Bloom filters.

the same path from the root node. The trie, placed in memory, is traversed from the root node to the leaf nodes in searching for a pattern. The nodes of the trie are accessed based on the level of a node using specific sections of the memory address.

The two main contributions are the incoming content filter and the trie pruning. The content filter divides packet payloads into multiple substreams allowing string matching performed in a pipelined execution. A detected match in a substream stores the succeeding substream in a buffer. Other substreams from other packets are processed on other matching engines that allows parallelism. Trie pruning reduces the amount of memory required to store the patterns and allows multiple character matching in one cycle. The removed section of a trie is stored in off-chip memory. The address of the removed section is the sum of 1’s on that specific section. Figure 2.12 shows a pattern matching unit showing the block
components and their necessary connections.

Figure 2.12: Pattern matching unit of the trie. Match input is the signal that a character matches from a previous trie level. Sum of the 1’s is used to get the address of the next trie level from memory. The contents are compared to the next incoming symbols.

2.4 String Matching Approaches on GPU

Graphics processing units (GPU) are coprocessors commonly used for computer graphics computation. A typical GPU has hundreds of processor cores with a specialized pipeline for graphics processing. General purpose computation became prevalent to GPUs because of its high computing power compared to a typical CPU leading to the notion of General Purpose Graphics Processing Units (GPGPU) [63]. Programmers need knowledge on the graphics pipeline to develop applications for GPUs. OpenCL and CUDA, the dominant programming languages for GPUs, are introduced to overcome this hindrance. These languages are based on C-programming enabling developers to easily program applications
in a GPU environment. Because of high computing power and the programmability GPUs, they are widely used as alternative accelerators in a wide variety of applications.

Besides FPGAs, GPUs are used for high performance string matching. The same algorithms are implemented on these devices to achieve fast execution. The Aho-Corasick (AC) state machine is allocated efficiently to resolve problems caused by overlaps [38]. Overlaps are needed because portions of the state machine are used on different threads. Only one byte of an input stream is allocated to a GPU thread instead of one continuous stream to avoid redundant computations. Appropriate memory allocation is used in storing the state machines to enhance performance[58]. A message passing (MP) implementation of the AC algorithm is also proposed [61]. The state machine is divided in a number of GPUs and its performance is compared to a cluster of homogeneous processors.

The Smith-Waterman (SW) algorithm is implemented in GPGPUs for higher performance. The coprocessor core computes the cells of the score matrix using the CUDA programming environment [37]. The score matrix is properly divided into sub-matrices [39] [19]. Each sub-matrix is allocated to a memory space. A thread iteratively compute cells belonging to a sub-matrix. The SW algorithm is also implemented in a heterogenous CPU-GPU system. The workload is equally shared between the CPU and GPU. This is in contrast to a partition where a computationally intensive search process is allocated only to the GPU [53]. Studies using GPUs to search for patterns in long texts such as genomes using the SW algorithm are also explored [50] [17].

GPUs are used in index-based searching such as suffix trees and suffix arrays [21]. The evaluation of the two indices resulted in a better performance of suffix arrays than
suffix trees due to a fewer number of memory accesses. FM-index has also been explored in GPGPUs. The seed generation used for the FM-index is generated using GPUs [41]. The FM-index is also divided into overlapping partitions so that searching is performed by multiple processing units. Each processing unit operates on a partition for a more efficient memory access [70]. In addition to splitting the FM-index, the patterns are also organized into an efficient data structure for a more systematic search process [20]. The size of the partition is dependent on the number of GPUs and size of the text.

2.5 Conclusion

String matching algorithms evolve from the very simple naive approach to the very smart index-based approaches. The development and evolution of string matching algorithms are driven by these two main factors:

1. Higher performance: The weakness of the plain naive approach was eliminated by Knuth-Morris-Pratt (KMP) and Boyer-Moore (BM) algorithms. Recent algorithms build on the weaknesses of previous ones to improve execution time. The Aho-Corasick (AC) algorithm allows more parallelism in searching for multiple patterns concurrently on a text.

2. Application requirements: The Smith-Waterman (SW) and Needleman-Wunsch (NW) algorithms were developed in addressing string-to-string alignment which is very useful for approximate string matching. Index-based searches were developed for applications involving extremely long texts that require a pre-built index for a more
systematic and efficient searching.

Most hardware approaches in string matching are geared towards network applications specifically in intrusion detection and routing. These applications require exact string matching that utilize a naive or automata based approach. These applications require an invariant set of patterns or signatures where packets of data are streamed to check occurrences of a threat. From a string matching problem perspective, the patterns are invariant and the text that represents the stream of packets are changing all the time.

The developing field of bioinformatics led to an application where the text representing reference genomes is invariant and the patterns representing the reads are changing all the time. An approach for such a type of application is the Smith-Waterman dynamic programming algorithm. Numerous researches were conducted to implement this algorithm in hardware. However, the main problem of this method is the huge space requirement in storing the score matrix for extremely huge texts such as reference genomes.

The development of the FM-index overshadows the Smith-Waterman in searching for patterns in long texts. This is because algorithms using the FM-index are very smart and sophisticated that result in very low memory requirements and superior time complexity. Although this type of algorithm is far superior than conventional types (naive, automata-based, and dynamic programming-based), it is not explored as much as the Smith-Waterman algorithm because of complex data structures and functions such as pointers and recursions that are not appropriate for hardware implementation.
Chapter 3

Exact String Matching

3.1 Introduction

In this chapter, we introduce four hardware approaches addressing exact string matching. Our first approach is using finite state machines (FSM) based on the Aho-Corasick (AC) algorithm. Our second approach is directly encoding patterns as constants and using direct comparison of an input stream of characters. Our third approach is a combination of the first two approaches where the prefix of the pattern is implemented as a constant and the suffix is implemented as a state machine. The fourth approach uses content addressable memories (CAM) generated using an IP core generator of Xilinx.

Figure 3.1 illustrates a general block diagram implementing the approaches. It is composed of three main blocks: the fan-out tree, the matching block, and the encoder. The fan-out tree distributes a single character stream to the matching blocks to minimize the long time delays coming from the input stream. String matching is performed on the
Figure 3.1: **Block diagram of architecture implementing exact string matching implementations.**

matching block that is composed of \( m \) matching engines. The matching engine corresponds to any of the four hardware approaches. A matching engine has different numbers of input signals depending on the applied approach. It has \( n \) one-bit output signals representing \( n \) patterns placed in the engine. In total, the matching block has \( mn \) output signals passed to the encoder block. The encoder block identifies the matching patterns depending on the input signals from the matching block. Listed are the major subsections of this chapter:

1. The first section discusses automata or finite state machines (FSM) using the AC algorithm. With a fixed number of patterns, the patterns are grouped in sets. The design space is explored to determine an optimal member count that yields the best throughput per slice.

2. The second section discusses patterns encoded as constants on the FPGA. With a fixed number of patterns, the patterns are grouped in sets. The longest common prefix is identified in a group. The design space is explored to determine an optimal member count that yields the best throughput per slice.
3. The third section discusses a hybrid implementation which is a combination of constants and finite state machines. We group the patterns and implement the longest common prefix as a constant. The remaining suffixes are implemented as a FSM.

4. The fourth section discusses a CAM implementation. The effect of word count is investigated on the operating frequency of the CAM. The optimal number word count results in a maximum operating frequency that also maximizes the throughput.

5. The fifth section discusses our evaluation of the four approaches. We compare and analyze the approaches qualitatively and quantitatively.

3.2 Finite State Machines

The Aho-Corasick (AC) algorithm generates the finite state machines (FSM). The algorithm is composed of two phases, the preprocessing and searching phases. Preprocessing operates on a set of patterns and creates the FSM. The searching phase is the streaming of consecutive characters of the text on the FSM. The FSM asserts a match signal when a state transition triggers a matching pattern. The input set of \( l \) patterns is lexicographically sorted so that successive patterns of the list have long common prefixes. Grouping patterns with common prefixes minimizes the number of states of the FSM when the AC algorithm is applied to the group. FSMs, as matching engines, are successively created from the groups. A matching engine has \( n \) output signals corresponding to \( n \) patterns of the group.

Throughput, as a function of pattern length, number of patterns and group size, is shown in Figure 3.2. The occupied area, as a percentage of the FPGA area, is shown in
Figure 3.3. Throughput per slice is shown in Figure 3.4. Large group sizes result in higher throughput, smaller areas and higher throughput per slice.
Figure 3.2: Throughput versus number of strings for groups of size 4 to 16 with a pattern length of 36 and groups of size 8 to 32 for a pattern length of 16.
Figure 3.3: FPGA area versus number of strings for groups of size 4 to 16 with a pattern length of 36 and groups of size 8 to 32 for a pattern length of 16.
Figure 3.4: Throughput/slice versus number of strings for groups of size 4 to 16 with a pattern length of 36 and groups of size 8 to 32 for a pattern length of 16.
The strength of this approach relies on increasing throughput as group size increases as shown in Figure 3.2. These graphs suggest a large group is better in order to achieve a higher throughput. However, the synthesis tool imposes a significant limitation on increasing the group size. More patterns in a group exponentially increases the number of states of the FSMs. This increases the time to synthesize with place and route the FSM. The number of states as we increase the group size is shown in Figure 3.5. Note that a size of 32 patterns of 36 characters each resulted in over 800 states.

Furthermore, the advantage of decreasing area for increasing group size diminishes as shown in Figure 3.3. We can extrapolate that logic area starts to increase as more patterns are included in a group. This is not illustrated in our graphs because of the limitation of the synthesis tool. Another issue for this approach is programmability because the FPGA requires reprogramming for a different set of patterns.

![Number of States](image)

Figure 3.5: Number of FSM states versus pattern length for group sizes of 4, 8, 16, and 32 using the Aho-Corasick implementation.
3.3 Encoded Patterns

A pattern is translated into a string of 1’s and 0’s and encoded as a constant. The text is streamed using a shift register and matched to the encoded patterns. The longest common prefix in a group of patterns is identified and shared among multiple patterns to conserve logic resources. Similar to the previous section, the set of patterns is sorted lexicographically. The longest common prefix is identified in a group of $n$ patterns. The patterns are divided into the common prefix and remaining suffixes of each individual pattern. A window of the streaming characters of the text is matched on both sections, the common prefix and each individual suffix. A match on both sections indicates the pattern exists. Figure 3.6 illustrates the diagram of an existing match occurring in a group of two patterns.

![Diagram of a match occurring in a group of two patterns.](image)

Figure 3.6: A match occurs when both the common prefix and individual suffix of a pattern match.

Throughput, as a function of pattern length, number of patterns and group size, is shown in Figure 3.7. The occupied area, as a percentage of the FPGA, is shown in Figure 3.8. Throughput per slice is shown in Figure 3.9.

Figure 3.7 shows that throughput decreases as group size increases. This is because a large group size requires enabling of more distinct suffixes resulting in a high fan-out. A
Figure 3.7: Throughput versus number of strings for groups of size 1 to 16 with a pattern length of 36 and 16.
Figure 3.8: Area versus number of strings for groups of size 1 to 16 with a pattern length of 36 and 16.
Figure 3.9: Throughput/slice versus number of strings for groups of size 1 to 16 with a pattern length of 36 and 16.
circuit with high fan-out operates at a lower frequency that negatively affects the throughput. Figure 3.8 shows the area decreasing to its lowest point and starts to increase as group size increases. This is theoretically replicated by estimating the logic units of prefixes and suffixes of a fixed length. Assuming one character uses one logic unit, the total number of logic units used is counted for a fixed number of patterns with a specific length. Figure 3.10 shows the estimated logic units used by 4096 patterns with a length of 36 characters. The initial logic utility downward trend is due to the sharing of prefixes among patterns. As group size increases, the length of the longest common prefix decreases resulting in a smaller number of shared logic resources. The outcome is an increase in logic area after reaching a lowest point. Throughput per slice is inversely proportional to the logic area that shows us an optimal group size.

![Logic Units](image)

Figure 3.10: Logic units versus group size for 4096 patterns of 36 characters in length.

The advantage of using encoded patterns is low logic utility allowing more patterns to be placed in the FPGA. The use of constants also results in high operating frequency
positively affecting throughput. With high throughput, this approach is very suitable for hardware pattern matching. However, since constants are used, programmability is one of the major issues because the FPGA requires reprogramming for a different set of patterns.

### 3.4 Hybrid Approach

This approach is almost similar to encoded patterns. The longest common prefix in a group of patterns is encoded as a constant. The individual suffixes are implemented as finite state machines (FSMs). FSMs are generated using successive characters as state transitions. A match exists when the last state of the FSM is reached.

A window of the streaming characters of the text is compared to the longest common prefix. A match on the prefix latches a signal enabling the FSM corresponding to the suffixes. The FSM accepts one character at a time from the stream. If a character fails any succeeding state of the FSM, the latch is disabled and resets the suffix state machine. Figure 3.11 shows a sample diagram of the implementation.

![Figure 3.11: A match on the prefix enables a latch that enables the suffix state machine.](image)

Figure 3.12 shows decreasing throughput as group size increases. High fan-out is
the factor in decreasing the operating frequency of large group sizes. Figure 3.13 shows a fast increase of logic area as group size increases. The length of the longest common prefix decreases as group size increases, resulting in longer suffixes implemented as a FSM. FSMs require more logic than constants, which explains the fast increase in logic area. Note that when a group is composed of a single pattern, the entire pattern is implemented as a constant and is no different from an encoded pattern implementation.

The optimal group of the hybrid approach is the smallest size composed of only two patterns. With this group size, the majority of the pattern is part of the common prefix implemented as a constant. The remaining suffix implemented is a small FSM. We extrapolate that one pattern per group is more optimal eliminating the suffix implemented as an FSM. This makes the performance of this approach secondary to the encoded patterns.
Figure 3.12: Throughput versus number of strings for groups of size 1 to 16 with a pattern length of 36 and 16.
Figure 3.13: Area versus number of strings for groups of size 1 to 16 with a pattern length of 36 and 16.
Figure 3.14: Throughput/slice versus number of strings for groups of size 1 to 16 with a pattern length of 36 and 16.
3.5 Patterns on Content Addressable Memory

Content addressable memory (CAM) is a memory element consisting of $n$ words. Each word is $l$ bits long. The input of the CAM is a word of length $l$. If a word that belongs to the CAM matches the input word, then a match signal corresponding to the address of the matching word is asserted. To use the CAM in exact string matching, the text streams in a shift register and a window of length $l$ is used as input to the CAM. The output of each word of the CAM is inputted to the encoder to determine which word of the CAM matches the text window. Figure 3.15 shows a representative diagram of the CAM used for string matching.

![Diagram of CAM used for string matching](image)

Figure 3.15: The text streams in a shift register. The contents of the shift register are used as the word input of the CAM.

The matching block of Figure 3.1 is composed of multiple small CAMs or one large CAM. The operating frequency of the CAM is observed to determine the best CAM
configuration. Operating frequency of CAMs depend on the size of the CAM. Parameters that dictate the size of the CAM is the word length $l$ and the number of words $n$. The word length $l$ is dependent on the length of the pattern. Therefore, the frequency based on the number of words is only investigated.

The CAM architectures used for this design space exploration are generated using the XILINX IP core generator. The IP core generator accepts the parameters of the CAM as inputs and it automatically generates a netlist. In this experiment, the number of patterns is fixed to 2048 patterns. This pattern set is divided into groups. Each group is a CAM configured with the group size as the number of words. Each configuration has a corresponding operating frequency shown in Figure 3.16. Observe that CAMs with lower word counts operate at higher frequencies.

![Frequency vs CAM word count](image)

Figure 3.16: Frequency vs CAM word count for two word lengths with 67% of the total number of slices of the FPGA.
The main advantage of CAMs is reprogrammability. The CAMs are reloaded and the text is restreamed for a new set of patterns. A new set of pattern does not require reprogramming of the FPGA contrary to the three previous implementations. However, being a memory element, this approach suffers a low operating frequency resulting in a lower throughput.

3.6 Evaluations

Table 3.1: Throughput and area comparison of the five exact string matching implementations.

<table>
<thead>
<tr>
<th>Approach</th>
<th>Length=16</th>
<th>Length=36</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Throughput (10^9 char/sec)</td>
<td>Area (%)</td>
</tr>
<tr>
<td>CAM</td>
<td>15,250</td>
<td>82</td>
</tr>
<tr>
<td>FSM</td>
<td>19,680</td>
<td>87</td>
</tr>
<tr>
<td>Encoded pattern</td>
<td>83,810</td>
<td>42</td>
</tr>
<tr>
<td>Hybrid</td>
<td>79,249</td>
<td>41</td>
</tr>
</tbody>
</table>

Table 3.1 shows the best throughput of all approaches using two pattern lengths. Encoded pattern, using only one pattern per group, is the best hardware implementation based on throughput. It is followed by the hybrid approach using two patterns per group. The Aho-Corasick implementation is the third with a group size of 16 and the CAM-based approach performs least using 64 words per group.

The encoded pattern is superior in throughput and area compared to all implementations because of relatively simpler implementation. The hybrid approach is always secondary to the encoded patterns. Finite state machines (FSM) generated by the Aho-Corasick (AC) algorithm suffers from high logic utility and low operating frequency even
though the algorithm smartly incorporates patterns into sharing states. A CAM-based approach performs the least because of the nature of CAMs. CAMs are inherently slow compared to logic that decreases the operating frequency. The only advantage of the CAM-based approach is reprogrammability.

Figure 3.17 shows the throughput per slice of each implementation. The encoded pattern is shown as the most efficient in using logic resources. Constants yield high throughput because of high operating frequency and low logic utilization. This is not the case for the other approaches where only one factor is enhanced. Table 3.2 summarizes qualitatively the strengths of each implementation.

Figure 3.17: Throughput/slice of each exact string matching implementation for two pattern lengths.
Table 3.2: Qualitative comparison of four exact string matching implementations.

<table>
<thead>
<tr>
<th>Approach</th>
<th>Throughput</th>
<th>Area</th>
<th>Throughput per Area</th>
<th>Reprogrammable</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAM</td>
<td>Low</td>
<td>Low</td>
<td>Low</td>
<td>Yes</td>
</tr>
<tr>
<td>FSM</td>
<td>Moderate</td>
<td>Low</td>
<td>Moderate</td>
<td>No</td>
</tr>
<tr>
<td>Encoded patterns</td>
<td>High</td>
<td>High</td>
<td>High</td>
<td>No</td>
</tr>
<tr>
<td>Hybrid</td>
<td>High</td>
<td>High</td>
<td>High</td>
<td>No</td>
</tr>
</tbody>
</table>

3.7 Conclusion

In this chapter, we implemented four approaches showing that constants achieve the highest throughput per slice. We derive these important points from this chapter:

1. Performance of algorithms in hardware does not reflect worst case run times executed in software. A sophisticated algorithm such as the Aho-Corasick algorithm has the same worst case run time as the naive approach, because implementation of the latter is circumvented in hardware resulting in a higher throughput.

2. Hardware units such as block RAMs, registers, and constants are a huge factor in the implementation of matching circuits. Simplicity plays a role that enables more patterns placed in hardware resulting to higher throughput and operating frequency.

3. Reprogrammability is a main factor in an application such as DNA sequence mapping. Although constants achieve the highest throughput per slice, this implementation is not practical because it requires a complete reprogramming of the FPGA for a fresh set of reads. Reprogramming time of the FPGA must be considered in the search time because it is iteratively done for every set of reads.
Given the points stated, we develop approaches considering approximate matching in the next chapter. Our approaches avoid using hardware units such as constants and state machines implemented in logic because this requires reprogramming the FPGA. Our approaches use memory elements such as registers for easy reloading of patterns. We focus on two algorithms, a sophisticated convolution-based approach and a simple naive based approach.
Chapter 4

Approximate String Matching

4.1 Introduction

Approximate string matching is important in applications where minor errors in a pattern require detection and matching. Examples of applications using approximate string matching are spell checking and DNA sequence mapping, which is the target application of this research. In this chapter, we start addressing patterns as reads and the text as reference genome. We develop and discuss our approaches considering approximate string matching. We use key points from the previous chapter and apply those to our implementations. We learned that the naive approach, implemented in hardware using constants, has superior performance than finite state machines generated using the Aho-corasick algorithm. We learned that simple logic units are important so that more reads can be placed in hardware to achieve higher throughput. Reprogrammability must also be considered, besides simple logic, to avoid reprogramming the hardware when a massive amount of data requires
processing, which is the case in DNA sequence mapping.

The approximate string matching implementations discussed in this chapter are the naive approach and the convolution-based approach. Both approaches significantly differ structurally and cannot be generalized compared to the exact string matching implementations. Listed are the major subsections of this chapter:

1. The first section discusses the naive approach. The naive approach is the most basic string matching where the read is matched on all locations of the genome. We implement a massive parallel architecture that matches multiple pattern to multiple character streams. We explore the design space to determine the optimal number of streams that will yield the best throughput.

2. The second section discusses the convolution-based approach. Convolution based string matching uses the convolution operation of the read and genome to determine the number of mismatches between the two strings. We implement a deeply pipelined architecture performing the convolution of the two strings to obtain a high operating frequency.

3. The third section discusses our evaluation of the two hardware approaches. We compare and analyze the two approximate string matching approaches qualitatively and quantitatively.
4.2 Naive Approach

Figure 4.1 illustrates the basic hardware architecture that carries out the naive approach string comparisons. The two strings are placed on a pair of registers. In the set-up phase the read is loaded in a register. Note that we are limiting our alphabet to four symbols resulting in 2-bit characters. Subsequently, characters of the genome are streamed through a shift register and matched against the read at every cycle using comparators. Outputs of the comparators are used as inputs for a saturating adder counting mismatches between the two strings. The adder is implemented as a tree that counts the total number of zeros from all the comparator outputs. The tree is pipelined to eliminate long critical paths to avoid low operating frequency. Figure 4.2 shows a diagram of the implementation of the tree adder for a four-bit input. The adder assert a match signal when the allowed number of mismatches is reached. Note that the naive approach can also be used for exact string matching by replacing the saturating adder with an AND gate.

Figure 4.1: A read is held in a register (here shown with four characters) for the duration of the current stream. The stream, of 2-bits per character, is run through a shift register.
The basic structure of Figure 4.1 is replicated in two dimensions [49]. Figure 4.3 shows the structure in Figure 4.1 replicated four times in the horizontal dimension. The characters of the genome are streamed through four string matching structures. To increase throughput while preserving a high clock frequency, we replicate the structure in Figure 4.1 in the vertical dimension as shown in Figure 4.4. The genome is divided into four character streams as shown in Figure 4.4. Each stream is matched to the same read, forming a block. The overall structure is shown in Figure 4.5. The outputs of the comparators of one block are OR-ed to reduce the pressure on the fan-out. The occurrence of a match, a rare event, is tagged as a location in each of the streams.
This design space exploration determines the structure to maximize the throughput, defined as the number of character comparisons per second. It is the product of the operating frequency, the number of read comparisons per cycle, and the read length. We implemented structures for read lengths of 16, 24 and 36, using one, two, four and eight character streams. Saturating adders are used to support up to three mismatches. All
data reported are post synthesis, place and route. In all experiments we have attempted to maximize the FPGA area utilization as measured in percent area of the total FPGA slices.
Figure 4.6: Number of string comparisons versus read length for 1 to 8 streams with 90% of the total number of slices of the FPGA (60% register slices) using the naive implementation allowing one mismatch and three mismatches.
Figure 4.7: Operating frequency versus read length for 1 to 8 streams with 90% of the total number of slices of the FPGA (60% register slices) using the naive implementation allowing one mismatch and three mismatches.
Figure 4.8: Throughput versus read length for 1 to 8 streams with 90% of the total number of slices of the FPGA (60% register slices) using the naive implementation allowing one mismatch and three mismatches.
The number of read comparisons performed per cycle on the FPGA is shown in Figure 4.6 and the operating frequencies in Figure 4.7. From these figures we observe the following:

- The number of read comparisons on the FPGA increases as more streams are used. This observation can be explained using Figures 4.4 and 4.3. Both support four concurrent read comparisons; Figure 4.4 requires five registers while Figure 4.3 requires eight registers.

- The operating frequencies generally decrease as more streams are used due to fan-out penalties incurred when comparing a read to multiple shift registers. These penalties translate to longer clock periods before a comparison is completed.

The throughput results are shown in Figure 4.8 allowing one mismatch and three mismatches using one, two, four and eight streams. Observe that two or four streams have the highest throughput for all read lengths and numbers of mismatches. This is because of the balance of high operating frequency and high number of character comparisons per cycle. Although a single stream operates at a higher frequency than any stream count, it utilizes more logic resources, reducing the number of character comparisons per cycle that fits on the FPGA. More streams utilize less logic resources but operate at a low frequency because of high fan-out yielding a lower throughput. From this analysis, we arrive at a conclusion that two or four streams are the optimal points that maximize throughput [22].

The main strength of the naive approach is high operating frequency and reprogrammability due to usage of registers. Registers allow reprogrammability in logic without
sacrificing speed. However, with the wide use of registers in this approach, logic utilization is very high. Furthermore, the naive approach allocates the same amount of logic resources for all reads of the same length. Since all characters of a read are compared in parallel to a specific genome location, resources are wasted due to unnecessary comparisons for non-existing reads.

4.3 Convolution Based Approach

Convolution-based string matching provides the total number of matching characters of the read in all possible positions of the genome [26]. Given an alphabet 1...t, a genome T and a read P on this alphabet. The following are the steps of the algorithm:

1. For each symbol i in the alphabet 1...t, construct a binary string Ti from the genome T by replacing all occurrences of symbol i by 1 (0 otherwise). Apply the same transformation to the read P. This step produces t binary strings each for the genome and the read, one for each character in the alphabet.

2. Reverse all binary reads Pi, and obtain Ri.

3. Perform the convolution between Ti and Ri for all i, to obtain the sequence Ci.

4. Adding together all Ci provides the total number of matching characters for all positions of P in T.

Figure 4.9 illustrates an example. It is easy to verify that the resulting sum of all Ci’s corresponds to the total number of matching characters for every position of P on T.
Figure 4.9: An example of convolution-based string matching between read \( P \) and genome \( T \). The output is a vector of integers representing the total number of matches between \( P \) and \( T \), for all possible locations in \( T \).

Convolution is performed along the lines of the example in Figure 4.9. Row sequences are added to obtain the convolution of each symbol \( i \). A row sequence could be a sequence of 0’s or a shifted genome \( T \) based on the position of a 1 on the decoded read \( P \). A pipeline register is added for each row sequence to perform convolution of multiple reads per cycle and to obtain a high operating frequency. The number of reads processed per cycle is equal to the length of the read because the length is the total number of rows in the pipeline. Saturating adders are used in adding the row sequences for allowing a maximum number of mismatches.
The genome length is varied from 64 to 512 characters using read lengths of 16 and 36. This implementation achieves a frequency much higher than the naive approach shown in Figure 4.10 because all component blocks only drive a single output resulting in high operating frequency. Figure 4.11 shows a linear increase of logic area as the genome section length increases. This is because the size of the convolution block increases linearly as well. Note that implementations with a read length of 36 and genome length larger than 320 do not fit on the target FPGA.

![Frequency graph](image)

Figure 4.10: **Frequency versus genome length for read lengths of 16 and 36.**

The effective throughput is computed from the product of the difference of the genome length and the read length, multiplied by the frequency and the read length. The difference of the genome length and the read length corresponds to the effective number of read comparisons performed in a cycle. Figure 4.12 shows the throughput of the convolution
The main strength of the convolution-based approach is the high operating frequency compared to all of the string matching implementation. This is because the architecture is deeply pipelined using registers to pass output from stages. However, the problem with this approach is high logic utility due to adders and registers for each pipeline stage. Furthermore, long genome and read lengths require more pipeline stages that use up resources of the FPGA resulting in low throughput of this implementation.

Figure 4.11: **Area versus genome length for read lengths of 16 and 36.**
Figure 4.12: Throughput versus genome length for read lengths of 16 and 36.

Table 4.1: Throughput and area comparison of naive and convolution-based string matching implementations.

<table>
<thead>
<tr>
<th>Approach</th>
<th>Length=16</th>
<th></th>
<th>Length=36</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Throughput</td>
<td>Area (%)</td>
<td>Throughput</td>
<td>Area (%)</td>
</tr>
<tr>
<td>Naive</td>
<td>11,620</td>
<td>94</td>
<td>9,670</td>
<td>91</td>
</tr>
<tr>
<td>Convolution</td>
<td>4,244</td>
<td>84</td>
<td>4,199</td>
<td>92</td>
</tr>
</tbody>
</table>

4.4 Evaluations

We now compare the convolution-based approach and the naive approach. Table 4.2 shows the best throughput for both approaches for read lengths 16 and 36 allowing three mismatches. The naive approach significantly leads in throughput due to a complex hardware architecture of the convolution-based approach.

Both approaches use registers so that a new set of read and genome are easily
reloaded without reprogramming the FPGA. The biggest difference between the two approaches is the required arithmetic operations. The convolution-based approach algorithm requires a significant number of arithmetic and logic operations before the locations of the read in a genome are obtained. Furthermore, a longer read or genome requires computation linearly proportional to the number of symbols of the alphabet. In our sample application of DNA sequence mapping, a genome that is one character longer would require four additional computations corresponding to each symbol. The additional computation would require a multiple of four more registers and connections between logic and the new result is only one additional location. A disproportionate increase in logic utilization required for an outcome in the convolution-based approach is much of a disadvantage compared to the naive approach. This weakness is inherent in the algorithm and cannot be resolved easily by an efficient hardware architecture because multiple different computations are required.

Figure 4.13 shows the throughput per slice of each implementation. It shows that the naive approach is more efficient in using logic resources for character comparisons compared to the convolution-based approach. Table 4.1 summarizes qualitatively the strengths of each implementation.

**Table 4.2: Qualitative comparison of naive and convolution-based string matching implementations.**

<table>
<thead>
<tr>
<th>Approach</th>
<th>Throughput</th>
<th>Area</th>
<th>Throughput per Area</th>
<th>Reprogrammable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Naive</td>
<td>Moderate</td>
<td>High</td>
<td>Moderate</td>
<td>Yes</td>
</tr>
<tr>
<td>Convolution</td>
<td>Low</td>
<td>Low</td>
<td>Low</td>
<td>Yes</td>
</tr>
</tbody>
</table>

The main objective of this study is searching for a string matching algorithm implemented in hardware that would surpass the performance of existing software tools.
Figure 4.13: Throughput/slice of each approximate string matching implementation for two read lengths.

We compare our best hardware implementations of exact and approximate string matching to four software tools: RMAP [54], MAQ [35], and BOWTIE [34]. All three tools are index-based approaches. The first two tools use hashes and seeds to speed up the process of finding potential matches. Bowtie is one of the recent software tools based on the FM-index.

All three tools were executed on a Quad-core Intel Xeon Harpertown at 2.5 GHz with 12 MB cache and our hardware designs were implemented on Xilinx Virtex 5. Only one CPU was used to perform chip-to-chip comparison between CPU and FPGA. Note that the Harpertown is a 45 nm technology while the Xilinx Virtex 5 is a 65 nm technology device. The data set used are the human genome (3.3 billion characters) and one million reads of 16 and 36 characters in length allowing only zero mismatches for exact string matching and three mismatches for approximate string matching. Table 4.3 shows a comparison of the software tools showing their execution times for considering perfect matches and allowing
three mismatches. Table 4.4 shows the execution times of RMAP and BOWTIE compared to the FPGA with the corresponding speed up for each read length.

Table 4.3: Execution times of software tools for read lengths of 16 and 36 considering zero and three mismatches.

<table>
<thead>
<tr>
<th></th>
<th>Pattern Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>SW Tool</td>
<td></td>
</tr>
<tr>
<td>Exact Matching</td>
<td>16</td>
</tr>
<tr>
<td>RMAP</td>
<td>11,602</td>
</tr>
<tr>
<td>MAQ</td>
<td>N/A</td>
</tr>
<tr>
<td>BOWTIE</td>
<td>32.0</td>
</tr>
<tr>
<td></td>
<td>36</td>
</tr>
<tr>
<td></td>
<td>17,074</td>
</tr>
<tr>
<td></td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td>29.3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Pattern Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td></td>
</tr>
<tr>
<td>RMAP</td>
<td>11,531</td>
</tr>
<tr>
<td>MAQ</td>
<td>285,976</td>
</tr>
<tr>
<td>BOWTIE</td>
<td>89.0</td>
</tr>
<tr>
<td>36</td>
<td></td>
</tr>
<tr>
<td></td>
<td>19,931</td>
</tr>
<tr>
<td></td>
<td>32,791</td>
</tr>
<tr>
<td></td>
<td>905.0</td>
</tr>
</tbody>
</table>

Table 4.4: Comparison of software execution time to FPGA execution time.

<table>
<thead>
<tr>
<th></th>
<th>Pattern Length=16</th>
<th>Pattern Length=36</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>FPGA</td>
<td>RMAP</td>
</tr>
<tr>
<td>Exact</td>
<td>2,820</td>
<td>11,602</td>
</tr>
<tr>
<td>Approximate</td>
<td>4,510</td>
<td>11,531</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Exact</td>
<td>2,370</td>
<td>17,074</td>
</tr>
<tr>
<td>Approximate</td>
<td>12,300</td>
<td>19,931</td>
</tr>
</tbody>
</table>

The naive approach operating on the FPGA achieves some considerable speed up compared to RMAP. The main reason of the performance gain of the naive approach on hardware over RMAP is the massive parallel searching of reads on multiple streaming characters of the genome. Furthermore, since RMAP is based on indexed seeds, the tool requires a verification step to validate potential existing reads on a specific location of
the genome. Although the speed up is as high as 18x for exact matching, the speed up diminishes for approximate matching. This is due to high logic required in implementing the tree adder to detect the mismatches. Furthermore, the speed up could further disappear for longer read lengths because a fewer number of reads could fit on a target FPGA.

In comparison to new generation tools such as Bowtie, the naive approach on the FPGA offers no speed up and even a slow down. The slow down is due to a smarter approach used by Bowtie not exhibited by the naive approach. Bowtie, a search tool based on the FM-index, abandons searching for the read when certain conditions are met. These conditions are iteratively checked for each character of the read. This is not performed by the naive approach which blindly compares the entire read to a number of locations on the genome defined by the number of character streams.

4.5 Conclusion

In this chapter, we implemented two approximate string matching approaches. We showed that a massive parallel architecture using the naive approach obtains a better throughput than a sophisticated convolution-based approach. We also showed that new sequencing tools, such as Bowtie, offers an extremely better execution time than the naive approach operating on the FPGA. We can derive these important points from this chapter:

1. A smarter algorithm must be considered instead of the naive, automaton and dynamic programming-based algorithms. A good candidate algorithm is based on the FM-index similar to Bowtie implemented in hardware to obtain a significant performance gain.
2. The candidate algorithm must be carefully analyzed and remodeled to address compatibility with a hardware implementation.

3. Reprogrammability must be maintained using hardware resources of the FPGA to address huge data such as a long genome and a huge number of reads.

Given the points stated, we develop the first hardware implementation of string matching algorithm based on the FM-index. Our approach defines a new data structure also based on the FM-index so that a search process is easily implemented in hardware. We utilize block RAMs of the FPGA to store the reads and data structures required for the search.
Chapter 5

Hardware Implementation of the FM Index

5.1 Introduction

We learned from the previous chapter that approaches based on conventional algorithms such as naive, finite state machines and dynamic programming algorithms implemented in hardware do not match the performance of new sequencing tools based on the FM-index. This is because of two main reasons:

- The search of the FM-index is localized to positions that are inherently common. Locations are clustered together because of the sorting of suffixes in creating the Burrow-Wheeler transform (BWT) of the genome.

- The search of the FM-index supports early search termination of a non-existing read. Conventional string matching algorithms implemented in hardware allocate hardware
resources for the entire read which may not be required because the read may not exist on the genome.

In this chapter, we present the first FPGA-based hardware implementation of the FM-index [23]. We formulate an exact string matching approach and then derive hardware modifications that consider approximate string matching. Listed are the major subsections of this chapter:

1. The first section discusses the Burrow-Wheeler Transform (BWT). We introduce data structures based on the BWT of the genome as an alternate representative index of the genome.

2. The second section discusses a detailed operation of the search process using the data structures mentioned from the previous section. We demonstrate through an example the search process of an existing and non-existing read on the genome.

3. The third section discusses and illustrates our hardware architecture using BRAMs of the FPGA. We explore the design space of our architecture to minimize execution time. We estimate the execution time of our optimal implementation and compare to actual execution time of Bowtie. Our comparison shows an estimated speed up as high as 196x.

4. The fourth section discusses and illustrates our hardware architecture using external memory. We expand our implementation to cover approximate string matching. We compare the performance of our hardware design to Bowtie where we obtain an actual speed up as high as 70x.
5.2 Data Structures of the Burrows-Wheeler Transform

In this section we build up the required data structures for the hardware implementation of the FM-Index. Given a genome Q we denote by BWT(Q) its Burrows-Wheeler transform. The BWT of a string is generated in five steps:

1. Terminate the genome Q with a unique character: "$".
2. Generate all rotations of the genome.
3. Sort all the rotations
4. Extract the last characters of all the entries of the sorted list.
5. Join the characters in the same order they appeared on the sorted list. The newly generated genome is the BWT(Q).

Table 5.1 illustrates an example of deriving BWT(Q). Notice in Table 5.1 that characters to the left of the "$" form a suffix. A suffix array indicates the position of each possible suffix in the original string. Table 5.2 shows the suffix array representation of the genome Q in Table 1. For example at index 5 the suffix value is "$c$" and its position in the original string is 13. The suffix at index 8 is "$gctaattaggtacc$" and its position is 0 since it is the whole string.

After generating the suffix array shown in Table 5.2, we sort the characters of BWT(Q) and denote the new string as SBWT(Q) as shown in Figure 5.1. We now generate the I-table from the SBWT(Q) and C-table from the BWT(Q).
Table 5.1: Example of deriving the Burrow-Wheeler Transform of a genome. The genome is terminated by the "$" symbol.

<table>
<thead>
<tr>
<th>Original String:</th>
<th>GCTAATTAGGTACC$</th>
</tr>
</thead>
<tbody>
<tr>
<td>rotations</td>
<td>sorted rotations</td>
</tr>
<tr>
<td>gctaattaggtacc$</td>
<td>$gctaattaggtacc - C</td>
</tr>
<tr>
<td>ctaattaggtacc$g</td>
<td>aattaggtacc$gc - T</td>
</tr>
<tr>
<td>taattaggtacc$gc</td>
<td>acc$gctaattagg - T</td>
</tr>
<tr>
<td>aattaggtacc$gct</td>
<td>aggtacc$gctaatt - T</td>
</tr>
<tr>
<td>attaggtacc$gcta</td>
<td>attaggtacc$gct - A</td>
</tr>
<tr>
<td>ttaggtaacc$gcta</td>
<td>c$gctaattaggta - C</td>
</tr>
<tr>
<td>taggtacc$gctaat</td>
<td>cc$gctaattagg - A</td>
</tr>
<tr>
<td>aggtacc$gctaat</td>
<td>ctaattaggtacc$ - G</td>
</tr>
<tr>
<td>ggtacc$gctaatta</td>
<td>gctaattaggtacc - $</td>
</tr>
<tr>
<td>gtacc$gctaatag</td>
<td>ggtacc$gctaatt - A</td>
</tr>
<tr>
<td>tacc$gctaattagg</td>
<td>gtacc$gctaatta - G</td>
</tr>
<tr>
<td>acc$gctaattagggt</td>
<td>taattaggtacc$g - C</td>
</tr>
<tr>
<td>cc$gctaattaggta</td>
<td>tacc$gctaattag - G</td>
</tr>
<tr>
<td>c$gctaattaggtac</td>
<td>taggtacc$gctaa - T</td>
</tr>
<tr>
<td>$gctaattaggtacc</td>
<td>ttaggtaacc$gcta - A</td>
</tr>
</tbody>
</table>

Burrow-Wheeler Transform: CTTTACAG$AGCGTA

- **Table-I** - For every element x of the alphabet of Q, Table-I[x] = index of its first occurrence in SBWT(Q). For example, in Figure 5.1, A, C, G, and T appear at index 1, 5, 8, and 11 in SBWT(Q).

- **Table-C** - For each index n in BWT(Q) and for each character x in the alphabet Table-C[n,x] = number of occurrences of x in BWT(Q) in the range [0, n-1]. As an example, consider index n = 10 in the C-Table in Table 5.4, column A has a value of 3 because there are three occurrences of A in the range n = 0 to n = 9.
Table 5.2: Example of deriving the suffix array of a genome.

<table>
<thead>
<tr>
<th>Index</th>
<th>Sorted Suffixes</th>
<th>Suffix Array</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$</td>
<td>14</td>
</tr>
<tr>
<td>1</td>
<td>aattaggtacc$</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>acc$</td>
<td>11</td>
</tr>
<tr>
<td>3</td>
<td>aggtacc$</td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td>attaggtacc$</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>c$</td>
<td>13</td>
</tr>
<tr>
<td>6</td>
<td>cc$</td>
<td>12</td>
</tr>
<tr>
<td>7</td>
<td>ctaattaggtacc$</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>gtaattaggtacc$</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>ggtacc$</td>
<td>8</td>
</tr>
<tr>
<td>10</td>
<td>gtacc$</td>
<td>9</td>
</tr>
<tr>
<td>11</td>
<td>taattaggtacc$</td>
<td>2</td>
</tr>
<tr>
<td>12</td>
<td>tacc$</td>
<td>10</td>
</tr>
<tr>
<td>13</td>
<td>taggtacc$</td>
<td>6</td>
</tr>
<tr>
<td>14</td>
<td>ttaggtacc$</td>
<td>5</td>
</tr>
</tbody>
</table>

Figure 5.1: Example of sorting the BWT(Q).

Table 5.3: I-Table stores the first occurrence of each character on the sorted BWT(Q).

<table>
<thead>
<tr>
<th>I Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>A C G T</td>
</tr>
<tr>
<td>1 5 8 11</td>
</tr>
</tbody>
</table>

5.3 Pattern Searching Using the FM-Index in Hardware

The FM-index is a pattern searching technique operating on the Burrow-Wheeler Transform (BWT). The FM-index searching technique uses two pointers: top and bottom.
Table 5.4: The C-table stores the count of each character on a previous location. The leftmost column of the table shows the indices of the suffix array.

<table>
<thead>
<tr>
<th>Index</th>
<th>BWT(Q)</th>
<th>A</th>
<th>C</th>
<th>G</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>C</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>T</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>T</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>T</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>A</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>C</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>A</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td>G</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>8</td>
<td>$</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>9</td>
<td>A</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>10</td>
<td>G</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>11</td>
<td>C</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>12</td>
<td>G</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>13</td>
<td>T</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>14</td>
<td>A</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>15</td>
<td>Total</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

The top points the index of the suffix array identifying the first location of a read occurring in the genome. The bottom points to the index that limits the locations where a read exists.

A read does not exist on the genome if the bottom points to an index that is less than or equal to an index pointed to by the top.

### 5.3.1 Searching and Locating a Pattern

We initialize the top and bottom pointers to the first and last indices of the C-table respectively. One character is processed at a time beginning with the last character of the read. The top and bottom pointers move to a different suffix array index depending on the processing of the current character processed and the current index where the top and bottom are pointing. To compute for the new location of the pointers, we follow Equation 82.
5.1 and 5.2 for the top and bottom pointer respectively.

\[
Top_{\text{new}} = C(n, Top_{\text{current}}) + I(n) \tag{5.1}
\]

\[
Bottom_{\text{new}} = C(n, Bottom_{\text{current}}) + I(n) \tag{5.2}
\]

Figure 5.2 shows an example of searching the read "TAGG" on the example string in Table 5.4. We initialize the top and bottom pointers to 0 and 15 respectively. We begin with the last character, G, of the read. We then apply Equation 1 four times corresponding to four characters of the read. After the fourth iteration, the top and bottom pointers are at index 13 and 14 respectively. Since the index of the top pointer is less than the index of the bottom pointer, the read TAGG occurs on the string.

**Text:** GCTAATTAGGTACC

**Pattern:** TAGG

<table>
<thead>
<tr>
<th>1st iteration: n = G</th>
<th>3rd iteration: n = A</th>
</tr>
</thead>
<tbody>
<tr>
<td>Top_{\text{new}} = C_G(Top_{\text{current}}) + I(G)</td>
<td>Top_{\text{new}} = C_A(Top_{\text{current}}) + I(A)</td>
</tr>
<tr>
<td>= 0 + 8 = 8</td>
<td>= 2 + 1 = 3</td>
</tr>
<tr>
<td>Bot_{\text{new}} = C_G(Bot_{\text{current}}) + I(G)</td>
<td>Bot_{\text{new}} = C_A(Bot_{\text{current}}) + I(A)</td>
</tr>
<tr>
<td>= 3 + 8 = 11</td>
<td>= 3 + 1 = 4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>2nd iteration: n = G</th>
<th>4th iteration: n = T</th>
</tr>
</thead>
<tbody>
<tr>
<td>Top_{\text{new}} = C_G(Top_{\text{current}}) + I(G)</td>
<td>Top_{\text{new}} = C_T(Top_{\text{current}}) + I(T)</td>
</tr>
<tr>
<td>= 1 + 8 = 9</td>
<td>= 2 + 11 = 13</td>
</tr>
<tr>
<td>Bot_{\text{new}} = C_G(Bot_{\text{current}}) + I(G)</td>
<td>Bot_{\text{new}} = C_T(Bot_{\text{current}}) + I(T)</td>
</tr>
<tr>
<td>= 2 + 8 = 10</td>
<td>= 3 + 11 = 14</td>
</tr>
</tbody>
</table>

Figure 5.2: Example of searching the read "TAGG" on the string "GCTAATTAGGTACC" using the FM-index. After the 4th iteration the read is found because the index of the top pointer is less than the bottom pointer.

Three methods are described to identify the location of the read once its existence is determined:
• Use characters from BWT(Q) of the genome instead of the read to trace the end of
the genome using Equation 5.1. The total number of iterations of using Equation 5.1
is the location where the read exists on the genome. No additional storage is required
for this method but it has a disadvantage of using a large number of iterations before
the location is found.

• Store the suffix array elements that indicate the positions of the suffixes. After deter-
mining that the read exists, the element corresponding to the suffix array index pointed
to by the top is the location of the read. The main advantage is the immediate avail-
ability of the locations. However, the entire suffix array must be stored and it is
potentially huge for an extremely long genome.

• Store only samples of the suffix array and trace until a sampled suffix array element is
reached. The number of iterations of using Equation 5.1 and the suffix array element
is added to compute the location of the read. This method combines the advantages of
both approaches and minimizes the effect of the disadvantages. We used this approach
in our initial implementation.

To continue our example, we locate the position of the read in the genome. Figure
5.3 shows the sampled suffix array elements and how the read "TAGG" is located on the
sample string. After identifying that the read exists, we check if the top points to a sampled
suffix array element. If not, we utilize Equation 5.1 using the BWT character as the
symbol for the C and I tables. We also increment a counter to track the count of using the
equation. We continuously use Equation 5.1 and increment the counter until the top points
to a sampled suffix array element. Figure 5.3 shows that the read "TAGG" is located after the end of the second iteration.

![suffix array example](image)

<table>
<thead>
<tr>
<th>BWT</th>
<th>Suffix array sample</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>C</td>
</tr>
<tr>
<td>1</td>
<td>T</td>
</tr>
<tr>
<td>2</td>
<td>T</td>
</tr>
<tr>
<td>3</td>
<td>T</td>
</tr>
<tr>
<td>4</td>
<td>A</td>
</tr>
<tr>
<td>5</td>
<td>A</td>
</tr>
<tr>
<td>6</td>
<td>C</td>
</tr>
<tr>
<td>7</td>
<td>G</td>
</tr>
<tr>
<td>8</td>
<td>$</td>
</tr>
<tr>
<td>9</td>
<td>A</td>
</tr>
<tr>
<td>10</td>
<td>G</td>
</tr>
<tr>
<td>11</td>
<td>C</td>
</tr>
<tr>
<td>12</td>
<td>G</td>
</tr>
<tr>
<td>13</td>
<td>T</td>
</tr>
<tr>
<td>14</td>
<td>A</td>
</tr>
</tbody>
</table>

Figure 5.3: Example of locating the read "TAGG" on the string "GCTAATTAG-GTACC" using the third approach. After the 2nd iteration the read is located on position 6 of the genome.

Suppose we search for the read "CCGA" on the example string shown in Table 5.4. Starting from the last character, the pointers move until the second character where the pointers index the same suffix array element. This read search is shown in Figure 5.4.

In this example, we show that the FM-index abandons the search when it determines the
read does not exist from the initial processing of characters.

**Text:** GCTAATTAGGTACC  
**Pattern:** CCGA

1st iteration: $n = A$  
$\text{Top}_{\text{new}} = \text{C}_A(\text{Top}_{\text{current}}) + I(G)$  
$= 0 + 1 = 1$  
$\text{Bot}_{\text{new}} = \text{C}_G(\text{Bot}_{\text{current}}) + I(G)$  
$= 4 + 1 = 5$

2nd iteration: $n = G$  
$\text{Top}_{\text{new}} = \text{C}_A(\text{Top}_{\text{current}}) + I(A)$  
$= 0 + 8 = 8$  
$\text{Bot}_{\text{new}} = \text{C}_A(\text{Bot}_{\text{current}}) + I(A)$  
$= 0 + 8 = 8$

Figure 5.4: Example of searching the read ”CCGA” on the string ”GCTAATTAGGTACC” using the FM-index. After the 2nd iteration the read is not found because the top and bottom pointers index the same suffix array location.
5.4 Hardware FM-index using Block RAMs

This section describes the hardware implementation of the FM-index using Block RAMs (BRAM) available on the FPGA. The objective of this implementation is proof that the concept of exact string matching using the FM-index is realizable in hardware. The first subsection illustrates the hardware architecture. The second subsection discusses our evaluations and comparisons to the hardware naive approach and the Bowtie software tool.

5.4.1 Architecture

The read is placed on a shift register, where the last character is fed as input to the architecture shown in Figure 5.5. A section of the genome is placed on the FPGA using internal BRAMs. Two memory banks are used to store the C and I tables. The top and bottom pointers address the C-table. The memory bank outputs of the C-table are four values corresponding to columns of A, C, G, and T. These values are inputs to a multiplexor and the current character symbol, \( n \), is used as the selector. The symbol \( n \) also addresses the I-table.

The output of the I-table and the multiplexors of the C-table are added to compute the new values of the pointers which are used as addresses of the next iteration. An additional multiplexor is used for initializing the pointers as addresses of the C-table. Figure 5.5 shows the block diagram showing the architecture using the FM-index.

The number of bits required to represent the C-table is the product of the genome size (in number of characters), the logarithm of the genome size, and the size of the alphabet. The genome size is the number of rows of the C-table, the size of the alphabet is the number
of columns, and the logarithm of the genome section length is the number of bits per element of each column entry of the C-table. The number of bits required to implement the I-table is a product of the logarithm of the genome size and the size of the alphabet.

Figure 5.6 shows the modified block diagram of Figure 5.5 that supports finding the location of an existing read. Additional storages are required for the sampled suffix array elements, flag bits and the BWT characters of the genome section. The flags indicate a sampled character. The BWT(Q) characters address the I-table and select a column of the C-table when the occurrence of the read is established. Lastly, a finite state machine is required to facilitate the transfer of data from different storage structures.

Figure 5.7 shows the diagram of the controller. It has four states. The Init state initializes the top and bottom pointers. This state asserts the init signal used as multiplexor select for initialization. The Search phase is where the pointers move in finding the read on the genome. A counter is set on the Search phase to execute a number of times equal to
the length of the read. If the read does not occur on the genome, the state machine goes back to the Init phase. If the bottom pointer addresses a location that is greater than the top pointer at the end of the count, then the next state is the Check Flag indicating that the read appears on the genome. This state asserts the loc-en signal so that the C and I tables use the BWT(Q) characters as the multiplexor instead of read characters. The Check flag state checks if a character position is a sampled suffix array element. If the position
is sampled, then it returns to the \textit{Init} phase; Otherwise, it moves to the \textit{Increment state}. The \textit{Increment state} adds one to a counter that indicates the number of steps the trace occurred. After the \textit{Increment state}, it returns again to the \textit{Check Flag state}.

![Controller diagram]

Figure 5.7: \textbf{Controller used for architecture.}

### 5.4.2 Evaluation

We perform experiments to determine advantages of splitting one large genome, with a single set of C and I tables, into multiple genome sections with a set of C and I tables for each section. In this experiment, we start with a genome of 262,144 characters corresponding to 16 modules of 16K characters each, or 32 modules of 8K characters, or 64 modules of 4K characters etc. We sample the suffix array every 32 elements and the read length is 36 characters. A genome section corresponds to one hardware module. The length of the genome section in one module ranges from 1K to 16K characters.

We synthesized with place and route our implementations on a Xilinx Virtex 6
(XC6VLX760) FPGA. Figure 5.8 shows the operating frequency and slices utilized (as % of total slices) for increasing the length of the genome section per module. Increasing the section size, from 1K to 16K characters, decreases the operating frequency by 20% due to larger adder size and larger circuit modules. The percentage of slices decreases from 25% to 2.5% due to more logic requirements for processing multiple FM-indices compared to a single unit for one long genome section. Looking at both graphs, it is better to implement the FM-index on an FPGA using longer genome lengths.

Software tools using the FM-index algorithm search the read on the entire BWT of the genome to be more efficient. To achieve the same efficiency, we fit a longer genome on the FPGA resulting in a higher effective throughput. We also attempt to maximize the clock frequency by pipelining the modules.

Table 5.5: **Resource utilization and Frequency of FM-index**

<table>
<thead>
<tr>
<th>Module size</th>
<th>BRAM-18 Utilization (%)</th>
<th>Slices (%)</th>
<th>Freq (MHz)</th>
<th>Throughput (10^{12} CC/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>FM-1K</td>
<td>1430 (99.0%)</td>
<td>35</td>
<td>251</td>
<td>84.31</td>
</tr>
<tr>
<td>FM-2K</td>
<td>1366 (94.8%)</td>
<td>19</td>
<td>244</td>
<td>81.10</td>
</tr>
<tr>
<td>FM-4K</td>
<td>1422 (98.7%)</td>
<td>13</td>
<td>241</td>
<td>112.40</td>
</tr>
<tr>
<td>FM-8K</td>
<td>1422 (98.7%)</td>
<td>8</td>
<td>213</td>
<td>100.13</td>
</tr>
<tr>
<td>FM-16K</td>
<td>1432 (99.4%)</td>
<td>4</td>
<td>201</td>
<td>88.5</td>
</tr>
</tbody>
</table>

We experimentally evaluate the tradeoffs between smaller genome sections having high clock frequencies and larger genome sections with low clock frequencies. Table 5.5 shows the resource utilization, operating frequency and throughput of the five FM-index implementations with a read length of 36 on the Virtex 6 LX760 having a total of 1,440 BRAM-18. We used a mix of BRAM-18 and BRAM-36 but are reporting the total number
Figure 5.8: Operating frequency and area versus genome section length. Note that the frequency decreases by only 20% as the genome section size increases from 1K to 16K. Logic area decreases exponentially as we increase the genome subsection length.

Partitioning the genome across multiple modules requires that segments of the genome be replicated in contiguous modules. This introduces some area overhead as shown in Table 5.6 for a read length of 36. Longer overlaps occur for longer read lengths because of longer repeated start and end sections of the genome. These overlaps are deducted from
the total genome characters placed on the FPGA, which reduces the effective length of the genome.

**Table 5.6: Overhead due to genome overlap for search string length 36.**

<table>
<thead>
<tr>
<th>Module Size</th>
<th>Total Characters</th>
<th>Overlap Characters</th>
<th>Overlap Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>FM-1K</td>
<td>360448</td>
<td>24570</td>
<td>6.8%</td>
</tr>
<tr>
<td>FM-2K</td>
<td>344064</td>
<td>11690</td>
<td>3.4%</td>
</tr>
<tr>
<td>FM-4K</td>
<td>475136</td>
<td>8750</td>
<td>1.8%</td>
</tr>
<tr>
<td>FM-8K</td>
<td>475136</td>
<td>3990</td>
<td>0.84%</td>
</tr>
<tr>
<td>FM-16K</td>
<td>442368</td>
<td>1820</td>
<td>0.41%</td>
</tr>
</tbody>
</table>

We compare the FM-index hardware implementation to a naive approach similar to the previous chapter implemented for exact matching. We use the naive approach as reference because of its reprogrammability having the highest throughput. As a review of the naive approach, we store a set of **reads** in registers of the FPGA. The genome is streamed into shift registers and concurrently compared to the **reads** in registers. A signal is asserted when a match occurs. To maximize parallelism, the genome is divided into multiple concurrent streams. This is implemented on a Xilinx Virtex 6 LX760 FPGA with a **read** length of 36. We utilized 92% of logic slices so that we could place the most number of **reads**.

**Table 5.7: Character comparisons (CC) per second for FM-index and naive implementations.**

<table>
<thead>
<tr>
<th></th>
<th>Freq (MHz)</th>
<th>CC/cycle</th>
<th>Throughput (10^{12} CC/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>FM-8K</td>
<td>241</td>
<td>466,386</td>
<td>112.40</td>
</tr>
<tr>
<td>Naive</td>
<td>342</td>
<td>200,880</td>
<td>68.75</td>
</tr>
</tbody>
</table>

Table 5.7 shows the total number of character comparisons performed for both
approaches. For the FM-index, we use the FM-4K implementation because of its highest effective throughput. We compute the throughput of the FM-index as the product of the genome size and the operating clock frequency because we are effectively searching for a read on an entire genome section as we process each character of the read. Throughput of the FM-index is almost double of the naive approach.

We mentioned that each character of the read is effectively searched on the entire genome section. As each character is processed sequentially, a possibility of a mismatch can occur per character resulting in a search termination. Therefore, given a read of length \( p \), unless there is match, the number of character comparisons is \( c \times p \). In the previous section, we derived the raw throughput in terms of character comparisons the hardware can deliver every cycle. However, this throughput does not reflect the ability of the algorithm to abandon the search and hence the expected run time. It is proven that the expected shared prefix between any two substrings in a random genome, \( e \), is \( e = \log(n)/E \), where \( E \) is the entropy of the random source generating the genome and \( n \) is the genome size [5]. Using the DNA alphabet and the genome section length, the expected length of the common prefix ranges from 5 (\( n = 1\text{K} \)) to 7 (\( n = 16\text{K} \)), as shown in Table 5.8. One can expect than very soon after this 5-7 matches to see a mismatch, and therefore abandon the search.

We compare our FM-index hardware implementation to the Bowtie software tool used for mapping DNA sequences to a reference genome. We execute the software tool using only one core of a Quad-core Intel Xeon at 2.5GHz with 12 MB cache. We measure the execution time of searching one thousand DNA sequences with lengths of 36, 72 and 108 on a section of the E-coli genome with a length of 490,000 characters. We compare the total
Table 5.8: \textbf{Expected length of shared suffixes between two substrings for a specific genome length.}

<table>
<thead>
<tr>
<th>Text section Length</th>
<th>Expected Length of Shared Suffix</th>
</tr>
</thead>
<tbody>
<tr>
<td>1K</td>
<td>5</td>
</tr>
<tr>
<td>2K</td>
<td>5.5</td>
</tr>
<tr>
<td>4K</td>
<td>6</td>
</tr>
<tr>
<td>8K</td>
<td>6.5</td>
</tr>
<tr>
<td>16K</td>
<td>7</td>
</tr>
</tbody>
</table>

execution times of the software tool to the execution time of the hardware implementation based on throughput on Table 5.9.

Table 5.9: \textbf{Execution times of Bowtie and FPGA Implementation with the number of matching DNA sequences on the E-coli genome in percentages.}

<table>
<thead>
<tr>
<th>Pattern Length</th>
<th>Software Run Times (s)</th>
<th>FPGA Run Times (s)</th>
<th>Matching Patterns (%)</th>
<th>Speed Up</th>
</tr>
</thead>
<tbody>
<tr>
<td>36</td>
<td>0.0115</td>
<td>0.0000586</td>
<td>6</td>
<td>196</td>
</tr>
<tr>
<td>72</td>
<td>0.0075</td>
<td>0.0000602</td>
<td>3</td>
<td>124</td>
</tr>
<tr>
<td>108</td>
<td>0.0075</td>
<td>0.0000565</td>
<td>1</td>
<td>133</td>
</tr>
</tbody>
</table>

We computed separately the execution times of non-existing and existing \textit{reads}. The cost in cycles in searching for a non-existing \textit{read} is twice the expected shared suffix length for a 4K module shown in Table 5.8 to be conservative in the cost before the search is abandoned. For existing \textit{reads}, we use \textit{read} length as the cost in cycles for searching the \textit{read}. Table 5.9 shows that the FPGA achieved a speed up as high as 196 compared to software execution times.

One of the limitations of the FM-index hardware implementation using BRAMs is the huge dependence on available memory. For extremely large genomes such as the human genome, BRAMs of the FPGA are not enough to hold all the sections of the genome. Besides
dividing the genome into sections, we need to cycle the different sections on the FPGA to cover the entire genome. To avoid such scenarios, a large number of BRAMs is required, which is not supported by FPGAs.

5.5 Hardware FM-index using External Memory

This section describes the hardware implementation of the FM-index using external memory. We introduce FHAST (FPGA Hardware Accelerated Sequencing-matching Tool) to address the main limitation of the previous implementation of memory requirements. Using external memory entails long access times resulting in poor performance. To address this problem, we implement a multi-threaded approach to hide memory latency. Each thread represents one read. We issue multiple memory requests for multiple threads and track the order of returning data using BRAMs as queues. The first subsection illustrates our hardware architecture specific for exact string matching. We extend our hardware architecture to cover approximate string matching in the second subsection. The third subsection discusses our evaluations and comparison to software execution time of Bowtie.

5.5.1 Exact Matching Architecture

Figure 5.9 shows the block diagram implementing exact matching using external memory. The implementation consists of five main blocks: the fetch, update, send, receive, and locate blocks. Each block consists of queues used to maintain the order of thread requests while performing other tasks. The C-table and list of reads are located on external memory. The I-table is placed on LUTs of the FPGA.
Figure 5.9: Block diagram of exact string matching algorithm using external memory.

The fetch block requests external memory in accessing reads and generates an ID for each read. The update block inserts the fetched reads to the send block. The update block determines if a read requires more iterations or if the read matches or mismatches. The update block forwards the read to the send block if the read needs more iteration. The send block issues addresses to access the C-table for the top and bottom pointers. The I-table is also accessed simultaneously using the current character of the read. As addresses are issued to external memory, the send block passes the ID, data from the I-table and the iteration count of the read to the queue of the receive block. When data returns from external memory, the queues are popped and the new pointers are computed and passed to the update block for further processing.

The send block continuously issues addresses of different reads to external memory and read information to the receive block until queues of the external memory and the receive block are full. This is the multithreaded search where multiple reads are waiting
in queues for memory while other reads are processed. When memory returns data, the receive block performs the computation and passes it to the update block.

The update block decides if a processing of a read terminates based on two conditions: 1) The read exists on the genome. This happens when the two pointers, after processing the last character of the read, represent a valid range. The read is passed to the locate block to identify the position of the match. 2) A read does not exist on the genome when data returned from memory represent an empty range of locations. The read is then discarded. In both cases, a new read from the fetch block is introduced to keep the queues full.

The performance of the hardware just discussed highly depends on the number of external memory requests. To reduce this number, memory addresses are precalculated for all character combinations up to a specific length such that each combination of characters represents a range on the C-table. Instead of initializing the address to the first and last rows of the C-table as indicated in the algorithm, we instead initialize the pointers to the precalculated values. We store the precalculated values in a block RAM and use the last 1 characters to access the precalculated values. Figure 5.10 shows a detailed update block including precalculation.

5.5.2 Approximate Matching Architecture

For minimal change to the structure of the exact matching architecture, we use multiple exact matching engines to expand the capability of approximate matching. For every n allowed mismatch, we use n + 1 exact matching engines. Figure 5.11 is a block
Figure 5.10: **Update block including a block RAM for precalculated addresses.**

A diagram showing the connection between exact matching engines.

Figure 5.11: **Block diagram of approximate matching architecture using n exact string matching architectures.**

Engine 0 performs exact string matching and requests reads from external memory. When a read on iteration k fails on Engine 0, it is passed to Engine 1. The important information passed to the next engine is: the read, the iteration count reduced by one, and...
address of pointers on iteration k-1. The data passed are inputted to a replace block that follows a heuristic enabling mismatched search. The heuristic and the replace block are discussed in the next section. Reads passing Engine 1 register as a matching read with one mismatching character. A failing read on Engine 1 is passed to Engine 2 to detect reads with two mismatching characters. Reads passing the three matching engines are all passed to the locate block that determines the location of the read on the genome.

The replace block executes the heuristic allowing approximate matching. The heuristic creates three copies of the failing read on Engine 0 with each copy having the same accepted ID. The failing character of each copy is replaced by the other three characters of the alphabet. Each copy becomes a new thread and is inserted in the queue of the next engine. Figure 5.12 shows an example of the heuristic showing the replacement of the failing character by other characters.

![Figure 5.12: A failing character of the read creates three copies of the read with the failing character replaced.](image)

A flag f is set for each read copy as each is inserted to the queue of the update
block of Engine 1. The flag indicates read copy that is recently inserted to Engine 1. If the copy fails on its first iteration on the new engine, the copy is discarded and not passed to a succeeding engine. If the copy proceeds to the next iteration on Engine 1, then the flag f of the copy is reset. If the copy fails on any succeeding iterations, the copy is passed to Engine 2 where new copies are created again.

Figure 5.13: **Update block including the replace block for approximate matching.**

Figure 5.13 shows the block diagram of the replace block inserted in the update block of Engine 1. Engine 1 accepts reads from the replace block instead of the fetch block. The update block of Engine 1 selects reads from the replace block when processing of a read terminates.

Reads that exit from the matching engines are passed to a locate block to search for the exact position of the *read* on the genome. The data passed for each read are the *read* ID and the last pointers returned from memory. The architecture of the locate block is similar to send and receive blocks. Figure 5.14 shows the block diagram of the locate block.
consisting of send queues for memory requests and waiting queues for data returned from
memory.

Figure 5.14: Locate block have send and receive queues similar to send and receive
blocks for finding the location of a read from the suffix array residing in main
memory.

The locate block sends the top pointer as an address to the suffix array placed in
external memory. External memory returns the location that is written to the output file.
If a read exists at multiple locations, we send multiple addresses to memory for the required
locations.

5.5.3 Evaluations

Figure 5.15 illustrates the role of the hardware as coprocessor in searching for
reads on the genome. The software performs memory allocation for reading the C tables,
the suffix arrays, the reads and writing the results to external memory. After allocating
memory and setting up the coprocessor registers, the host CPU calls the coprocessor to
Figure 5.15: **Role of software is mainly on memory allocation and reporting. Hardware performs the search algorithm.**

perform the search algorithm that writes matching *reads* to memory allocated by software. Software then writes results to an output file.

We conduct our experiments on the ConveyHC-1 hybrid core computing system composed of a dual core Intel Xeon processor running at 2.13 GHz as the host processor and four Xilinx Virtex 5 FPGAs as coprocessor. All processors, both host processor and FPGA coprocessors, have one shared cache coherent memory. Each FPGA has 16 memory channels from eight memory controllers. This memory subsystem supports a peak bandwidth of 80 GB/s. We implemented designs for a *read* length of 101 characters that supports 0, 1, and 2 mismatches using only one FPGA in the coprocessor. The design is synthesized with place and route on a Xilinx Virtex 5 (XC5VLX330) FPGA that occupies 23,923 slices or 46% of the FPGA. We set the frequency to 150 MHz that is the maximum operating frequency of the memory controllers of the Convey-HC1.

We compare our results to the Bowtie software tool used for mapping DNA se-
quences. We execute the Bowtie software tool using two systems whose specifications are shown in Table 5.10. CPU 1 is the host CPU on the HC-1. We measure the execution time of searching 18 million unique reads with 101 characters in length on Chromosome 14 of the human genome having a length of 107 million characters.

Table 5.10: Specifications of CPU running Bowtie.

<table>
<thead>
<tr>
<th></th>
<th>CPU1</th>
<th>CPU2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor type</td>
<td>Xeon L540B</td>
<td>Xeon E5520</td>
</tr>
<tr>
<td># of cores</td>
<td>2 dual cores</td>
<td>2 quad cores</td>
</tr>
<tr>
<td>Memory Size</td>
<td>192 GB</td>
<td>24 GB</td>
</tr>
<tr>
<td>Cache size</td>
<td>6 MB</td>
<td>8 MB</td>
</tr>
<tr>
<td>Frequency</td>
<td>2.13 GHz</td>
<td>2.27 GHz</td>
</tr>
</tbody>
</table>

Table 5.11 shows the execution time in seconds of FHAST and Bowtie running in both systems in detecting the reads having zero, one and two mismatches. The table shows longer execution time of Bowtie running in both CPUs compared to FHAST. Notice that there is a significant difference in the execution time of FHAST between searching exact and approximate matches. This is because the reverse of a read is required in the search to detect mismatches. This additional copy of a read represents an additional thread that uses up bandwidth and lengthens the execution time. Also notice that with FHAST, there is no significant difference in execution time between searching of reads with one and two mismatches. This small difference in execution time is due to simultaneous searching of reads in the three engines concurrently. The execution time for Bowtie increases significantly as more mismatches are allowed. Figure 5.16 shows the speed up of FHAST over Bowtie for the two CPUs. Observe that the highest speed up is achieved in detecting two mismatches where Bowtie execution time is the longest. By masking memory latency, the customized
multithreading approach allows us to achieve better than 70x speedup on a 150 MHz FPGA over large CPUs.

Table 5.11: Execution times of Bowtie and FPGA Implementation with the number of matching DNA sequences on the E-coli genome in percentages.

<table>
<thead>
<tr>
<th>Mismatch</th>
<th>FHAST</th>
<th>Bowtie</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CPU1</td>
<td>CPU2</td>
</tr>
<tr>
<td>0</td>
<td>55.43</td>
<td>715</td>
</tr>
<tr>
<td>1</td>
<td>71.17</td>
<td>1924</td>
</tr>
<tr>
<td>2</td>
<td>73.25</td>
<td>5410</td>
</tr>
</tbody>
</table>

Figure 5.16: Speed up of FHAST and Bowtie for exact matches, one and two mismatches.
5.6 Conclusion

In this chapter, we implemented the first hardware implementation of the FM-Index using FPGAs. We showed and demonstrated a new representation of the FM-Index that is easily referenced for hardware implementation. We demonstrated using external memory in storing the data structure resulting in the development of FHAST, which could be used as a drop-in replacement of Bowtie. Our evaluations showed an actual performance gain of 70x compared to Bowtie using the Convey super computing system using FPGA coprocessors. We derive these important points from the chapter:

1. Implementing an algorithm in hardware does not necessarily involve a direct translation of the operational codes of the algorithm to hardware functional units. A string matching algorithm using the FM-Index is analyzed and reassessed to develop a derivative data structure representation suitable for hardware implementation.

2. The amount of data that needs processing is a major factor in developing a hardware architecture that emulates an algorithm. Massive amounts of data necessitates the use of external memory to avoid processing data in batches. Using external memory requires a multi-threaded approach to mask memory access latency resulting in high performance gains.

3. The hardware platform is very significant in developing a hardware architecture that utilizes the derivative representation. The platform dictates the type of memory, the available memory ports, the memory bandwidth and the processor-coprocessor system, which must be considered to make an architecture operational.
Chapter 6

Summary and Conclusion

In the first chapter, we introduced the problem of string matching and its importance in the application of DNA sequence mapping. A fast search process is necessary to address technology advancement in next generation sequencing tools which involve a greater number of longer reads. We identified FPGAs as a platform to improve the performance of the search process.

In the second chapter, we explored two groups of literatures. The first literatures are string matching algorithms classified into four: naive, automata, dynamic programming and index-based algorithms. The second literatures are hardware implementations of the classified algorithms. We identified algorithms that are viable for hardware implementation and defined possible hardware improvements from previously known hardware implementations.

In the third chapter, we implemented four exact string matching hardware approaches: finite state machines, encoded reads, hybrid implementation, and CAM-based
implementation. In this chapter, we learned the importance of hardware units as a factor to improve throughput of an implementation. Simpler hardware units utilized in matching use less area allowing more reads placed on the FPGA yielding a higher throughput. We also learned the importance of reprogrammability in the application of DNA sequence mapping. Reprogrammability is required because it involves millions of reads that are impossible to fit on a single FPGA.

In the fourth chapter, we implemented two approximate string matching hardware approaches, keeping in mind requirements of reprogrammability and high throughput. In this chapter, we learned that algorithms requiring significant hardware resources such as the convolution based approach negatively affect throughput because of huge logic dedicated to a few character comparisons. We also learned that a massive parallel architecture using the naive approach is not enough to match the performance of the latest DNA mapping tools based on the FM-index. We proposed the need of using an FM-index based search instead of conventional search methods because of its efficiency and smarter search process.

In the fifth chapter, we implemented the first hardware implementation of the FM-index search method. We modified the search method and presented a new data structure as the new index where the search process operates. We implemented our modified search using external memory to accommodate actual reference genomes that results in the development of FHAST (FPGA Hardware Accelerated Sequencing Tool). The tool can be used as a replacement for the Bowtie tool with a performance gain as high as 70x in reference to our preliminary evaluations.

In conclusion, we started from a careful study of string matching algorithms and
hardware implementation to attempts in developing our own implementation of exact and approximate string matching. We learned from each attempt, which pointed us in the direction of developing FHAST to address the requirement of higher performance. We managed a significant performance gain as high as 70x compared to the Bowtie software tool.
Bibliography


