A High Performance Architecture for an Exact Match Short-Read Aligner Using Burrows-Wheeler Aligner on FPGAs

Dana Abdul Qader
Western Michigan University

A High Performance Architecture for an Exact Match Short-Read Aligner Using Burrows-Wheeler Transform on FPGAs

by

Dana Abdul Qader

A thesis submitted to the Graduate College
in partial fulfillment of the requirements
for the degree of Master of Science in Engineering (Electrical)
Department of Computer and Electrical Engineering
Western Michigan University
December 2015

A High Performance Architecture for an Exact Match Short-Read Aligner Using Burrows-Wheeler Transform on FPGAs

Dana Abdul Qader, M.S.E.

Western Michigan University, 2015

Due to modern DNA sequencing technologies vast amount of short DNA sequences known as short-reads is generated. Biologists need to be able to align the short-reads to a reference genome to be able to make scientific use of the data. Fast and accurate short-read aligner programs are needed to keep up with the pace at which this data is generated. Field Programmable Gate Arrays have been widely used to accelerate many data-intensive bioinformatics applications.

Burrows-Wheeler Transform has been used in the theory of string matching which has led to the development of many short-read alignment programs. This thesis presents a hardware implementation of Burrows-Wheeler Aligner on a Field Programmable Gate Array (FPGA). We specifically concentrated on the exact match of the short-reads and our implementation resulted in execution that is 82X faster than that of a CPU implementation.

Acknowledgements

I would like to express my sincere gratitude and appreciation to my advisor, Prof. Fahad Saeed, for his continuous support, patience, and motivation in my academic study and this thesis research. His guidance helped me in all the time of research and writing of this thesis.

Further I would like to thank my family for their continuous support and for encouraging me in all of my pursuits and inspiring me to follow my dreams for without them this would not have been possible. I would also like to thank my friends who helped me during my study at Western Michigan University.

Dana Abdul Qader

Contents

List of Figures

List of Tables

Chapter 1

Introduction and Objectives

1.1 Introduction

Next generation DNA sequencing technologies are able to generate millions and billions of short DNA sequences in a single run of the machine. The data produced by the sequencing machine is short, fragmented DNA base-pair strings known as short-reads. These short-reads represent the genome to be sequenced, however, the orientation of the short-read relative to the genome is unknown. Hence, the short-reads need to be reconstructed into their original genome to be able to make use of the data.

The short-reads are mapped to a reference genome. The process of finding the corresponding location of each short-read in the reference genome is known as short-read mapping or short-read alignment. The size of the reference genome is typically in billions of base-pairs. A base-pair (bp) is a pair of two complementary nucleotide bases. For example, the size of a human genome is about 3 billion base-pairs. By finding the location of the short-reads in the reference genome, the full sequence can be reconstructed, and differences can be found between the reference genome and the constructed sequence. Scientists usually have an interest in how and where the data is different from the reference genome, which means that the short-read alignment problem involves searching for inexact matches as well as exact matches in the reference genome. The complete sequencing of an organism helps scientist in exploring genetic diseases and cancer genomes, it is an important aspect of modern molecular biology.

Many algorithms have been proposed and used to solve this problem. These algorithms can be divided into two main categories. The first category is algorithms that are based on hash tables. Algorithms in this category work by either hashing the short-reads and scanning through the reference genome, or by hashing the genome and similarly scanning through the short-reads. When hashing the short-reads, there is the advantage of having a flexible memory footprint while having the overhead of scanning through the large reference genome. As for hashing the genome, the program can be easily parallelized, however, a large memory is required as the genome is very large. Programs in this category include RMAP [3], MAQ [4], SOAP [5], BFAST [6] and many others.

The second category includes algorithms based on the concept of prefix/suffix tries. Algorithms under this category usually use Burrows-Wheeler transform. The total size required for the human genome under these algorithms is very small (approximately 2GB). This has led to the development of SOAPV2 [7], BOWTIE [8] and Burrows-Wheeler Aligner [9]. The Burrows-Wheeler Aligner (BWA) mimics the top-down traversal of a prefix trie with the advantage of not having to save the prefix trie in memory. Also, since exact repeats are collapsed in one path on the prefix trie, the short reads do not need to be aligned with every repeat in the reference [9].

Many programs have been developed to solve the short-read alignment problem. However, higher processing speeds could have an enormous impact on fields such as biology, chemistry, and bioinformatics. The use of Field Programmable Gate Arrays (FPGAs) could accelerate the short-read alignment process. FPGAs are widely used for many reasons, one of which is to accelerate different data intensive applications. FPGAs are reconfigurable hardware that can be programmed on a bit level. Hence, they can be much more efficient than software programs. Due to the large memory and processing speed required to solve the short-read alignment problem, our proposed method is to implement Burrows-Wheeler Aligner on an FPGA, while concentrating on exact match of the short-reads against the reference genome.

1.2 Objectives

In this thesis, we introduce a high-performance architecture for short-read alignment using an FPGA and support more than 1 million short-reads. In chapter 2, we give a background on various concepts needed in this thesis. Chapter 3 describes the algorithm used in this thesis in detail. In chapter 4, we demonstrate our novel FPGA implementation for exact match short-read alignment. Finally, chapter 5 and 6 present our results, conclusion, and future work.

Chapter 2

Background

In this chapter, we will describe the scientific importance of identifying different genomes of different species. We will further explain the short-read alignment problem and the various approaches for solving the problem.

2.1 DNA

2.1.1 DNA: Overview

DNA, or deoxyribonucleic acid, is a molecule that consists of four types of nucleotides namely adenine (A), thymine (T), guanine (G), and cytosine (C). Each nucleotide is composed of nucleobases and sugars. The DNA has a double helix structure with two strands of a sugar-phosphate backbone with nitrogenous bases attached, running in opposite directions. The nucleobases on each strand always pair up such that adenine always pairs with thymine, and guanine always pairs with cytosine. The bases along the strands are held together by strong covalent bonds, whereas the base-pairs between the strands are held together with weaker hydrogen bonds. The pair of two such opposite nucleobases that are attached are known as a base-pair [10].

A gene consists of DNA that codes for a protein. An organism's complete set of DNA is known as its genome. The genome, hence, contains all the information required to build the entire human body, and describes every genetic trait of an individual. The difference in the genome between any two individuals is roughly 0.1%. So for every species, there is a single genome that is used as a representative of all the genomes of that species [10].

Under certain circumstances the DNA sequence may change, this is known as mutation. DNA mutation could lead to good outcomes such as causing a species to evolve or it could lead to a bad outcome such as a certain disease. Studying the DNA molecule is therefore very important in the fields of biology, chemistry, and medicine.

2.1.2 DNA Sequencing

Genome sequencing is the process of finding out the order of the DNA nucleotides in a genome. Being able to identify the genome is important to scientists as it gives them the opportunity to explore and understand different genetic diseases and cancer genomes. It could also help scientists improve diagnosis of diseases as disease gene identification could lead to a more accurate diagnosis. Also, it gives them the possibility of detecting genetic diseases earlier.

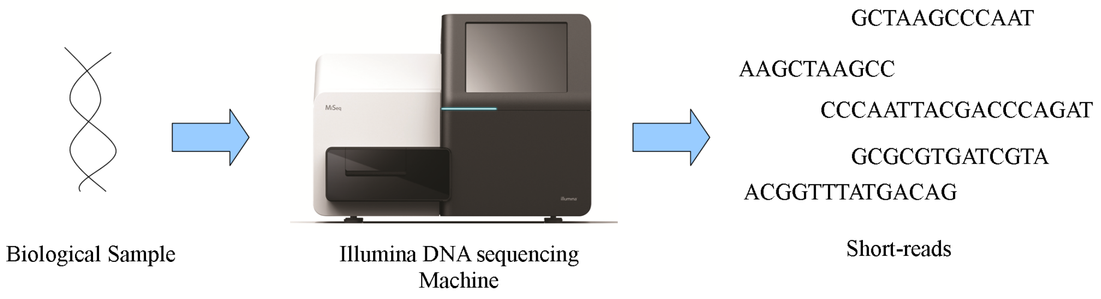


FIGURE 2.1: DNA Sequencing [1]

2.1.2.1 Next Generation Sequencing

Sequencing technology is evolving rapidly. Next generation sequencing is the term used to describe the new evolving technologies used for DNA sequencing. DNA sequencing is the process in which the precise order of the four nucleotide bases within a DNA molecule is found. Several available sequencing methods have been developed over the years.

Frederick Sanger developed the primary "first generation" method in 1977. The Sanger sequencing method was based on a chain-termination method. This way of DNA sequencing was very expensive and required radioactive materials. In 1987, an automatic sequencing machine called the AB370 was developed that was based on capillary electrophoresis which resulted in a much faster sequencing and around 500,000 bases were sequenced in a day. Both the Sanger sequencing and AB370 were used in completing the Human Genome Project. The Human Genome Project was a project that involved finding the exact sequence of the four nucleotide bases that make up the human genome [11].

The Next Generation Sequencing systems include the following:

- **Roche 454 System:** The 454 system was one of the first next generation sequencing machines which is based on pyrosequencing. This machine could generate up to 14 Gb per run of the machine with each short-read having a length up to 700 bp [11].

- **SOLid System:** This system does DNA sequencing by Oliga Ligation Detection. The output of such a machine is 100 Gb per run with a read length of up to 85 bp [12].

- **Illumina:** The Illumina machine uses sequencing by synthesis for DNA sequencing. The output of the machine is as high as 1800 Gb per run of a machine with a short-read length of 150 bp.[1].

- **Compact PGM Sequencers:** Compact PGM sequencers include Ion Personal Genome Machine (PGM) and MiSeq which were launched by Ion Torrent and Illumina [11].

The input to such a system is usually a biological sample or a collection of cloned molecules. The systems output is not the full genome that consists of the exact nucleotide base-pair pattern, but rather short sequences of the genome known as the short-reads (see figure 2.1). The length of the short-reads depends on the system used, as discussed above, and usually varies from 25 base-pairs to up to 700 base-pairs.

2.1.2.2 Costs

The enormous amount of short-reads generated by DNA sequencing machines has been possible due to the reduced cost of DNA sequencing that has led to a growth in DNA research. The cost associated with DNA sequencing has decreased significantly over the years, figure 2.2 illustrates the reducing cost per genome. This figure is from the U.S. based National Institute of Health [2].

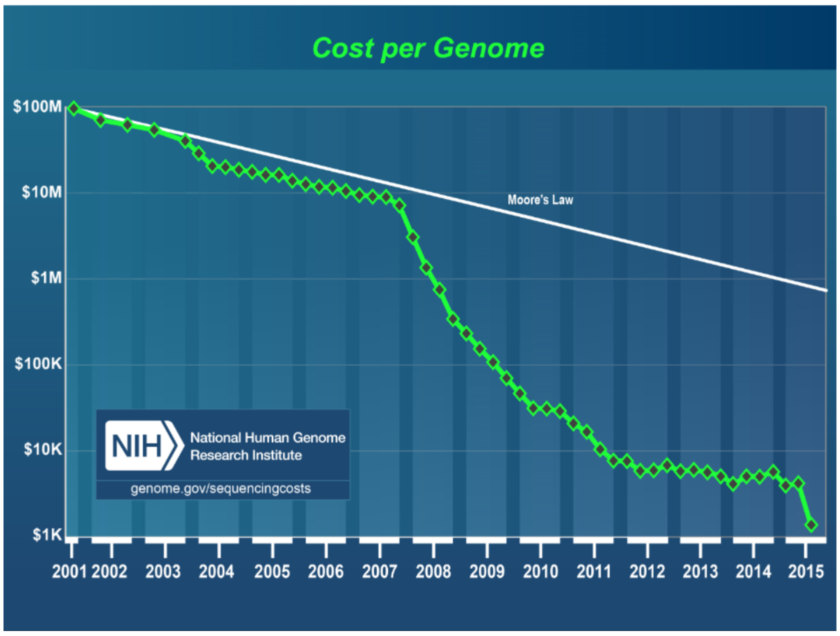


FIGURE 2.2: The decrease in the cost for sequencing the human genome with time from the NHGRI genome sequencing program [2]

2.2 Short-Read Alignment

The generated short-reads must be aligned against a reference genome to be able to reconstruct the sample genome. This process is known as short-read alignment or short-read mapping. Figure 2.3 illustrates the process of short-read alignment. Since the difference between the genomes of a given species is very small, this process simply involves finding the location of the short-reads in the reference genome. These locations can be determined by finding the locations in the reference genome at which an exact match of the short-read is found. Inexact matches can be of interest as well when mutations are considered. For the purpose of this thesis we will concentrate only on exact matches of the short-reads.

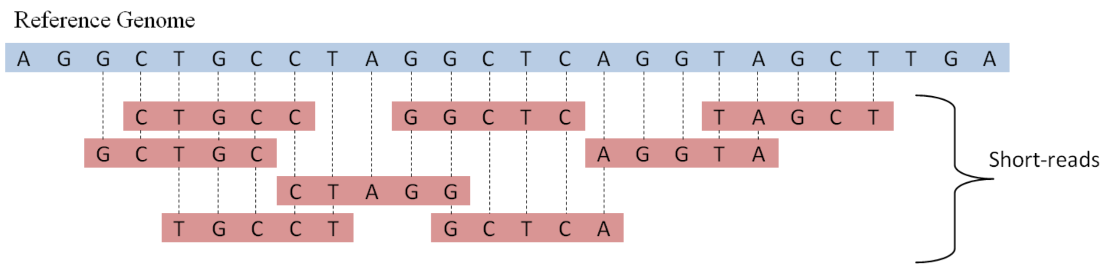


FIGURE 2.3: Short-read alignment

With the next generation sequencing machines generating billions of reads in a fast and inexpensive way, analyzing the data has become a challenge due to the size of the data involved. The reference genome can be very large. For example, the size of the human genome is around 3 billion nucleotide bases. Also, the number of short-reads is increasing rapidly and can be in the order of billions of reads. Searching for billions of reads in a large reference has led to an increase in execution time for software based solutions as the CPU and memory resource usage has increased.

2.2.1 Algorithms

Various algorithms are used to solve the short-read alignment problem. These algorithms are usually based on either hash-tables or prefix/ suffix tries.

2.2.1.1 Hash-Table based Algorithms

Algorithms based on hash-indexing method usually create an index of the entire reference genome. This index will include the location of all possible N base-pair combinations present in the reference genome as shown in figure 2.4, where N is the length of the short-read. Another approach is to use a seed-and-extend method where an index of the reference genome is created using a seed size smaller than the length of the short-read. When a portion of the short-read is matched, and the location is found in the reference genome, it is then extended until a complete match is found.

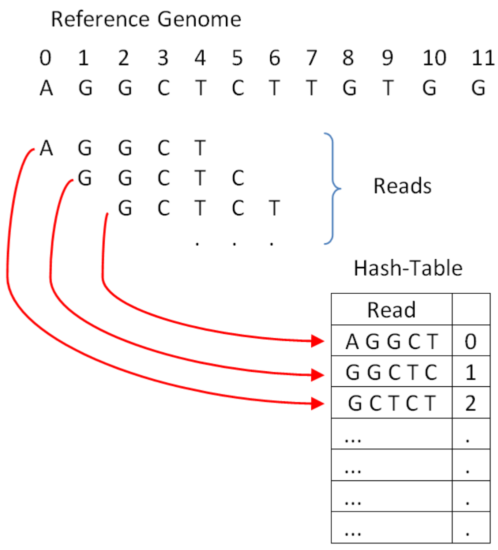


FIGURE 2.4: Hash-table index for short-read alignment

The drawback of using a hash-table approach is that it requires large memory as the reference genome is usually very large. Also, the smaller the short-read or the seed is, the larger the table.

2.2.1.2 Algorithms based on Suffix/Prefix tries

Algorithms in this category rely on a certain representation of suffix or prefix tries. The tries can be represented as either a tree, an array or FM-index [13]. One way to approach the problem is by building a prefix trie of the reference genome and searching through it. The prefix trie is simply a data structure that stores all the prefixes of the string. Figure 2.5 gives an example of a prefix trie for the string "AGGAGT" where the symbol ' ^ ' represents the beginning of the string. The advantage of searching through a prefix trie is that the repeats of a substring are collapsed in one path of the prefix trie. So all the locations of a repeated substring are found in one search [9]. Whereas with the hash-table approach, a search must be performed for each repeat.

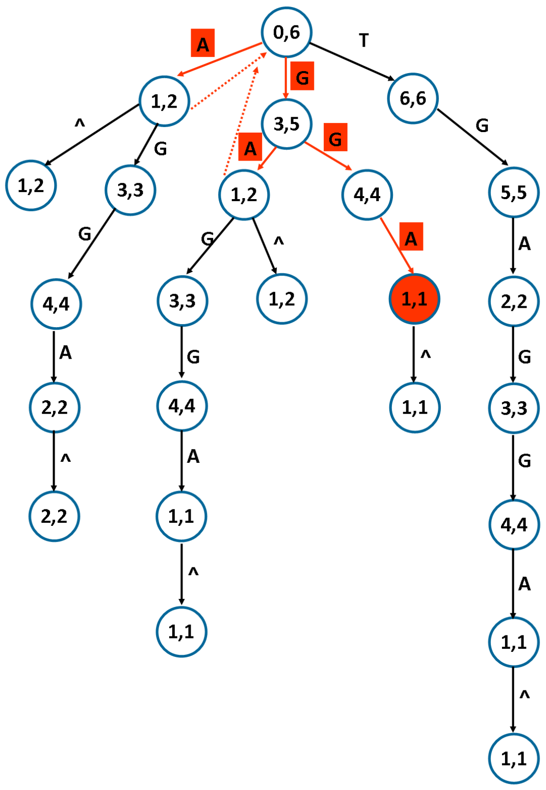


FIGURE 2.5: The Prefix Trie for the string X = "AGGAGT". The path shown in red is the path taken when searching for the string "AGG"

Building a prefix trie requires $O(L^2)$ space, where L is the length of the reference string. For a very large reference string, this would be very impractical to build. A prefix tree (see Figure 2.6), instead of a prefix trie, could be used to reduce the memory to theoretically $LlogL + O(L)$ bits. However, an efficient implementation of the prefix tree still requires 12-17 bytes per nucleotide. Therefore, for a human genome of 3 billion nucleotides it would require 36GB of memory also making it impractical. Another approach is an enhanced suffix array proposed by Abouelhoda et al. [14]. The enhanced suffix array takes 6.25 bytes per nucleotide and has the same time complexity as the suffix tree for exact matching [13].

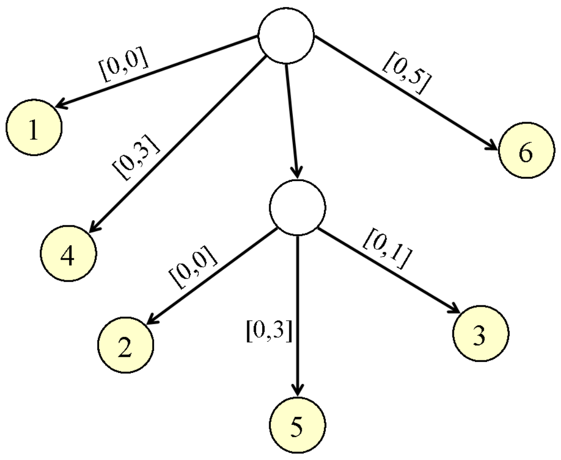


FIGURE 2.6: The Prefix Tree for the string X = “AGGAGT”.

Ferragina and Manzini [15] proposed the FM-index, which further reduced the amount of memory needed to build the data structure. The FM-index is based on the Burrows-Wheeler Transform. The FM-index of a certain string is smaller than the string itself when repeats exist in the string. However, for short-read alignment the index is not compressed to maintain accuracy. This improvement was made by realizing that a child of a certain node on a prefix trie can be found by backtracking on the FM-index data structure. The memory required to build such an index for a human genome is approximately 2 to 8 GB. The time complexity of this backward search is identical to the time complexity of searching a prefix trie and can be done in constant time [13]. Burrows-Wheeler Aligner is an algorithm that uses FM-index and Burrows-Wheeler Transform for short-read alignment, this is further discussed in detail in Chapter 3.

2.3 FPGA: Field Programmable Gate Array

Field Programmable Gate Arrays (FPGAs) are reconfigurable hardware devices designed so that the user can reprogram the device after it has been manufactured, hence the term “field-programmable”. The FPGAs comprise of an array of programmable logic elements and a routing interconnect that wires these elements together in the same way that logic gates can be connected to perform different functions. The reconfigurable nature of the FPGAs allows for the implementation of many different applications. The logic elements can be programmed to do simple logic gates or complex functions.

Central Processing Units (CPUs) tend to have an inherently sequential nature and as such they are unable to exploit the parallelism in particular algorithms. Graphical Processing Units (GPUs) are used for the purpose of parallelizing algorithms by dividing the algorithm into blocks of threads that are sent to the GPU. A streaming multi-processor then executes each block of threads, and a simplified core runs each thread in the block. However, adding more processors will add more overhead and power consumption.

Contrary to the CPU and GPU, the hardware of an FPGA is not predefined and can be reconfigured. The programmable logic blocks can be programmed to run in parallel allowing the FPGA to perform parallel processes efficiently. Also, the FPGAs ability to configure the hardware specifically for an application allows for a very efficient design.

2.3.1 FPGA Components

Figure 2.7 shows the FPGA architecture and its main components that are:

- **Configurable Logic Blocks (CLB):** This contains the logic of the FPGA. The block contains lookup tables (LUTs) which can be configured to perform combinational logic functions. The LUT is basically a RAM that is used to create the function. The CLB also contains a memory unit such as a flip-flop that is used to route data to be able to perform sequential logic.

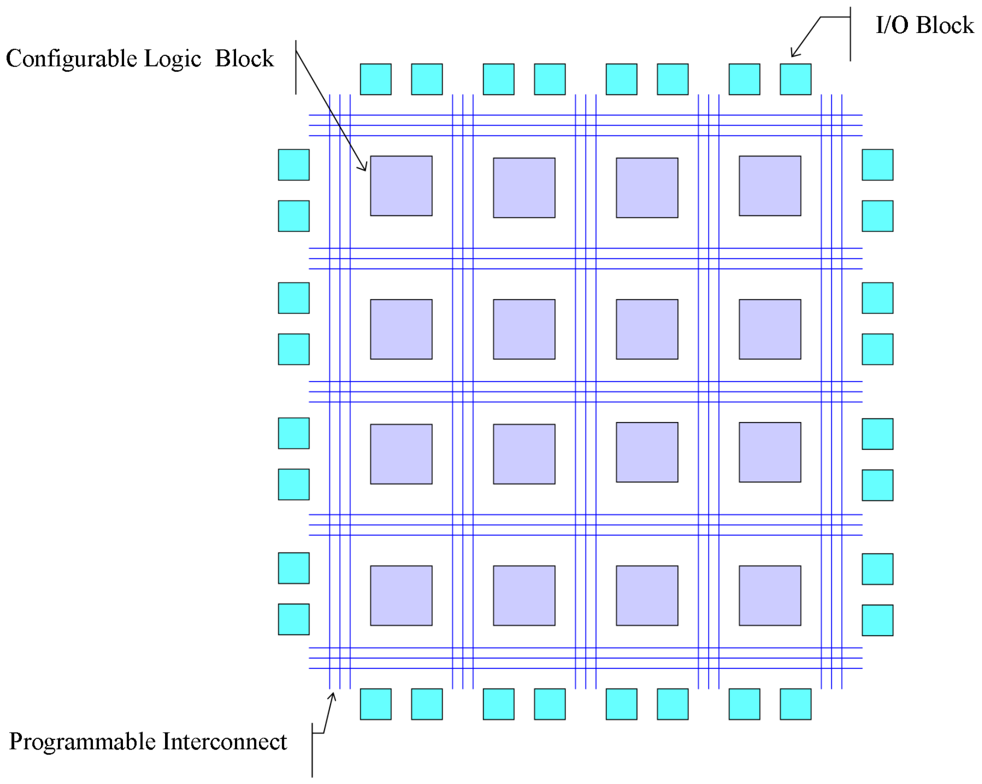


FIGURE 2.7: FPGA Architecture with CLB's, I/O blocks and Interconnect

- **Configurable I/O blocks:** an I/O block is responsible for receiving and sending signals. It does so by using an input buffer and an output buffer. This block makes it possible for the FPGA to connect to other devices and other resources.

- **Programmable Interconnect:** The programmable interconnect is what connects everything together. It connects logic blocks, I/O blocks as well as other resources. The interconnects itself can be programmed to fit the required design.

The configurable logic blocks (CLBs) contain look-up tables (LUTs) as well as flip-flops. The LUT is a very small form of RAM that is used to implement logic functions. The memory written in the LUT for any combinatorial logic is a truth table of the logic. Hence, logic functions are not actually implemented as logic gates as one imagines. A truth table of a logic function is a table that defines the output with a different combination of inputs.

Each FPGA can have many different components such as digital signal processing blocks (DSP), memory blocks and communication ports. This allows the FPGA to be used for many different applications.

2.3.2 Hardware Description Languages

Hardware description languages are languages used to model digital systems. The hardware description languages are able to describe anything from simple logic gates such as AND, NAND, XOR, etc. to complicated systems. C or C++ languages can not be used to define hardware as it does not support characteristics of real hardware.

The two most commonly used hardware description languages are Verilog and VHDL. Verilog is similar to C language and is popular for commercial purposes in the United States, the designs using Verilog are contained in modules. Whereas VHDL is similar to Ada, which is an unpopular high-level computer programming language. Designs using VHDL are contained within entity and architecture pairs and is usually harder to use than Verilog.

Some FPGA's support the use of OpenCL as the description language. OpenCL is a programming platform used for writing programs that can be executed on heterogeneous parallel devices and is based on the C language [16]. The use of OpenCL in FPGAs allows programmers to program in C and target FPGA functions by using OpenCL constructs. However, programming using OpenCL for describing hardware usually ends up in the synthesized hardware that is not as efficient as when using Verilog or VHDL as OpenCL is a higher level language. For the above reasons the chosen hardware description language for this thesis is Verilog.

2.4 Previous Work

Many software-based algorithms have been developed as a solution to the short-read alignment problem. Due to the rapid increase in the number of generated short-reads, many attempts have been made to accelerate such software by using either the GPU or FPGA. The following section will outline some existing solutions.

All recent DNA sequencing technologies are able to produce short-reads of the order of Giga base-pairs. For researchers to be able to keep up with this rapidly developing technology, fast and accurate algorithms are being developed.

Many algorithms have been developed based on hash-tables some of which are:

- **BLAST:** is one of the first algorithms developed based on hash-tables. BLAST works by hashing the short-reads, it keeps the position of all k-mer subsequences of the query in a hash table and works by using these k-mer subsequences as keys to search through the reference genome. It then uses a seed-and-extend method to extend the k-mer subsequence match until it finds a match of the full short-read. It then finally uses the Smith-Waterman alignment tool [17].
- **SOAP:** works by hashing the reference genome rather than the short-reads. This results in an algorithm that can be easily parallelized [5].
- **MAQ:** it stands for Mapping and Assembly with Quality. MAQ maps short-reads against a reference genome by using quality scores [4].
- **RMAP:** it was originally designed to map DNA sequences generated by the Illumina machine with short-read lengths ranging from 25 to 50 bases. This alignment algorithm also uses quality scores and weight-matrix matching to map the short-reads and improve accuracy [3].

- **SHRiMP:** it uses very larger memory. Originally it works by hashing the short-reads, however, some newer versions work by hashing the genome instead. Due to its need for very large memory, it is not widely used [18].

There are many more existing aligners that are also based on hash-tables such as BFAST [6], CloudBurst [19], PERM [20] and GNUMAP [21]. The use of suffix and prefix tries for aligning sequences has also led to the development of many algorithms. Some of these existing aligners are listed below.

- **Bowtie:** it implements the FM-index to map short-reads of lengths upto 50 base-pairs and is one of the first aligners to adopt FM-index for short-read alignment [8].
- **SOAPv2:** this is an algorithm that improves on SOAPv1 and is based on Burrows-Wheeler Transform. It also uses a hash-table to improve the speed by searching the BWT reference index using the hash-table [7].
- **BWA:** it stands for Burrows-Wheeler Aligner and is based on the Burrows-Wheeler Transform (FM-index) using a backward search. It is able to mimic the top-down traversal of a prefix trie without building the trie in memory, as a result reducing the required memory to map short-reads [9].
- **segemehl:** it is a short-read aligner that is based on enhanced suffix arrays. It is flexible with the read length, however, it uses much more memory than other suffix and prefix based algorithms [22].

Prefix and suffix based algorithms tend to use less memory than hash-table based algorithms. For this reason, we chose to implement the exact match of Burrows-Wheeler Aligner algorithm on an FPGA to accelerate the process of short-read mapping. For the purpose of this thesis our concentration is only on exact matching of short-reads.

Many attempts have been made to accelerate existing software aligners by using the GPU. Some of these aligners are listed below.

- **CUSHAW:** uses Compute Unified Device Architecture (CUDA) programming language to develop a GPU implementation based on the Burrows-Wheeler Transform. CUSHAW reports to be faster than both BWA and Bowtie [23].
- **BarraCUDA:** is a GPU implementation of BWA using CUDA. The BArraCUDA reports to be six times faster than a CPU implementation of BWA [24].
- **SOAPv3:** is also a GPU implementation using CUDA of SOAP2. It is reported to be 7.5 times faster than BWA and 20 times faster than Bowtie [25].
- **MUMmerGPU:** is a GPU implementation, also using CUDA, for an algorithm based on suffix tree. It reports to be 3.5 times faster than MUMmerGPU running on a CPU [26].

The use of reconfigurable hardware such as Field Programmable Gate Arrays has become very popular due to its inherent parallelism. It has been used to accelerate many different applications. Below is a list of short-read aligners accelerated using FPGA's.

- **Shepard:** is a hardware implementation of an exact match short-read aligner. It uses the Convey HC-1 which is a high-performance computer cluster that contains a motherboard and a coprocessor board. The coprocessor board includes at least 32GB of coprocessor memory and has a set of 14 FPGAs. It reports to align 350 million short-reads per second [27].

- **Olson et al.** implemented a hash-table based algorithm known as BFAST on FPGA. It reports to be 250 times faster than BFAST on CPU but only 31 times faster than Bowtie which is a suffix tree based algorithm. It does not make any comparisons to BWA [28].

- **Kasap et al.** designed and implemented an FPGA implementation of BLAST. They report having a speedup of 52 compared to similar software implementations [29].

- **Sogabe et al.** implemented a hash-table method on FPGA. They report to be 2.5 times faster than Bowtie and 0.44 times than Olson et al. [28]. Their implementation involves hashing the short-reads by further dividing them into smaller seeds [30].

Many efforts have been made to accelerate BWA. Chen et al. [31] developed Cgap-align, which achieves a speedup of around 1.2 in comparison to BWA. They do this by using a data structure that combines elements of both the suffix array and the suffix tree. Zhang et al. [32] improved the performance of BWA by reducing the last-level cache (LLC) misses by 30%. As far as our knowledge, there has been no attempts to accelerate BWA on hardware.

Chapter 3

Algorithm

Next Generation Sequencing generates billions of short-reads which are then mapped to a reference genome. There are many short-read alignment techniques, one of which is Burrows-Wheeler Aligner. Burrows-Wheeler Aligner is based on a backward search with FM-index and Burrows-Wheeler Transform. In this chapter, we will focus on how the Burrows-Wheeler Aligner is used for exact match short-read alignment.

3.1 Prefix Trie

The Prefix Trie is a data structure for storing all the prefixes of the string in a way that allows for a fast lookup. Figure 2.5 shows the prefix trie for the string "AGGAGT" where the symbol '^' marks the beginning of the string.

Each node in the prefix trie represents a string which is the edge symbol concatenation from the node to the root. If we are to search for the exact match of the substring AGG in the prefix trie, the path in red (Figure 2.5) is the path that would be taken to find the substring. This is done in $O(|W|)$ time where W is the length of the substring. The red node represents the found string AGG. Each node in the prefix trie is numbered with the suffix array interval of the string that is represented by that node. The suffix array is an array of the indexes of all the sorted suffixes of the string as can be seen in figure 3.1. The suffix array interval is the interval at which the substring occurs in the suffix array. For example, the node highlighted in red shows the suffix array interval {1,1} for the found string AGG and $S[1] = 0$, this means that the substring AGG occurs at location 0 of the original string. Another example will be if we are to search for the substring AG. The node representing the substring AG from (Figure 2.5) has the suffix interval {1,2} this means that the string AG occurs twice at locations $S[1] = 0$ and at $S[2] = 3$ of the original string. Note how we did not need to search for all the repeats of the substring as they are collapsed in one path of the trie, also since the suffix array is sorted only one such interval will exist for each substring [9].

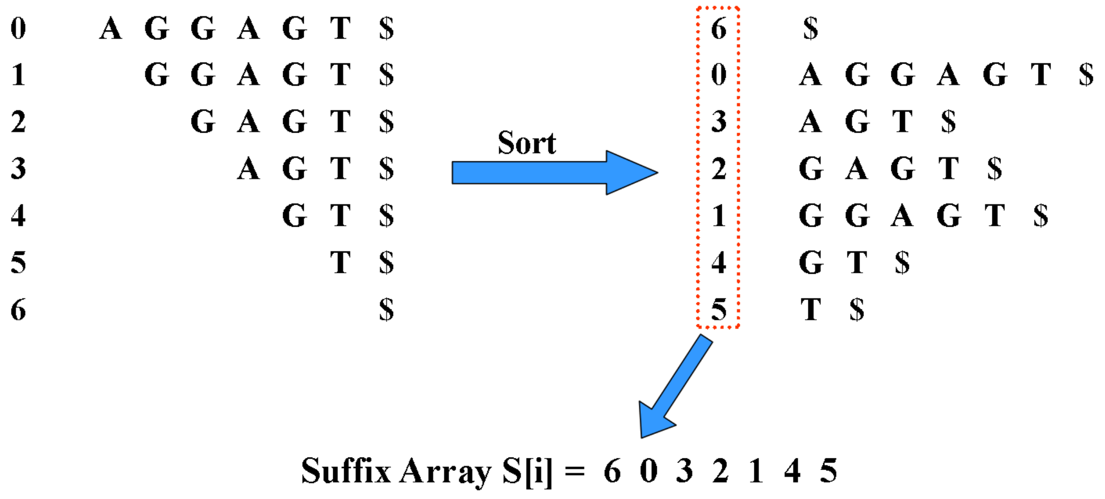


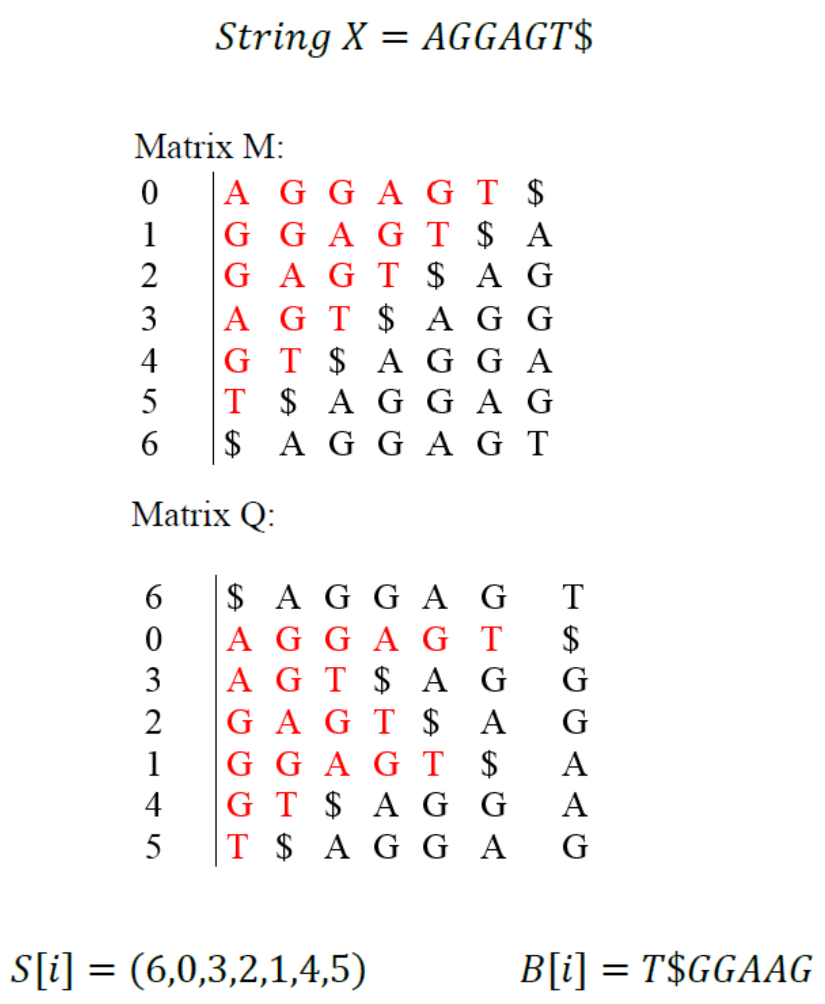
FIGURE 3.1: Suffix array for string X = AGGAGT

As mentioned earlier, it is impractical to build a prefix trie due to its memory requirements. The FM-index and the Burrows-Wheeler Transform are able to mimic the traversal of a prefix pie without having to build the trie in memory.

3.2 Burrows-Wheeler Transform and the FM-index

Burrows-Wheeler Transform is used in many different applications such as data compression, image processing and in string matching. The Burrows-Wheeler Transform was first introduced in [33] and was initially used for the purpose of data compression. The Burrows-Wheeler Transform of a string X is simply a permutation of that string. What makes Burrows-Wheeler Transform unique is the fact that the permuted string is reversible.

Let $\Sigma = \{'A',' G',' C',' T'\}$ be the alphabet over which the input string X is created. A dollar sign '\$' is added to the end of the input String X and is lexicographically the smallest. The BWT permuted string B is generated by performing all the cyclic rotations on the original input string X. Figure 3.2 shows two matrices, the first matrix M is simply the cyclic rotations of the string $X =$ "AGGAGT\$". The second matrix Q is the result of sorting the rows in matrix M lexicographically. The Burrows-Wheeler Transform of the string X, string B, is now simply the last column of Matrix Q. S[i] is the suffix array of string X, such that S[i] is the position of the ith lexicographically smallest suffix in String X, and so we can say that $B[i] = X[S(i) - 1]$. The suffixes of string X are shown in red in figure 3.2. In the second matrix, these suffixes are then sorted lexicographically. The indices resulting from sorting these cyclic rotations gives us the suffix array S[i]. In this way, the Burrows-Wheeler Transform is able to mimic the prefix trie without having to save the trie in memory [9].



$String\ X = AGGAGT\$$

Matrix M:

0	A	G	G	A	G	T	$
1	G	G	A	G	T	$	A
2	G	A	G	T	$	A	G
3	A	G	T	$	A	G	G
4	G	T	$	A	G	G	A
5	T	$	A	G	G	A	G
6	$	A	G	G	A	G	T

Matrix Q:

6	$	A	G	G	A	G	T
0	A	G	G	A	G	T	$
3	A	G	T	$	A	G	G
2	G	A	G	T	$	A	G
1	G	G	A	G	T	$	A
4	G	T	$	A	G	G	A
5	T	$	A	G	G	A	G

$S[i] = (6,0,3,2,1,4,5)$ $B[i] = T\$GGAAG$

FIGURE 3.2: BWT for input string X = AGGAGT

The BWT string B is the output of the Burrows-Wheeler Transform algorithm. The output is usually easier to compress than the input string because the Burrows-Wheeler Transform tends to group similar characters together. Since the BWT string B can be reversible, BWT is used in lossless string compression tools such as bzip.

3.2.1 Backward Search

The Burrows-Wheeler Transform along with the suffix array can be used for string matching. Ferragina and Manzini [15] developed an algorithm based on Burrows-Wheeler Transform that is able to search for substrings in the compressed string. For the purpose of higher accuracy for the short-read alignment problem, the search is done without compressing the BWT string B.

Each occurrence of a substring W of String X will occur in an interval in the suffix array since the suffixes are in lexicographical order [9]. The following equations from [9] are then defined.

$$\underline{R}(W) = min\{k : W \text{ is the prefix of } X_{S(k)}\}$$

$$\bar{R}(W) = max\{k : W \text{ is the prefix of } X_{S(k)}\}$$

The interval $[\underline{R}(W), \bar{R}(W)]$ is known as the suffix interval. If we know the interval we then know the locations, meaning that string alignment is equivalent to searching for the suffix array intervals of the substrings that match string X. If we are looking for exact matches then only one such array can exist. We need to define a count array C(a), which is the number of symbols in X that are lexicographically smaller than the symbol a. Given that a is a symbol in Σ. Also, O(a, i) is the occurrence array which is the number of occurrences of symbol a in B[0, i]. Then if W is a substring of X the following equations which were proven by [15] can be defined:

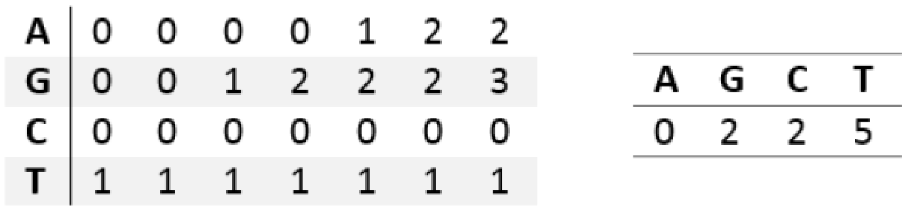
$$\underline{\mathrm{R}}(aW) = C(a) + O(a, \underline{R}(W) - 1) + 1$$

$$\bar{R}(aW) = C(a) + O(a, \bar{R}(W))$$

This makes it possible to test whether W is a substring of X by iteratively calculating $\underline{R}(W)$ and $\bar{R}(W)$ from the end of W, this is known as backward search.

The suffix array of an empty string would then be $[0, |X| - 1]$ where $|X|$ is the size of the reference string X /citeBWA. For example, from figure 3.2 the suffix interval of "AG" is [1, 2]. Hence, "AG" exists in locations S $(\underline{R}(AG)) = S(1) = 0$ and S $(\bar{R}(AG)) = S(2) = 3$ of the original reference string.

After we construct the BWT along with the suffix array we must construct the count array $C(.)$ and the occurrence array $O(.,.)$ for each alphabet $\Sigma = \{`A', `G', `C', `T'\}$. Figure 3.3 shows an example for such arrays. The first array stores the counts of symbols lexicographically smaller than each letter of the alphabet Σ. The second array counts the number of occurrences of each letter up to each position in B. After these arrays have been constructed, they are used to search for matches of the substring in the original string in linear time. Using the equations, we can iteratively find a match by prepending a single character from the query to the current substring. If the lower bound is smaller than the upper bound, this would mean that the substring exists in the original string. However, if the lower bound is greater than the upper bound this indicates that the string does not occur in the original string. If the query is found, the lower and upper bound can be used to find the exact location in the original string.



A	0	0	0	0	1	2	2
G	0	0	1	2	2	2	3
C	0	0	0	0	0	0	0
T	1	1	1	1	1	1	1

A	G	C	T
0	2	2	5

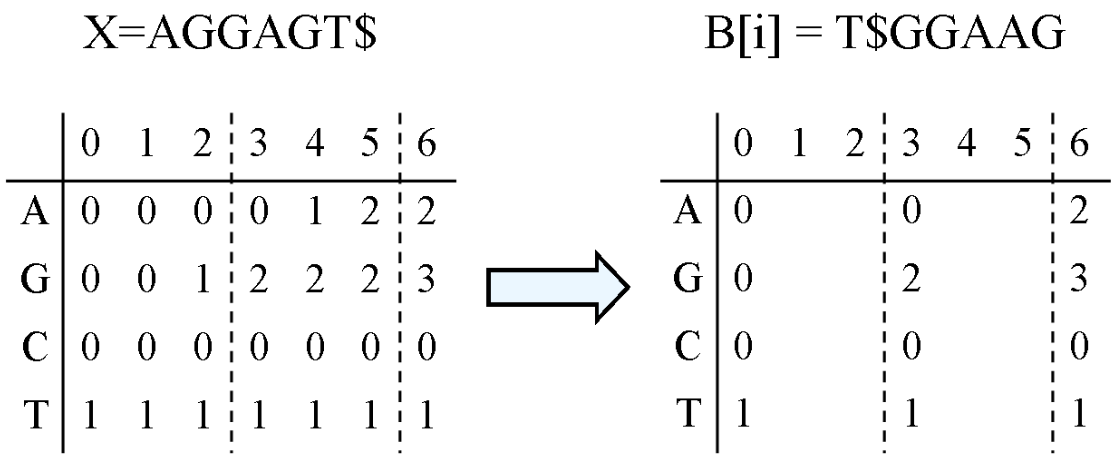
FIGURE 3.3: The left table shows the O (.) arrays for each symbol and the right table shows the C (.) arrays for each symbol

The Burrows-Wheeler Transform string B does not require much memory to store as only 2 bits per nucleotide can be used. Hence, the BWT string B requires $2.|X|$ bits of memory where $|X|$ is the length of the input, String X. For a human genome, this would be around 750MB, which can be stored in a memory of a standard desktop PC. The array $C(.)$ is just an integer that stores the count and so does not require much memory. However, the problem is with the occurrence array $O(.,.)$. The memory requirement for the occurrence array is $O(|X|)$. If a full human genome is to be indexed, a 32-bit integer is required and for all four of the occurrence arrays this could mean $32.4.|X|$ which would approximately be 48GB for the human genome. Burrows-Wheeler Aligner developed a way to help reduce the amount of memory; this will be discussed in the next section.

3.3 Burrows-Wheeler Aligner

Li et al. [9] developed a fast algorithm that has been very widely used and trusted for DNA sequencing short-read alignment. The algorithm also supports inexact matching, however, we will only concentrate on exact matching.

Burrows-Wheeler Aligner developed a method of reducing the memory required for building the occurrence arrays. Instead of building the entire array, they divide the array into buckets and only store one entry from each bucket and calculate the rest on the go. Figure 3.4 shows how this method works for the string $X = AGGAGT\$$ and its BWT string $B = T\$GGAAG$. Here the occurrence array is divided into buckets and only the first entry is recorded. If, for example, we want to find the value of O[‘G’,2], we know that O[‘G’,0] = 0 then we just need to check if ’G’ occurs in B[1] and B[2] and increment O[‘G’,0] accordingly. Here ‘G’ does not occur in B[1] but does occur in B[2] and as such we increment O[‘G’,0] to get O[‘G’,2] = 1. If we check back to figure 3.3 we can see that O[‘G’,2] is in fact equal to 1.



X=AGGAGT$

	0	1	2	3	4	5	6
A	0	0	0	0	1	2	2
G	0	0	1	2	2	2	3
C	0	0	0	0	0	0	0
T	1	1	1	1	1	1	1

B[i] = T$GGAAG

	0	1	2	3	4	5	6
A	0			0			2
G	0			2			3
C	0			0			0
T	1			1			1

FIGURE 3.4: Reducing the occurrence array by using a bucket size of 3 nucleotides

The occurrence array is divided into buckets of N base-pair intervals and is stored in memory. When trying to find a value from the occurrence array, the first entry smaller than the one required is loaded. After that, occurrences of the nucleotide base is found by directly looking into string B and incrementing the value that has already been loaded.

Using this method the human genome, for example, would only require around 375MB for all four occurrence arrays, which is much smaller and much more reasonable than the 48GB that was seen in the earlier section. This is if the occurrence array was divided into 128 base-pair intervals. So the memory requirement for the occurrence array is now $O(|X|/N)$ where N is the base-pair interval. Note that by reducing the memory that is needed for the occurrence array the computational cost increases.

The Burrows-Wheeler aligner algorithm proposed in [9] has been edited to account only for exact matching. The algorithm is shown in figure 3.5. The input to this algorithm is the reference string X and substring W. The output is k and l where k is the lower bound of the suffix interval and l is the upper bound of the suffix interval.

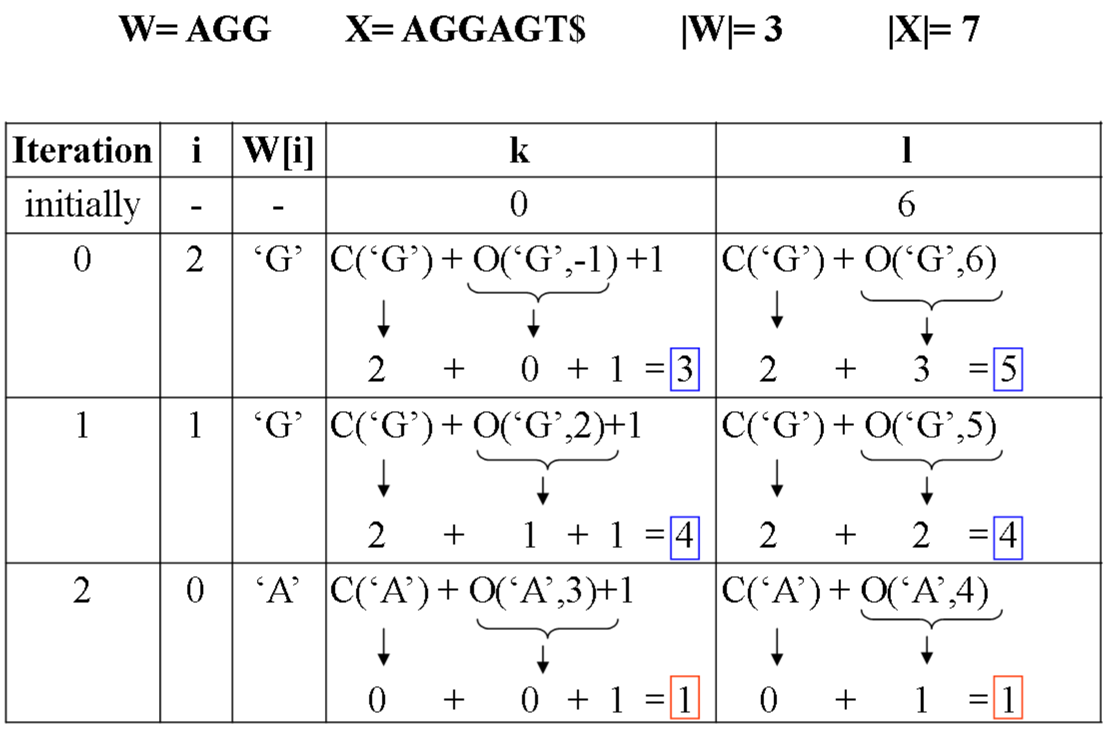
```
Pre-Calculations

    Calculate BWT string B for reference String X
    Calculate Array C(.)
    Calculate Array O(.,.)

ExactSearch(W, B, C(.), O(.,.))

    k = 0
    l = |X| - 1
    for i = |W|  - 1 to 0 do
        k = C( W[i] ) + O( W[i] , k-1 ) + 1
        l = C( W[i] ) + O( W[i] , l )
        i = i - 1
        if k > l
            BREAK
    if k > l
        return "NOT FOUND"
    else
        return [k,l]
```

FIGURE 3.5: Algorithm for exact matching



W= AGG   X= AGGAGT$ |W|= 3 |X|= 7

Iteration	i	W[i]	k	l
initially	-	-	0	6
0	2	‘G’	$\underbrace{C(\text{‘G’})}_{2} + \underbrace{O(\text{‘G’},-1)}_{0} + 1 = \boxed{3}$	$\underbrace{C(\text{‘G’})}_{2} + \underbrace{O(\text{‘G’},6)}_{3} = \boxed{5}$
1	1	‘G’	$\underbrace{C(\text{‘G’})}_{2} + \underbrace{O(\text{‘G’},2)}_{1} + 1 = \boxed{4}$	$\underbrace{C(\text{‘G’})}_{2} + \underbrace{O(\text{‘G’},5)}_{2} = \boxed{4}$
2	0	‘A’	$\underbrace{C(\text{‘A’})}_{0} + \underbrace{O(\text{‘A’},3)}_{0} + 1 = \boxed{1}$	$\underbrace{C(\text{‘A’})}_{0} + \underbrace{O(\text{‘A’},4)}_{1} = \boxed{1}$

FIGURE 3.6: This figure shows how the algorithm from figure 3.5 works when searching for the substring AGG in the original reference string AGGAGT

For example, if we are to search the above algorithm for the String $W = AGG$ in the reference string $X = AGGAGT$ the algorithm would follow as seen in figure 3.6. Note that $O(., -1)$ is always zero. The output of the algorithm would be $k = 1$ and $l = 1$ which is the suffix interval $\{1, 1\}$ of the substring $W = AGG$.

Chapter 4

FPGA Implementation

In this chapter, we describe our hardware implementation of an exact match short-read aligner. First we need to describe the data flow of the algorithm. The flow chart in figure 4.1 shows the steps that need to be taken before we are able to map the short-reads to a reference genome. We then go on to describe how each part is implemented.

There are two main components to our implementation, the first component is software which constitutes of the generation of the random short-reads and the reference genome as well as the Burrows-Wheeler Transform string B of the reference genome. Our second component is the hardware which calculates the arrays and maps the short-reads.

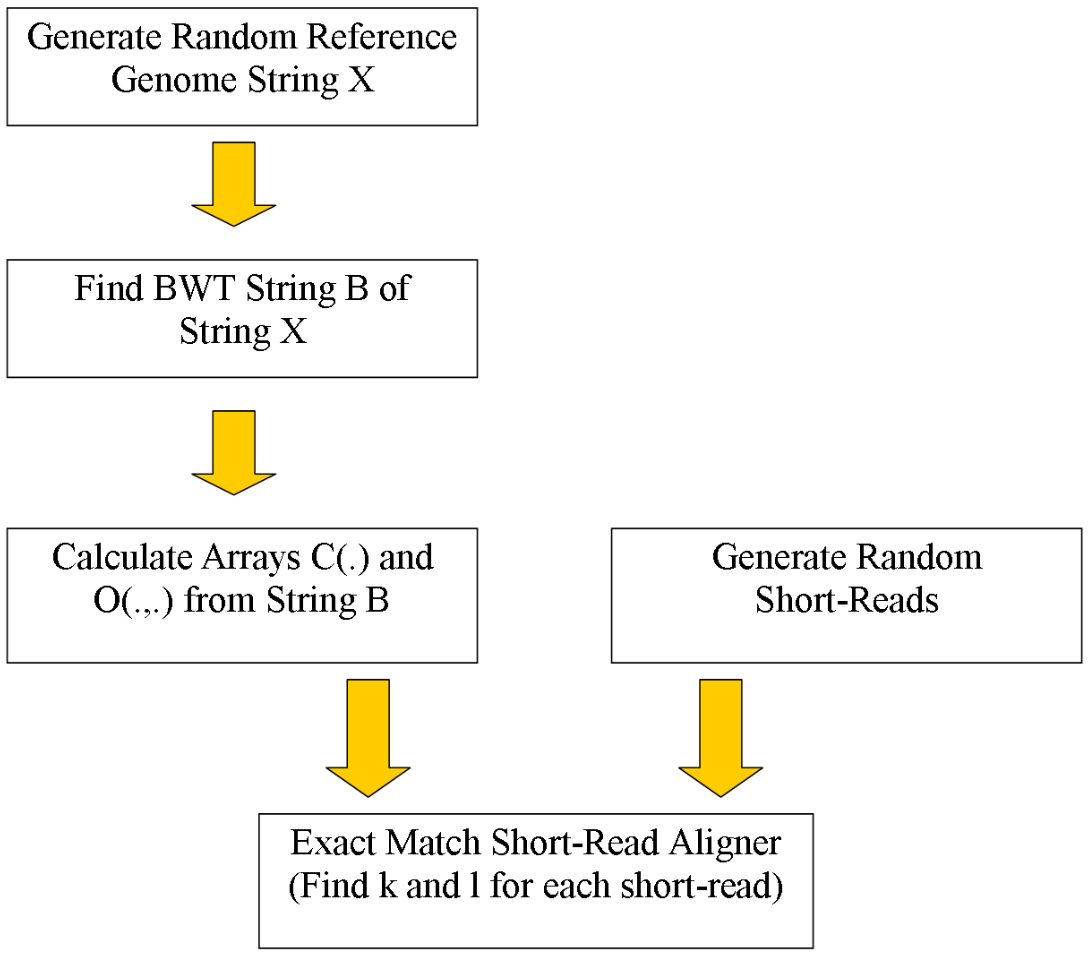

Generate Random Reference Genome String X
Find BWT String B of String X
Calculate Arrays C(.) and O(.,.) from String B
Generate Random Short-Reads
Exact Match Short-Read Aligner (Find k and l for each short-read)

FIGURE 4.1: Flow chart describing the data flow of the algorithm

It is important to note that the Burrows-Wheeler Transform as well as creating the arrays C(.) and O(.,.) is only done once. Hence, the BWT and the arrays are only found once for a given reference genome, and usually there is only one such reference genome for a particular species. For example, for the human genome there is one most recent version of the reference genome that is most commonly used, and that is known as GRCh38 [34]. Also for our implementation we disregard the '\$' sign and use 2-bits to represent a nucleotide base.

4.1 Software Component

4.1.1 Random String Generator

For our design, we randomly generate the short-reads as well as the reference genome. In practice the DNA sequencing machines generate the short-reads and reference genomes are available online. However, for the purpose of testing our simulation we chose to generate the short-reads and the reference genome using pseudo-random number generator. The pseud-code in figure 4.2 represents the code used to generate the short-reads. In the same manner, we generate the reference genome. The input to this algorithm is the Number_ShortReads which is the number of short-reads and ShortRead_Length which is the length of each short-read. We generate a random number using the function *random*() and by taking the modulus of that number we insure the result is between 0 and 4 since we only have four nucelotide bases. The result of the modulus is then an index of the nucleotide array which includes all four bases. We then store the candidate nucleotide in our short-read array.

```
ShortRead_Generator(Number_ShortReads, ShortRead_Length)

            value = 0
            char sequence[Number_ShortReads][ShortRead_Length]
            nucleotide[] {'A', 'G', 'C', 'T'}
            while (value < Number_ShortReads)
            {
                for  i = 0 to ShortRead_Length
                {
                    sequence[value][i] += nucleotide[(random()%4)]
                }
                value = value +1
            }
```

FIGURE 4.2: Short-read generator

4.1.2 Burrows-Wheeler Transform

To create the BWT string B of a reference string X, we must generate all cyclic rotations of string X as described in Chapter 3, section 3.2. If the length of String X is $|X|$ than this means that we would need $O(|X|^2)$ memory to be able to generate all the cyclic rotations. Since the reference string is usually very large, this is not practical. In practice, the BWT is generated by first generating the suffix array. Hon et al. [35] developed an algorithm that would only require around 1GB of working space for the human genome.

If we look back at figure 3.2 we can see that the first column of matrix Q is simply the original string sorted lexicographically. Remember that the '\$' sign is lexicographically the smallest. Also, we can notice that each character of the last column is the prefix of each character in the first column of the same row due to the cyclic rotations. Since each column is just a permutation of the original string, they can be presented as indexes of the original string. This means that the last column, which is the BWT string B, indexes can be found by subtracting one from the indexes of the first column. Figure 4.3 helps elaborate this further.

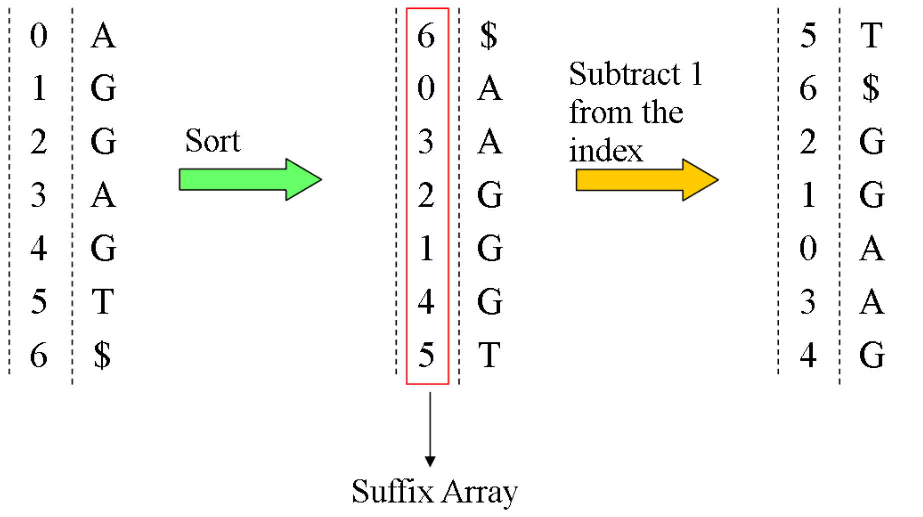


FIGURE 4.3: This figure shows how the Burrows-Wheeler Transform can be found without generating all the cyclic rotations of the string

Now, the Burrows-Wheeler Transform can be seen as simply a sorting problem. If there are no repeated characters in the input string X, then the sorting operation can be done in one iteration. However, the problem occurs when we have many repeated characters in String X, which is usually the case. In order to sort string X with the existence of repeated characters, the character is replaced by its suffix, and it is sorted again. This is repeated up until no more repeats exist. This is the method used in our implementation for finding the BWT string B.

Since the Burrows-Wheeler Transform is calculated only once for the reference genome, we chose to implement the BWT in software. The output of the BWT is then passed to our simulation on the hardware side.

4.2 Hardware Design

The Burrows-Wheeler Transform of the reference genome, as well as the short-reads, are loaded to the FPGA. The FPGA then, given the BWT, calculates the occurrence array, as well as the count array $C(.)$. The arrays are then fed to the exact matcher unit of the FPGA. The overall architecture used to implement the short-read mapping is shown in figure 4.4.

4.2.1 Occurrence Array Unit

We recall from section 3.2.1 that the occurrence array is found by cumulatively counting the occurrences of each of the four bases in the BWT string B (see figure 3.3). In our design, we generate four separate occurrence arrays, one for each nucleotide base $'A','G','C','T'$.

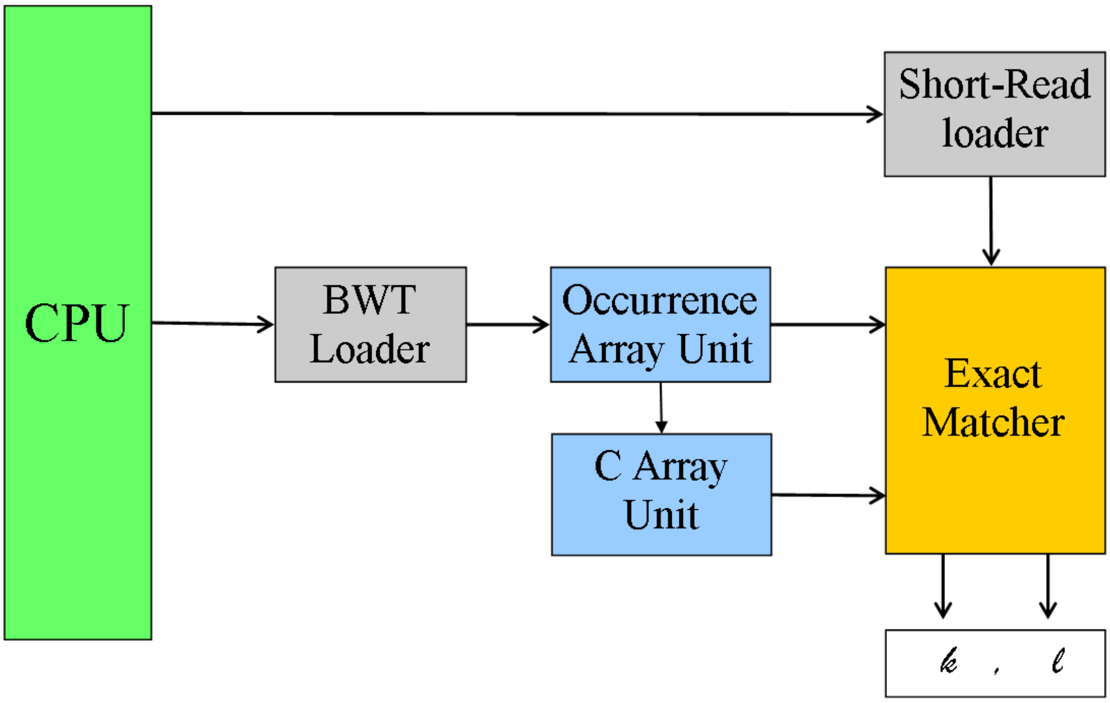


FIGURE 4.4: Overall architecture of our design

The Burrows-Wheeler Transform string B, which is computed on the host CPU, is loaded onto the FPGA. The BWT string B is then shifted in character by character. The arrays are initialized to zero upon reset. Depending on the character, the corresponding occurrence array is incremented. A single counter is used to index the arrays. Every time a character is shifted in, the counter increments by one. As a result, with every clock cycle the counter would increment one element of a certain array and leaving the other arrays the same. Figure 4.5 represents the architecture of such a unit.

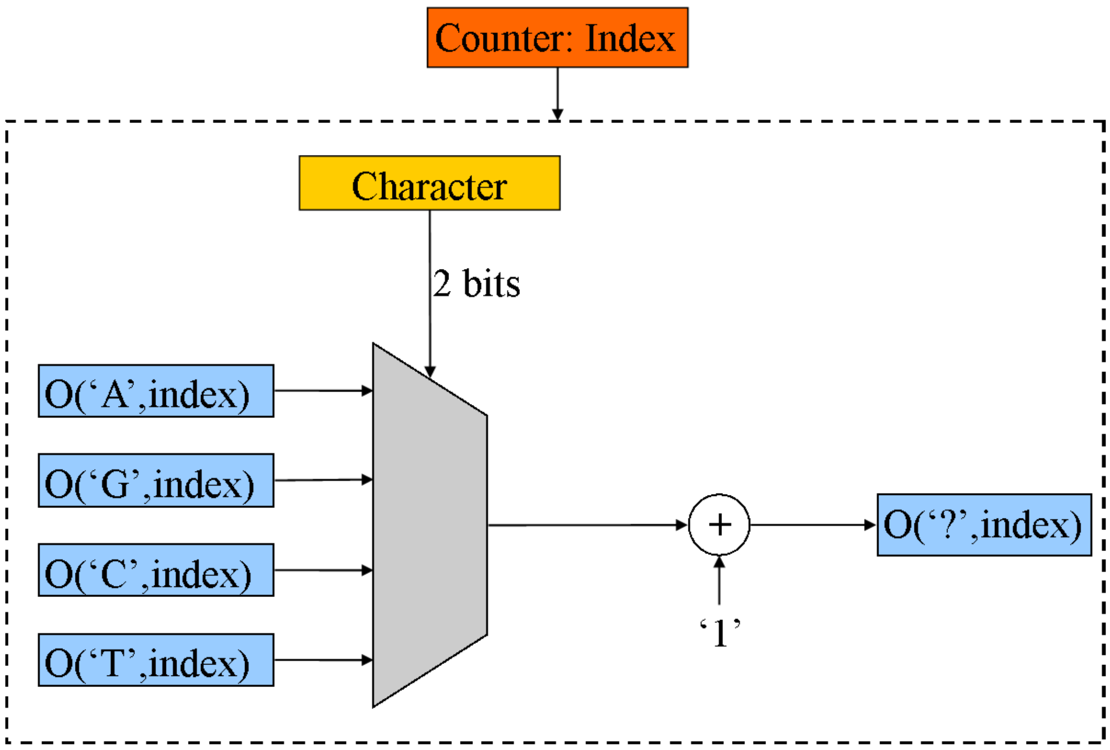


FIGURE 4.5: Occurrence array unit

4.2.2 C Array Unit

To create the count array C, we implement four registers one for each nucleotide base. Remember from section 3.2.1 that the count array for any nucleotide is simply the number of nucleotide bases in string B lexicographically smaller than the nucleotide itself. Register C(A) (count array for 'A') is always zero since there is no character lexicographically smaller than 'A'. Since 'A' is the only letter lexicographically small than 'C', then register C(C) is basically the number of 'A's in string B. Hence, array C(G) is the sum of the count of 'A's and 'C's in string B. Finally, array C(T) is the sum of the count of 'A's, 'C's and 'G's in string B.

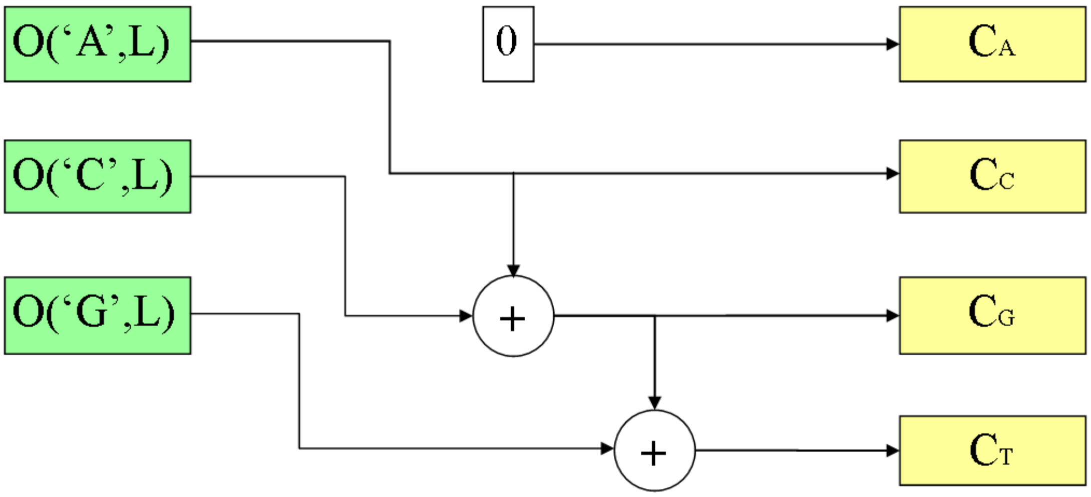


FIGURE 4.6: C array unit

The counts can be found directly from the occurrence arrays. The last value in the occurrence array for each nucleotide corresponds to the total count of that nucleotide in String B. Figure 4.6 shows the architecture for the C Array Unit.

4.2.3 Exact Matcher Unit

The exact matcher unit takes in one character from the short-read at a time. Also, the output from the occurrence array unit as well as the C array unit is fed as input to the exact matcher unit. We can recall from section 3.2.1 that finding k and l is an iterative process. After every iteration the new value of k and l is used as input, i.e. is fed back to the system. Also, it is important to note that the number of iterations required to find the exact match is exactly the length of the short-read. For example, if the length of the short-read is 50, then it would take 50 iterations to find the values of k and l. So if we have 100 short-reads, then finding the exact match of all 100 short-reads would require 100$*$50 iterations, this means that the time complexity to find k and l for N short-reads each of length $|W|$ is $O(N * |W|)$. However since the number of short-reads is much greater than the length of the short-read, and the length of the short-read is a constant, the time complexity becomes $O(N)$. One very important observation is that the time complexity is independent of the length of the reference genome.

Figure 4.7 shows the architecture used in our design for the exact matcher unit. Upon reset, the value of k is initialized to 0, and the value of l is initialized to the length of the string. K and l both are used to index the occurrence arrays. Depending on the value of the character, a particular occurrence array is added to its respective count array C. This value is then fed back and used as an index to the occurrence array. With every iteration, a counter increments its value. Once the value of this counter reaches the length of the short-read, the values of k and l are then sent as output as they would correspond to the suffix array interval of the current short-read. After that, the value of k and l will reset, and the exact matcher unit will read a new character of a new short-read.

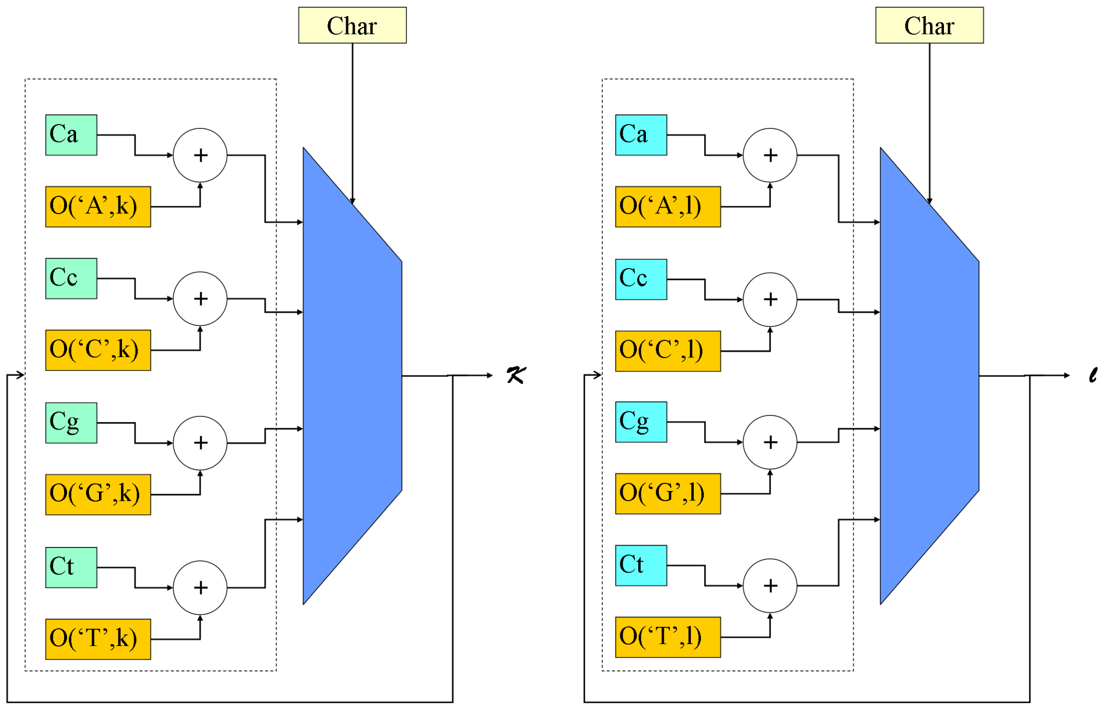


Figure 4.7: Exact matcher unit

Since each short-read can be mapped independently of another short-read, the exact matcher unit can be realized as many times as the FPGA can hold. With every extra exact matcher unit, the time to match all the short-reads will be cut in half. However, for the purpose of this thesis, only one such unit has been realized.

Chapter 5

Experimental Setup and Results

5.1 Experimental Setup

Our design was synthesized for the Stratix V GX 5SGXEA7N2F45C2 FPGA using Quartus II 14.0, and was simulated using ModelSim-Altera 10.3c. Modelsim is a Hardware Description Language (HDL) simulation environment for simulating different HDL languages, such as Verilog and VHDL. Quartus II is a design software that enables the design of various reconfigurable devices using HDL languages. Quartus II allows the user to use the analysis and synthesis tool for their design. The software has many features such as timing analysis as well as device configuration that enables the user to directly program their device by using Quartus.

The Stratic V GX 5SGXEA7N2F45C2 can contain up to 8GB of DDR3 memory; it also contains many other components such as various communication ports, and general input/ output ports. The Stratix V was chosen due to the high number of available resources, such as the number of ALMs that it contains. ALM stands for Adaptive Logic Module, and it consists of registers, adders, and the combinational logic that includes the LUT. The resource utilization of a device is usually given in terms of the number of ALMs used and as a percentage of the total device resources.

Our Register Transfer Logic code (RTL) was written in terms of the length of the genome and the length of the short-read. A test bench file is used along with ModelSim to simulate the code. The randomly generated short-reads are given in the test bench file, and the length of the genome was fixed to 1024 bp.

5.2 Results

Table 5.1 shows the resource utilization of the device with a short-read length of 32 bp, 64 bp, and 128 bp. Note that the only memory used in our design is the memory required to hold the short-read and the reference genome. The complexity of the hardware can be seen by the number of registers, as the registers can represent many things such as buffer registers and multiplexers. Also note the number of ALMs given in table 5.1 includes the number of registers.

Table 5.1: Resource Utilization

Length of short-reads (bp)	32	64	128
Registers	45,008	45,102	45,202
ALMs	68,703 (29%)	68,836 (29%)	68,947 (29%)
Memory bits	2,112	2,176	2,304

It can be seen that the hardware complexity does not change much as the length of the short-read changes, as we expected. This is because our exact matcher unit does not depend on the length of the short-read as it takes in one character at a time regardless of the length.

To be able to test the effect of increasing the number of short-reads on the time elapsed, we tested our code with 100 thousand, 500 thousand, 1 million and 5 million short-reads. Figure 5.1 shows the number of clock cycles as the number of short-reads varies. It can be seen that the number of clock cycles is linear with respect to the number of short-reads as expected. The maximum frequency that is achieved in our design is 125 MHz. Using the maximum frequency, and the number of clock cycles, the time elapsed can be calculated theoretically. Figure 5.2 shows the time elapsed as the number of short-reads increases.

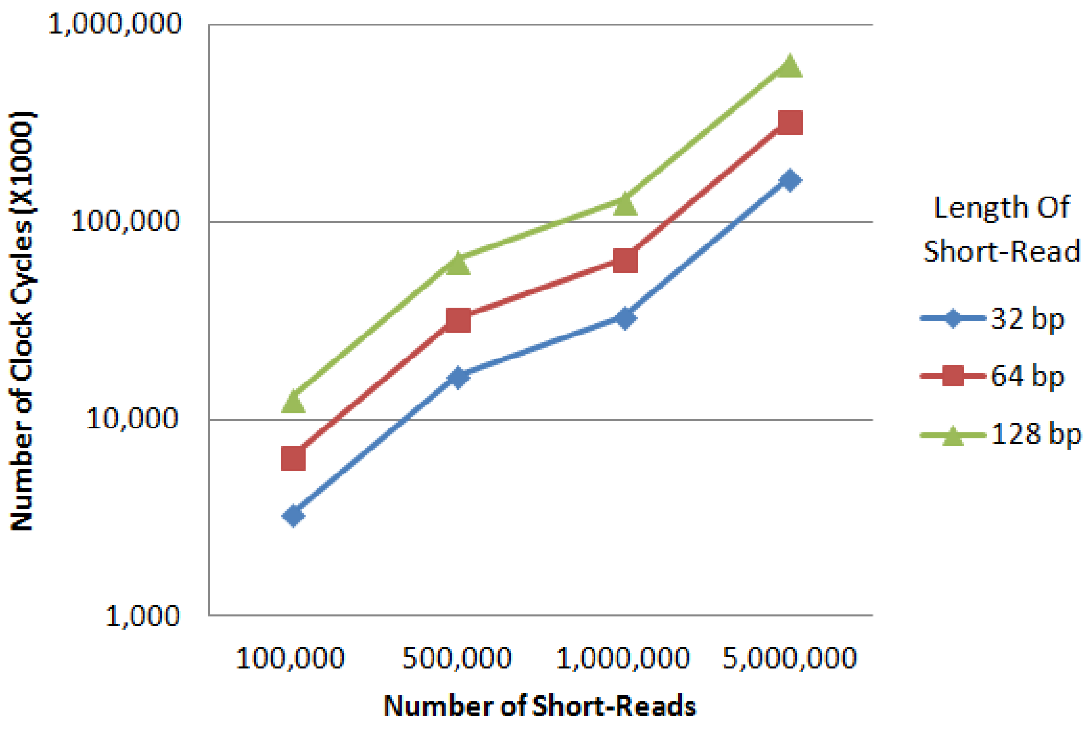


FIGURE 5.1: Performance in terms of the number of clock cycles as the length of the short-read increases

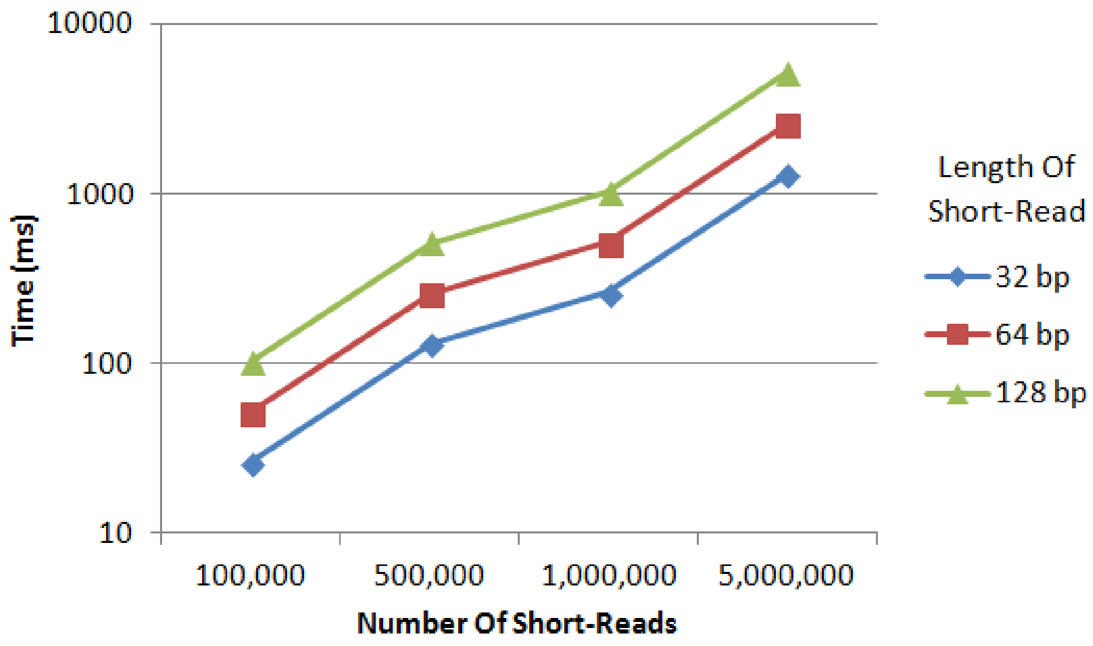


FIGURE 5.2: Time in ms as the number of short-reads vary

We compare our results with an exact match Burrows-Wheeler Aligner that runs on a core i7 with 8GB of memory. For a fair comparison, we used the same data for both the hardware and the software. Figure 5.3 shows the result of this comparison. It can be seen that our design runs on average 82X faster than the software version.

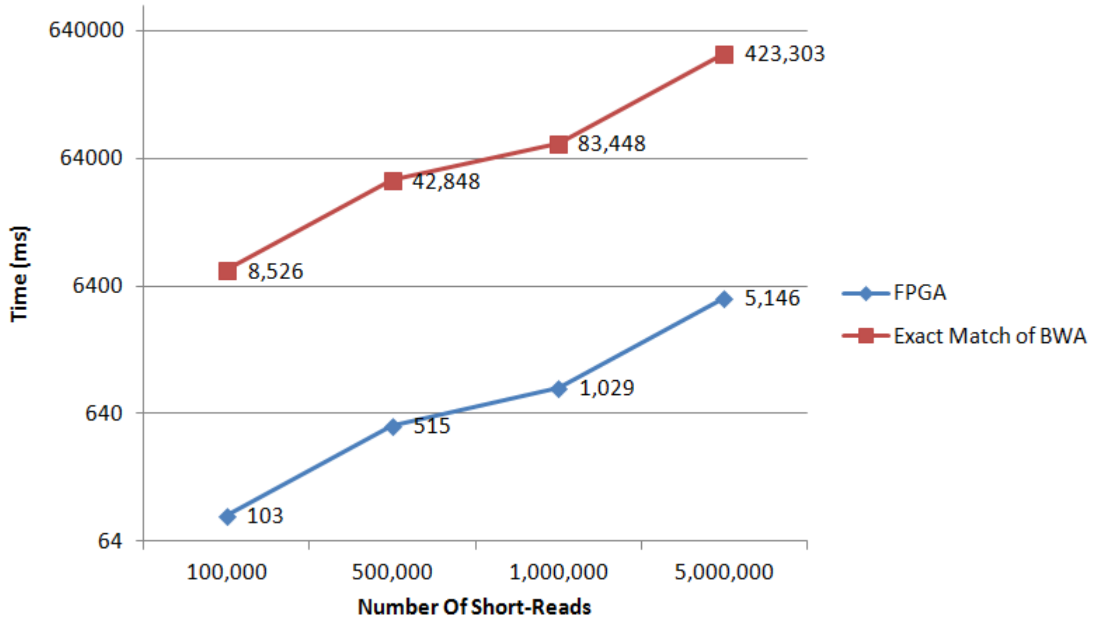


FIGURE 5.3: Performance comparison between exact match of BWA and FPGA with a short-read length of 128 bp

It is not easy to make a fair comparison with other hardware designs, as many different components need to be considered. However, we discuss our design in comparison with Shepard that is represented in [27] as it is an exact match short-read aligner.

Shepard [27] uses a Convey HC-1, which is a computing cluster that contains a standard motherboard, 32GB of memory and 14-FPGAs. They compare their results to Bowtie [8], and SOAP2 [7], and report to achieve a speedup of the order of hundreds of thousands. Their design is based on a hash-table that requires 23.3GB of memory for a Human Genome. However, such a design can not fit into a single FPGA, as a single FPGA can contain up to a maximum of 8GB of RAM. Whereas our design is compatible and can fit into almost any FPGA. It is also important to note that the current cost of a Convey HC-1 is $67,100 as this thesis is written, whereas a reasonable FPGA cost is considerably lower [27].

Chapter 6

Conclusion and Future Work

6.1 Conclusion

In this thesis, we demonstrated a high performance architecture for an exact match short-read aligner. We discussed the problem of aligning billions of short-reads to a large reference genome and realized that with the rapid development of DNA sequencing machines, faster short-read alignment tools are necessary.

Many software short-read alignment tools have been developed, but not many have been accelerated using hardware. We discussed two main methods for short-read alignment and came to the conclusion that suffix/prefix trie based methods require less memory than hash-table based methods. Burrows-Wheeler Aligner is a software alignment tool that has been very widely used and trusted for mapping short-reads. We concentrated on the exact match of the short-reads and chose to implement the exact match component of Burrows-Wheeler Aligner on an FPGA.

Our FPGA implementation proved to be remarkably faster than a CPU implementation. We ran experiments for up to 5 million short-reads and achieved a speedup of 82X compared to Burrows-Wheeler Aligner exact match. Our design can also be synthesized on different FPGAs.

6.2 Future Work

Our future work will concentrate on further improving the speed and flexibility of our design. One first approach would be to realize more than one exact matcher unit (refer to figure 6.1). Also, we can perform the occurrence array and c array calculations on the CPU and send the data to the FPGA memory, since they are calculated only once for a given reference genome. By removing the occurrence array unit and the C array unit, we could reduce the complexity of the hardware allowing more exact matcher units to be realized and, as a result, speeding the process. Another improvement to our design would be to allow for inexact matches as well as exact matches.

The maximum clock frequency for our design was 125.33MHz that was achieved on a Stratix V GX 5SGXEA7N2F45C2 FPGA, which can be further improved by optimizing our code and design. Our ultimate goal is to create a real-time system that can directly map the short-reads to a given genome.

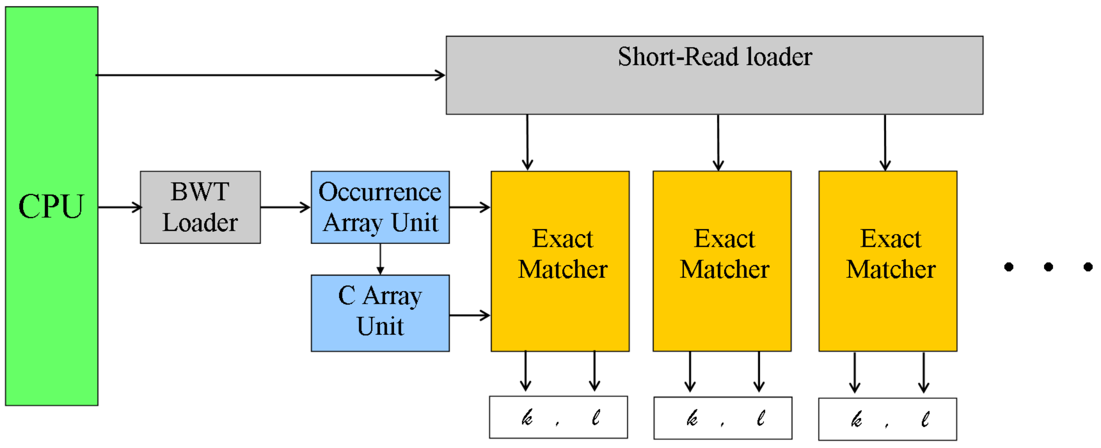


FIGURE 6.1: Future work

Bibliography

[1] Inc. Illumina. An introduction to next-generation sequencing technology. http://www.illumina.com/content/dam/illumina-marketing/documents/products/illumina_sequencing_introduction.pdf. Accessed: 10-03-2015.

[2] Wetterstrand KA. Dna sequencing costs: Data from the nhgri genome sequencing program (gsp). http://www.genome.gov/sequencingcosts. Accessed: 10-03-2015.

[3] Andrew D Smith, Zhenyu Xuan, and Michael Q Zhang. Using quality scores and longer reads improves accuracy of solexa read mapping. *BMC bioinformatics*, 9(1):128, 2008.

[4] Heng Li, Jue Ruan, and Richard Durbin. Mapping short dna sequencing reads and calling variants using. 2008.

[5] Ruiqiang Li, Yingrui Li, Karsten Kristiansen, and Jun Wang. Soap: short oligonucleotide alignment program. *Bioinformatics*, 24(5):713–714, 2008.

[6] Nils Homer, Barry Merriman, and Stanley F Nelson. Bfast: an alignment tool for large scale genome resequencing. *PloS one*, 4(11):e7767, 2009.

[7] Ruiqiang Li, Chang Yu, Yingrui Li, Tak-Wah Lam, Siu-Ming Yiu, Karsten Kristiansen, and Jun Wang. Soap2: an improved ultrafast tool for short read alignment. *Bioinformatics*, 25(15):1966–1967, 2009.

[8] Ben Langmead, Cole Trapnell, Mihai Pop, Steven L Salzberg, et al. Ultrafast and memory-efficient alignment of short dna sequences to the human genome. *Genome biol*, 10(3):R25, 2009.

[9] Heng Li and Richard Durbin. Fast and accurate short read alignment with burrows–wheeler transform. *Bioinformatics*, 25(14):1754–1760, 2009.

[10] Neil C Jones and Pavel Pevzner. *An introduction to bioinformatics algorithms*. MIT press, 2004.

[11] Lin Liu, Yinhu Li, Siliang Li, Ni Hu, Yimin He, Ray Pong, Danni Lin, Lihua Lu, and Maggie Law. Comparison of next-generation sequencing systems. *BioMed Research International*, 2012, 2012.

[12] ”Thermo Fisher Scientific Inc.”. Solid 4 system product description. https://products.appliedbiosystems.com/ab/en/US/adirect/ab?cmd=catNavigate2&catID=607061. Accessed October 5, 2015.

[13] Heng Li and Nils Homer. A survey of sequence alignment algorithms for next-generation sequencing. *Briefings in bioinformatics*, 11(5):473–483, 2010.

[14] Mohamed Ibrahim Abouelhoda, Stefan Kurtz, and Enno Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *Journal of Discrete Algorithms*, 2(1):53–86, 2004.

[15] Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. In *Foundations of Computer Science, 2000. Proceedings. 41st Annual Symposium on*, pages 390–398. IEEE, 2000.

[16] Altera Corporation. Implementing fpga design with the opencl standard. http://www.altera.com/en_US/pdfs/literature/wp/wp-01173-opencl.pdf. Accessed: 09-06-2015.

[17] Stephen F Altschull, W Gish, Webb Miller, Eugene W Myers, and David J Lipman. Basic local alignment search tool. *J. Mol. Biol*, 215:403–410, 1990.

[18] Stephen M Rumble, Phil Lacroute, Adrian V Dalca, Marc Fiume, Arend Sidow, and Michael Brudno. Shrimp: accurate mapping of short color-space reads. 2009.

[19] Michael C Schatz. Cloudburst: highly sensitive read mapping with mapreduce. *Bioinformatics*, 25(11):1363–1369, 2009.

[20] Yangho Chen, Tade Souaiaia, and Ting Chen. Perm: efficient mapping of short sequencing reads with periodic full sensitive spaced seeds. *Bioinformatics*, 25(19):2514–2521, 2009.

[21] Nathan L Clement, Quinn Snell, Mark J Clement, Peter C Hollenhorst, Jahnvi Purwar, Barbara J Graves, Bradley R Cairns, and W Evan Johnson. The gnumap algorithm: unbiased probabilistic mapping of oligonucleotides from next-generation sequencing. *Bioinformatics*, 26(1):38–45, 2010.

[22] Christian Otto, Peter F Stadler, and Steve Hoffmann. Fast and sensitive mapping of bisulfite-treated sequencing data. *Bioinformatics*, 28(13):1698–1704, 2012.

[23] Yongchao Liu, Bertil Schmidt, and Douglas L Maskell. Cushaw: a cuda compatible short read aligner to large genomes based on the burrows–wheeler transform. *Bioinformatics*, 28(14):1830–1837, 2012.

[24] Petr Klus, Simon Lam, Dag Lyberg, Ming S Cheung, Graham Pullan, Ian McFarlane, Giles SH Yeo, and Brian YH Lam. Barracuda-a fast short read sequence aligner using graphics processing units. *BMC research notes*, 5(1):27, 2012.

[25] Chi-Man Liu, Thomas Wong, Edward Wu, Ruibang Luo, Siu-Ming Yiu, Yingrui Li, Bingqiang Wang, Chang Yu, Xiaowen Chu, Kaiyong Zhao, et al. Soap3: ultra-fast gpu-based parallel alignment tool for short reads. *Bioinformatics*, 28(6):878–879, 2012.

[26] Michael C Schatz, Cole Trapnell, Arthur L Delcher, and Amitabh Varshney. High-throughput sequence alignment using graphics processing units. *BMC bioinformatics*, 8(1):474, 2007.

[27] Chad Nelson, Kevin Townsend, Bhavani Satyanarayana Rao, Phillip Jones, and Joseph Zambreno. Shepard: A fast exact match short read aligner. In *Formal Methods and Models for Codesign (MEMOCODE), 2012 10th IEEE/ACM International Conference on*, pages 91–94. IEEE, 2012.

[28] Corey B Olson, Maria Kim, Cooper Clauson, Boris Kogon, Carl Ebeling, Scott Hauck, and Walter L Ruzzo. Hardware acceleration of short read mapping. In *Field-Programmable Custom Computing Machines (FCCM), 2012 IEEE 20th Annual International Symposium on*, pages 161–168. IEEE, 2012.

[29] Server Kasap, Khaled Benkrid, and Ying Liu. High performance fpga-based core for blast sequence alignment with the two-hit method. In *BioInformatics and BioEngineering, 2008. BIBE 2008. 8th IEEE International Conference on*, pages 1–7. IEEE, 2008.

[30] Yusuke Sogabe and Tetsuhiro Maruyama. An acceleration method of short read mapping using fpga. In *Field-Programmable Technology (FPT), 2013 International Conference on*, pages 350–353. IEEE, 2013.

[31] Yaoliang Chen, Ji Hong, Wanyun Cui, Jacques Zaneveld, Wei Wang, Richard Gibbs, Yanghua Xiao, and Rui Chen. Cgap-align: a high performance dna short read alignment tool. *PloS one*, 8(4):04, 2013.

[32] Jing Zhang, Heshan Lin, Pavan Balaji, and Wu-chun Feng. Optimizing burrows-wheeler transform-based sequence alignment on multicore architectures. In *Cluster, Cloud and Grid Computing (CCGrid), 2013 13th IEEE/ACM International Symposium on*, pages 377–384. IEEE, 2013.

[33] Michael Burrows and David J Wheeler. A block-sorting lossless data compression algorithm. 1994.

[34] Genome Reference Consortium. Human genome overview. http://www.ncbi.nlm.nih.gov/projects/genome/assembly/grc/human/. Accessed: 10-03-2015.

[35] Wing-Kai Hon, Tak-Wah Lam, Kunihiko Sadakane, Wing-Kin Sung, and Siu-Ming Yiu. A space and time efficient algorithm for constructing compressed suffix arrays. *Algorithmica*, 48(1):23–36, 2007.

[36] Fahad Saeed. Pyro-align: Sample-align based multiple alignment system for pyrosequencing reads of large number. *arXiv preprint arXiv:0901.2751*, 2009.

[37] Fahad Saeed and Ashfaq Khokhar. Sample-align-d: A high performance multiple sequence alignment system using phylogenetic sampling and domain decomposition. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–9. IEEE, 2008.

[38] Fahad Saeed and Ashfaq Khokhar. A domain decomposition strategy for alignment of multiple biological sequences on multiprocessor platforms. *Journal of Parallel and Distributed Computing*, 69(7):666–677, 2009.