Hardware acceleration of short-read mapping with the Burrows-Wheeler Aligner

J.W. Peltenburg

Abstract

Over the last decade, the costs of sequencing a human DNA molecule have been drastically reduced. It is therefore expected that the DNA sequence of patients will be used more often as a diagnostics tool in future clinical applications. However, DNA sequencers deliver millions of small fragments from random positions of the very large DNA string. These fragments are called short-reads. To assemble the short-reads such that the original DNA sequence becomes known, various software tools called short-read mappers are available. The long run-times of these tools do not favor the large-scale deployment of DNA sequencing technologies in clinical applications. Therefore, the work presented in this thesis focuses on the analysis and acceleration of one widely used mapping tool, called the Burrows-Wheeler Aligner (BWA). This tool is used to align short-reads to a reference genome. The Convey Hybrid-Core computing platform is used to implement an FPGA-based design of the BWA aln step of the algorithm. The current design has the same functional capabilities as the software, and runs over multiple cores in parallel. The hardware run-time is still between 12X and 22X longer than the software run-time. This is mostly due to using a Depth-First-Search over Breadth-First-Search-approach in searching and the high memory latency of the platform. However, the design still allows for many improvements; the current design does not utilize all resources and it can still be pipelined. Furthermore, the scalability of the design is better than that of the multithreaded software implementation. It is estimated that, with such improvements, the design can obtain a speed-up of up to 13X over the software implementation.
Hardware acceleration of short-read mapping with
the Burrows-Wheeler Aligner

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

J.W. Peltenburg
born in Middelharnis, The Netherlands
Hardware acceleration of short-read mapping with the Burrows-Wheeler Aligner

by J.W. Peltenburg

Abstract

Over the last decade, the costs of sequencing a human DNA molecule have been drastically reduced. It is therefore expected that the DNA sequence of patients will be used more often as a diagnostics tool in future clinical applications. However, DNA sequencers deliver millions of small fragments from random positions of the very large DNA string. These fragments are called short-reads. To assemble the short-reads such that the original DNA sequence becomes known, various software tools called short-read mappers are available. The long run-times of these tools do not favor the large-scale deployment of DNA sequencing technologies in clinical applications. Therefore, the work presented in this thesis focuses on the analysis and acceleration of one widely used mapping tool, called the Burrows-Wheeler Aligner (BWA). This tool is used to align short-reads to a reference genome. The Convey Hybrid-Core computing platform is used to implement an FPGA-based design of the BWA aln step of the algorithm. The current design has the same functional capabilities as the software, and runs over multiple cores in parallel. The hardware run-time is still between 12X and 22X longer than the software run-time. This is mostly due to using a Depth-First-Search over Breadth-First-Search approach in searching and the high memory latency of the platform. However, the design still allows for many improvements; the current design does not utilize all resources and it can still be pipelined. Furthermore, the scalability of the design is better than that of the multithreaded software implementation. It is estimated that, with such improvements, the design can obtain a speed-up of up to 13X over the software implementation.

Laboratory: Computer Engineering
Codenumber: CE-MS-2014-09

Committee Members:

Advisor: Zaid Al-Ars, Computer Engineering, TU Delft
Chairperson: Koen Bertels, Computer Engineering, TU Delft
Member: Arjan van Genderen, Computer Engineering, TU Delft
Member: Rene van Leuken, Circuits and Systems, TU Delft
Dedicated to my grandfather
## Contents

List of Figures viii  
List of Tables ix  
Acknowledgements xi  

1 Introduction 1  

2 Background 3  
2.1 DNA ......................................................... 3  
   2.1.1 Overview of DNA ........................................ 3  
   2.1.2 DNA Sequencing ......................................... 4  
2.2 Short-read alignment ........................................ 6  
   2.2.1 Matches and mismatches ................................. 8  
   2.2.2 Gaps, insertions and deletions ......................... 8  
   2.2.3 Scoring ................................................. 9  
2.3 Existing aligners .......................................... 9  
   2.3.1 Software aligners ....................................... 10  
   2.3.2 GPU-based ............................................. 11  
   2.3.3 FPGA-based ............................................ 11  
2.4 Previous work ............................................. 12  

3 Analysis of the Burrows-Wheeler Aligner 13  
3.1 Burrows-Wheeler Transform ................................ 13  
3.2 FM-index .................................................. 14  
   3.2.1 The components of the FM-index ...................... 15  
   3.2.2 Exact matching using the FM-index ................. 16  
   3.2.3 Keeping the index small .............................. 18  
3.3 Burrows-Wheeler Aligner ................................ 20  
   3.3.1 Inexact matching .................................... 20  
   3.3.2 Limiting the search space ............................ 22  
   3.3.3 The inexact matching algorithm of BWA .......... 24  
3.4 Performance analysis .................................... 26  
   3.4.1 BWA function profile ................................. 26  
   3.4.2 BWA total run-time ................................... 28  
   3.4.3 Memory usage ........................................ 29  
   3.4.4 Discussion ............................................ 30
List of Figures

2.1 The molecular structure of DNA. ......................... 4
2.2 NHGRI Cost per genome .................................. 6
2.3 Concept of short-read alignment .......................... 7
2.4 A short-read without differences aligned to a reference ... 8
2.5 A short-read with one difference aligned to a reference ... 9
2.6 A short-read with an insertion aligned to a reference ... 9

3.1 All rotations of an input string and a sorted suffix array. ... 14
3.2 The suffix tree representation of a reference. ............... 15
3.3 Generation of the occurrence array. ........................ 15
3.4 All components of the FM-index of the example. .......... 16
3.5 Exact matching example in a suffix tree representation ... 18
3.6 Reduced occurrence array .................................. 19
3.7 An example of the different search paths of inexact matching on the suffix tree representation. ............ 22
3.8 The relative time spent in different function of BWA for synthetic reads. . 27
3.9 The relative time spent in different function of BWA for real reads. ... 27
3.10 Top 80 occurrence array access ............................ 29

4.1 Proposed top-level architecture ............................ 34
4.2 ALN core architecture ..................................... 35
4.3 Occurrence Obtain Unit ..................................... 36
4.4 Base-pair counter ........................................... 38
4.5 Design of the Calculate D unit ............................. 39
4.6 Design of the Inexact Matcher ............................... 40
4.7 Area usage .................................................. 44
4.8 Design for a pipelined Inexact Matcher. ............... 46

5.1 Graph legend ................................................ 52
5.2 Ungapped alignment, zero differences allowed. ............ 52
5.3 Ungapped alignment, two differences allowed. ............. 52
5.4 Ungapped alignment, four differences allowed. ............ 52
5.5 Seed length of 10, two differences allowed. ............... 54
5.6 Seed length of 10, four differences allowed. ............... 54
5.7 Seed length of 30, four differences allowed ............... 54
5.8 One gap open allowed, one difference allowed. ........... 56
5.9 One gap open allowed, two differences allowed. ........... 56
5.10 One gap open allowed, three differences allowed. ....... 56
5.11 Performance of aligning a subset of ERR000589 (51 bp reads), seed length 20, 3 mismatches, 1 open and 6 extensions allowed. ....... 57
5.12 Performance of aligning a subset of SRR062634(100 bp reads), seed length 30, 5 mismatches, 1 open and 6 extensions allowed. ....... 57
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>List of Tables</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>3.1</td>
<td>Exact matching example using the FM-index.</td>
</tr>
<tr>
<td>3.2</td>
<td>Index size of a full human genome</td>
</tr>
<tr>
<td>3.3</td>
<td>An example of the intervals to search when using inexact matching.</td>
</tr>
<tr>
<td>4.1</td>
<td>Example of how the position of interest is controlled for the occurrence obtain unit when the 75th base-pair in the interval is required.</td>
</tr>
<tr>
<td>4.2</td>
<td>Cycle times of a typical aln core iteration.</td>
</tr>
<tr>
<td>4.3</td>
<td>Area statistics for the design (grey is extrapolated data)</td>
</tr>
<tr>
<td>5.1</td>
<td>Software and hardware alignments comparison</td>
</tr>
<tr>
<td>5.2</td>
<td>Estimations for a pipelined implementation.</td>
</tr>
</tbody>
</table>
First of all I would like to thank professor Zaid Al-Ars for his truly unrivaled and contagious enthusiasm for the subject of bio-informatics. Thank you for allowing to spend plenty of time your time for our discussions and always making me leave with even more motivation to continue my work.

I would like to thank Giacomo Marchiori and Vlad-Mihai Sima of the Computer Engineering lab at the TU Delft, for helping me with the Convey implementation and for brainstorming about possible solutions. Thanks to Ies Nijman of the Department of Medical Genetics of the UMC in Utrecht, for providing me with real data-sets and for explaining some biology related parts of this work.

I thank Emile van de Logt heartily for his support, and for allowing me to spend so much time, next to my position as a lecturer at the Rotterdam University of Applied Science, to complete my studies and this work. I would also like to thank fellow student and colleague Wouter van Teijlingen for verbally sparring about my ideas on the subject of this work. I would like to thank all my other fellow students and colleagues as well, for their interest and support.

I thank my friends and family for their encouragements (and sometimes dragging me away from this work so that I could clear my mind). Last but not least, my thanks and love to Emma, for her unconditional support.

J.W. Peltenburg
Delft, The Netherlands
July 17, 2014
Introduction

The cells of every living organism contain the well known double-helix structured molecule called deoxyribonucleic acid, or DNA. Since its discovery halfway the previous century, biologists came to know that DNA encodes genetic traits and is therefore the source of several biological concepts, such as evolution, but also genetic diseases and cancer.

In medical biology and biochemistry, many advances were made regarding DNA research. On the one hand many clinical methods were developed that apply the knowledge of DNA to cure diseases. On the other hand, over recent years, the costs of sampling or sequencing DNA have been continuously lowered. Sequencing DNA has become so (relatively) inexpensive, that expectations are that, in the future, many hospitals and doctors will base their diagnoses and treatments - even for regular patients - on technologies involving DNA.

At the time of writing, modern methods of high-throughput DNA sequencing are called massive parallel sequencing or next-generation-sequencing (NGS). Depending on the technology, modern NGS machines take between half a day and a few days to obtain the genetic information from a human sample. The information that is the result of sequencing is, however, not one full sequence of someone’s DNA. Sequencing results in so called ”short-reads”, which are only extremely small fragments of the full DNA.

Considering the human genome contains around 3.2 gigabase-pairs (Gbp\(^1\))[1]. The length of these fragments, for modern machines, is only between tens of base-pairs up to a thousand of base-pairs, depending on the type of machine used and how it was used [2]. Thus, if one wants to know the full genome of a subject, while only in the possession of many of these short-reads, it is necessary to assemble the short-reads into a full genome.

This can happen in two ways:

- We try to obtain so many short-reads that we may calculate where they overlap and eventually come up with a full genome. This is called de novo assembly. Many overlapping short-reads are needed to obtain a high quality assembled genome.

- Since between two humans there is only about 0.4% difference in base-pairs [3], we may use a reference genome to map the short-reads onto. This is called short-read mapping or short-read alignment. Not as many overlapping short-reads are needed as for de novo assembly, which saves time during sequencing and assembly.

This report focuses on the latter method: short-read alignment.

Because there are many short-reads that may be (somewhat) independently mapped to

\(^1\)In biology, this number of base-pairs is often written as 3.2 Gb, but to prevent confusion with gigabits, we write Gbp.
a reference, short-read alignment applications already exhibit much parallelism at the
input. This report will focus on one particular application called the Burrows-Wheeler
Alignment tool (BWA)[4]. BWA is a tool that is extremely well known and used
amongst researchers who work on applications involving DNA.

Goal
With the availability of a FPGA-based high-performance computing platform from Con-
vey Computer (which has been shown to improve the run-time of many bio-informatics
applications) the goal of this thesis is to explore and analyze the BWA application, iden-
tify areas suited for speed-up, and create and implement a basic design on the Convey
HC-2 platform.

Contents of this thesis
The contents of this report is as follows.
In Chapter 2 the general concept of short-read alignment for DNA molecules is dis-
cussed, and several existing solutions are mentioned.
In Chapter 3 a thorough analysis of the algorithms that BWA uses, such as the
Burrows-Wheeler Transform and the FM-index, are presented. Furthermore, several
proposals are made that may speed up the application.
In Chapter 4, the architecture and detailed design of the FPGA implementation of
BWA are presented.
In Chapter 5, several results and comparisons to the software implementation are
made.
Finally, Chapter 6 concludes this work and gives recommendations for further re-
search.
In this chapter, relevant subjects concerning the background of this work are discussed. In Section 2.1, the properties of a DNA molecule are shown. Following in Section 2.2 is an overview of short-read alignment, which is the main function of BWA. In Section 2.3, existing software, GPU-based and hardware aligners are mentioned. Finally, in Section 2.4, previous work to accelerate BWA is discussed.

2.1 DNA

2.1.1 Overview of DNA

DNA (see Figure 2.1) consists of two strands of a sugar-phosphate backbone, with nitrogenous bases attached [5]. One such combined sugar, phosphate and nitrogenous based molecule is often called a nucleotide. The bases along a strand are held together by relatively strong covalent bonds, while opposite bases of the strands are held together by a weaker hydrogen bond. This effectively attaches the two strands to each-other. A pair of two of such nitrogenous bases that are attached are called a base-pair.

For human DNA, there are four types of bases made of substances called Adenine, Thymine, Guanine and Cytosine. An Adenine base on one strand, is always attached to a Thymine base on the other strand. The same goes for Guanine and Cytosine. This causes the two strands to be anti-parallel to each other, or it is said that the one strand is the reverse complement to the other. The importance of this property, related to this work, will become clear in Chapter 3. The molecular structure of DNA is shown in Figure 2.1.

To replicate DNA, the hydrogen bonds are broken and the strands become separated, forming two new template strands. By a biochemical process, the strand that is missing is replicated on each separated strand, and two new DNA molecules are formed.

The specific sequence of these base-pairs determine every genetic trait of an organism. For humans, the DNA encodes, for example, the color of someone’s eyes or whether someone is susceptible to heart diseases. While replicating, or under external influences (for example UV-light on skin cells), the DNA sequence may change (or mutate), although this is a relatively rare event. To an organism, these mutations may have a positive effect (such as causing the species to evolve) or a negative effect, such as errors in the DNA that cause uncontrolled growth, which causes cancer.

At many layers of biology and medicine, it is therefore extremely relevant to study the DNA molecule of a subject.
2.1.2 DNA Sequencing

2.1.2.1 Methods

To obtain the specific sequence of Adenine, Thymine, Cytosine and Guanine in a DNA molecule, many methods have been developed over the years, where different methods used in Next Generation Sequencing applications ([6] [7]) are available. Most methods are based on adding specific enzymes that emit light when polymerization occurs (which is the process where new nucleotides are added to a template strand). Each type of nucleotide is tagged with a different color. Several methods exist:

- **Pyrosequencing** involving luciferase; techniques where enzyme based reactions generate light.

- **Sequencing by Ligation** also involving labeling the sample by using enzymes and fluorescent dyes.

- **Sequencing by Synthesis** using specific enzymes that also emit light when added, but they block further polymerization and can be removed again, to repeat the process.
The above methods then take images of the substance at a high rate, so to detect whenever a new nucleotide is added. At a certain position in the image, where one (part of) a DNA molecule resides, a sequence of different colors will be detected, and the sequence becomes known.

Some more recent methods involve:

- **Ion Semiconductor sequencing** where protons that are released when new nucleotides are formed are eventually detected by a silicon chip.

- **Single Molecule Real-Time sequencing** (SMRT) using specific nano-structure arrays for light detection [8] and a different type of labelled nucleotides. It allows for *real-time* sequencing, in this case meaning that it allows the DNA to replicate at its normal, natural speed, which is much faster than with the methods described above.

The input of such sequencers is often a ‘library’ of the DNA, which is a collection of many cloned DNA molecules. All sequencers and techniques have the characteristic that their output is not one string of the base-pairs full genome, but rather millions of tiny parts of the genome, often called *reads*. These reads may have different sizes, depending on the technology.

Pyrosequencing and SMRT allow for relatively long read lengths (700 and up to 1500 base-pairs, respectively). Most other technologies allow only for much shorter read lengths. Depending on many parameters, these reads are between 30 and 200 base-pairs long.

When reads are obtained, bases are determine often one by one for a single read, as the nucleotides are added during synthesis of the DNA. This process is called ‘base calling’. During base-calling, errors may occur. DNA sequencers will not only give the base letter A,T,G or C, corresponding to the base-pair type Adenine, Thymine, Guanine and Cytosine, but also a *quality score* for the base call. This means that for every base-pair that the sequencer has called, a probability is given that the base call was correct.

### 2.1.2.2 Costs

One thing that drives DNA research forward very rapidly, are the decreasing costs of sequencing. A very well known graph that illustrates this is that of the U.S. based National Institute of Health concerning the costs per genome, which is shown in Figure 2.2.

Because of the decreasing costs, it is expected that DNA analysis and technology will steadily introduce itself into regular medicine, introducing many benefits for healthcare.

However, there are many technical challenges to be overcome, especially in computing, due to the enormous amounts of data that have to be processed and stored when using these technologies.
2.2 Short-read alignment

Since this thesis focuses on clinical applications that benefit human healthcare, only the human DNA is considered. The human DNA consists of around 3.2 gigabase-pairs, which is many orders of magnitude larger than the short-reads produced by the sequencing technologies described in the previous sections. This means that to reconstruct the full genome of a human subject involves either:

- **De novo assembly**: obtain so many overlapping reads that their relative positions can be calculated.

- **Short-read alignment**: since the differences between two human genomes is extremely small, we search for possible locations of the short-read in a reference genome.

For both methods, enough coverage is needed for a high-quality reconstruction of the sample genome. That is, at one nucleotide position we need not only one read, but many reads, to make sure the right base-pair is eventually determined at that position. This is due to factors described previously, including (and most likely) wrong base calls made by sequencers. The coverage is the average number of times that a nucleotide at some position is sequenced [10].

For the first method, called *de novo* assembly, more computation and/or data is needed. For the second method, called *short-read alignment* less computation and data
is needed, but a good quality can still be obtained. When less reads are required, the sequencing will be completed faster. Combining the fact that the genome of any two humans differs by approximately 0.4% only [3], this is the reason why many researchers use short-read alignment to determine the genome of a subject.

This thesis will continue to focus on short-read alignment, for which the concept is shown in Figure 2.3

![Figure 2.3](image)

Figure 2.3: The concept of short-read alignment is to find the most suitable location of a short-read in a reference genome.

Many researchers will use the same reference genome. There are several popular version. One example is the reference compiled by the Genome Reference Consortium, consisting of several institutions, such as the National Center for Biotechnology Information in the United States, and the European Bioinformatics Institute. The consortium updates the reference genome once in several years, and releases new versions of it. At the time of writing, the latest version is GRCh38 [1].

Depending on several factors, the coverage that is needed for a good quality assembled genome using short-reads of about 50 base-pairs long, is about 20X. This means at one position, we would like to have about 20 short-reads overlapping. Taking into account the length of the human genome, 20X coverage with 50 base-pair reads, we need to process almost 1.3 billion reads.

Observe that:

- Many short-reads have to be aligned to the reference.
- The reference genome is extremely big.
- The short-reads may contain mutations or base calling errors compared to the reference.

This makes finding the best location of a short-read of only tens of base-pairs long a difficult computational problem. These three properties make short-read alignment a difficult computational problem.
2.2.1 Matches and mismatches

Before several methods to align the short-reads are discussed, it is important to look into the specifics of a short-read compared to a genome.

From this point on, the reference may be referred to as $G$ of size $|G|$, the short-read may be referred to as $R$ of size $|R|$. The base-pair types are referred to as characters enclosed by single quotation marks: '\('A'\)', '\('C'\)', '\('G'\)' and '\('T'\)', and a sequence of base-pairs is denoted as a string of those characters enclosed by double quotation marks, for example; "CAT".

Consider, for illustrative purposes, a reference $G = "GATTACA"$ and a short-read without any mutations or base calling errors, $R_0 = "TAC"$. The goal of short-read alignment is now to find a position for $R_0$ such that it is correctly placed with respect to the rest of the genome at position 3. This is seen in Figure 2.4.

<table>
<thead>
<tr>
<th>position</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>reference</td>
<td>G</td>
<td>A</td>
<td>T</td>
<td>T</td>
<td>A</td>
<td>C</td>
<td>A</td>
</tr>
<tr>
<td>short-read</td>
<td>T</td>
<td>A</td>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 2.4: A short-read without differences aligned to a reference

Here, we can see that "TAC" aligns with the reference from position 3 to 5. In the case of aligning the given $R_0$ with $G$, the problem is relatively simple, we just search $G$ for an exact occurrence of $R_0$. This is called exact matching. If a short-read can be aligned to the reference, it is said that we have a hit for this short-read.

However, the short-read might contain a base-call error or the subjects DNA might even contain a mutation with respect to the reference, which is something DNA researchers are specifically interested in, since they can be the cause of diseases. When a single base-pair changes, in genetics, this is called Single-Nucleotide-Polymorphism (SNP).

This way, consider a $R_1 = "TGCA"$, where the base-pair 'A' in $R_0$ changed to a base-pair 'G'. A short-read alignment algorithm must still report the best position for the alignment, which is still between position 3 to 5. It is an alignment, containing one mismatch. Thus, if we allow one mismatch to occur in the alignment, we can still map the short-read $T_1$ to the reference genome $G$, i.e. we still have a hit. The process of mapping the short-read to a reference, allowing mismatches, is called inexact matching. This is seen in Figure 2.5.

2.2.2 Gaps, insertions and deletions

Consider a third read $R_2 = "TAGC"$. During short-read alignment, if we still only allow one difference, we will not get a hit for $R_2$. What may have happened is that an extra nucleotide has been added during the synthesis of the DNA molecule, and that the 'G'
2.3. EXISTING ALIGNERS

<table>
<thead>
<tr>
<th>position</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>reference</td>
<td>G</td>
<td>A</td>
<td>T</td>
<td>T</td>
<td>A</td>
<td>C</td>
<td>A</td>
</tr>
<tr>
<td>short-read</td>
<td>T</td>
<td>G</td>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mismatch</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>↑</td>
<td></td>
</tr>
</tbody>
</table>

Figure 2.5: A short-read with one difference aligned to a reference

In the read is actually a so called *insertion* of a base-pair in the genome. The correct alignment may therefore be as seen in Figure 2.6.

<table>
<thead>
<tr>
<th>position</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>reference</td>
<td>G</td>
<td>A</td>
<td>T</td>
<td>T</td>
<td>A</td>
<td>C</td>
<td>A</td>
</tr>
<tr>
<td>short-read</td>
<td>T</td>
<td>A</td>
<td>G</td>
<td>C</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>insertion</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>↑</td>
<td></td>
</tr>
</tbody>
</table>

Figure 2.6: A short-read with an insertion aligned to a reference

In this case, it is said that the extra base-pair ‘G’ in the short-read is either an *insertion* to the reference, or that the reference contains a *deletion* with respect to the short-read. Insertions and deletions are colloquially called *indels* or *gaps*.

When indels occur, the gap may not only be one base-pair wide, but may continue for several base-pairs.

2.2.3 Scoring

When a short-read is aligned to a reference, the alignment may thus contain one or more of the following:

- Base-pairs that match the reference.
- Base-pairs that mismatch the reference.
- Insertions or deletions.
- An extension of an insertion or deletion.

Because errors in the reads are relatively rare, DNA researchers often calculate an alignment score based on the sum of specific penalties for mismatches, indels, and extensions of indels. Thus, an alignment with a lower score is generally a ‘better’ hit than one with a higher score.

2.3 Existing aligners

Due to the decreasing costs of genome sequencing, many short-read alignment software tools have appeared by various researchers. For clarity; this thesis focuses only on
short-read alignment (up to 200 base-pairs long). There are many other tools for the alignment of much longer parts to a reference, but these are outside the scope of this work.

Most short-read alignment tools are targeted for regular PC platforms that researchers might have available in laboratories. There are a limited number of implementations on GPU’s and FPGA based computing platforms. An overview will be given in the next section.

2.3.1 Software aligners

Software aligners that are based on hashing the reads or the genome are:

- **Eland** is one of the first short-read aligners, and propriety software by Illumina, performs ungapped alignment only on 32 bp reads.

- **MAQ** is an old aligner that uses quality scores and supports ungapped alignment.

- **ZOOM** [11] is an old short-read aligner that only allows a limited number of mismatches.

- **SHRiMP** [12] originally indexed the reads, but the latest version indexes the genome. SHRiMP uses a lot of memory (48GiB) and SHRiMP is not maintained any more.

- **mrFAST/mrsFAST** [13] for respectively gapped and ungapped alignment. This short-read aligner uses a seed-and-extend strategy. It implements a cache-oblivious algorithm to optimize the use on regular general purpose platforms.

A seed-and-extend strategy, in general, is a method where the aligner first determines several *candidate locations* for a short-read, and then continues the mapping with a different algorithm, such as the Smith-Waterman algorithm. The Smith-Waterman algorithm [14] is a dynamic programming algorithm to score alignments of DNA, that is widely used in aligners for long parts of the genome as well.

There are also aligners that are based on using suffix trees and suffix arrays.

- **Bowtie & Bowtie2** [15] [16] are geared more towards shorter (up to 50bp) short-reads and ungapped alignment, while Bowtie 2 allows gapped alignment and works better for longer short-reads. Bowtie was one of the first aligners to use the FM-index.

The FM-index itself is built on the Burrows-Wheeler Transform (BWT), which is a specific rearrangement of the reference, originally used to increase the effectiveness of compression algorithms. The method creates a relatively small, but searchable index of the reference. When searching the FM-index, this effectively mimics a backwards-search in a suffix-trie. Because the FM-index is heavily involved in this work, a thorough explanation will be given in Chapter 3.
2.3. EXISTING ALIGNERS

- **BWA** [4] [17] is an aligner that internally contains different methods to align reads. Originally it performed gapped alignment based on inexact matching with the FM-index, but now also has algorithms that find seeds using the FM-index and, after filtering, extend the seeds using the Smith-Waterman algorithm.

- **segemehl** [18] is a more recent application that uses a specific suffix array representation to find seeds in, and extends the seeds to find gaps and mismatches with a dynamic programming method called the Meyers bit-vector algorithm. The algorithm also uses significantly more memory than BWA and Bowtie.

- **SOAP2** [19] uses a combination of the FM-index and hashing to increase the speed, but uses significantly more memory than BWA and Bowtie.

2.3.2 GPU-based

- **BarraCUDA** [20] which is a CUDA-enabled GPU implementation of BWA. It reports speed-ups of around 3X compared to the multi-threaded version of BWA.

- **CUSHAW** [21] is also a CUDA-enabled GPU implementation based on the FM-index. It reports speed-ups of around 6X compared to the multi-threaded version of BWA, only it does not allow gapped alignments.

- **SOAP3/GPU** [22] is a CUDA-enabled GPU implementation of SOAP. It reports speed-ups of up to 7.5X over BWA, but does not support gapped alignments.

2.3.3 FPGA-based

- **Shepard** [23] is a hashing based FPGA implementation on the Convey Computer’s high-performance FPGA based platform, the Convey HC1. However, it only supports exact matching. It reports extremely high speed-ups compared to other tools such as BWA, Bowtie and MAQ, but in the reported benchmark it is shown to align only a quarter of the reads. It uses a high amount of memory, which is available on the Convey HC platform. Because this work also involves the Convey HC platform, a more in depth overview of it is given in Chapter 4.

- **Arram et al.** [24] show an FPGA implementation of a seed-and-extend algorithm, achieving a speed up of over 200X. However, it does not compare its implementation to the seed-and-extend version of BWA, and runs BWA with 20 threads, which is not optimal for the used platform. Still, the implementation shows a reconfigurable design, where first only exact matching units are implemented. When enough short-reads that cannot be mapped without allowing differences are stored, the design reconfigures the FPGA to allow inexact matching units to start processing the short-reads that need differences allowed.

- **Xin et al.** [25] propose a hardware architecture that for one core processes two base-pairs at a time, but the index is kept in BRAM, effectively making their implementation only useful to align reads to extremely small genomes, such as for bacteria. In the next chapters, it will be explained that the memory and the
interface to the memory are key components not to overlook in designing a short-read aligner.

**Selection of the candidate application for acceleration**

Of all aligners, BWA and Bowtie are the most widely used and highly cited tools and have become *de facto* standards for short-read alignment. DNA researchers have trust in these tools, and know that if alignments are done with these tools, that they are of a high quality. It is therefore decided that analysing one of either one of the tools for potential speed-up might be beneficial to the implementation of personal genomics in the future.

Because BWA supports gapped alignment and initially has a higher flexibility than Bowtie, this work will focus on exploring possible acceleration of BWA using FPGA-based platforms. Furthermore, because both tools make use of the FM-index, it is expected that by researching and developing one implementation, insight is also given in potential speed-up of the other tool.

2.4 Previous work

There are a few works that have focused on accelerating BWA that will be discussed here.

In the work of Zhang et al. [26], a method to improve BWA in software only, by decreasing the amount of cache misses is shown. These improvements cause BWA to be accelerated around 1.3X.

CGAP align [27] is an adjusted version of BWA, where researchers included an extension to the suffix array data structure in BWA, which accelerates BWA around 1.2X.

In parallel to this work, Waidyasooriya et al. [28] showed a hardware accelerated version of BWA of which they claim to obtain a speed-up of around 10X. However, they only use a reference of a few thousands base-pairs. It will be explained in the next chapter that using a small reference results in a considerably lower run-time than when using a large reference.
Analysis of the Burrows-Wheeler Aligner

This chapter will focus on how the Burrows-Wheeler Alignment tool aligns reads. Some specific algorithms and data structures that BWA uses, called the Burrows-Wheeler Transform and the FM-index are explained in Section 3.1 and Section 3.2, respectively. Following, the specifics about BWA are discussed in Section 3.3. Finally, a performance analysis is given in Section 3.4 that discusses potential areas for speed-up.

It is important to note that BWA contains an algorithm for indexing and three main algorithms for alignment, but that this thesis focuses solely on the original algorithm that is sometimes called BWA backtrack or on the command line ‘bwa aln’. Furthermore, BWA requires an additional step after alignment, to generate output readable to downstream tools. These steps are called samse/samse. However, the scope of this work is to focus only on the aln algorithm. Therefore whenever BWA is mentioned henceforth, only ‘bwa aln’ is considered.

3.1 Burrows-Wheeler Transform

The Burrows-Wheeler Transform (BWT) which was first presented in [29], is an algorithm to rearrange a string in such a way that it becomes easier to compress. The rearrangement is done in such a way, that the transformed string shows longer runs of similar characters, and thus can be easily compressed using move-to-front encoding and run-length encoding.

Before discussing how BWA uses the BWT, the algorithm will be explained using several examples. The following symbols and values for the examples are used:

- $\Sigma = \{'A', 'C', 'G', 'T'\}$: the alphabet over which the string is built.
- $\$: a special character that is lexicographically smaller than all other characters in $\Sigma$.
- $G' = "GATGACCA"$: the input string.

The first step of the algorithm is to add the special character $\$ to the input string $G'$. Thus, we define $G = G'\$ = "GATGACCA\$"$ to be the input string to which the special character is added. This character is needed later to be able to inverse the Burrows-Wheeler Transform.

To determine the BWT, all rotations of $G$ are generated and placed in a conceptual array. This causes all possible suffixes of the string $G$ to appear in the array. This results in the leftmost array shown in Figure 3.1. The index of this array is shown as $i$. 

13
CHAPTER 3. ANALYSIS OF THE BURROWS-WHEELER ALIGNER

Since all suffixes of the string are in the array, suppose one would search for a read $R$ which might for example be "CCA", it would be useful to sort the suffixes in the array, so that we may quickly find the interval in the array where the substring will reside. To find if the reference contains the substring, one may parse each character one by one, as long as there is still a valid interval in the suffix array containing that character.

Although this was originally not the main reason for sorting in the BWT, this property is utilized in the FM-index, which will be explained in the next section.

The Burrows-Wheeler Transform now takes the last column of the sorted array. The is called the BWT string $B$. This is the output of the algorithm. It was shown by Burrows and Wheeler [29] that there exists a method to perform a reverse transformation, such that the original string $G$ is obtained. Moreover, the BWT string $B$ is easier to compress, because it has the tendency to group similar characters together. Thus, the BWT is an algorithm of which the benefits are exploited in well known compression tools such as bzip.

3.2 FM-index

Ferragina and Manzini [30] built upon the idea of the BWT and by slightly extending the data structure, developed an algorithm that allows the compressed string to be effectively searched, without completely decompressing it.

Consider the sorted suffix array, in our example the rightmost array of Figure 3.1. Here, $i$ is now the index of the sorted suffix array, while $S[i]$ is the index of that suffix in the non-sorted array. Through $S$, we may find the original position of any $i$ of the sorted suffix array, effectively finding the absolute position of some suffix in the reference $G$.

Observe that the sorted suffix array (from now on: suffix array) is a representation of a suffix tree. A suffix tree can be used to quickly find any substring in the suffix tree representation of that string. The corresponding suffix tree of the example reference $G$ is shown in Figure 3.2.

For example, to find $R = "CCA", in the suffix array we would look up the interval of the first character 'C', and we arrive at the interval $[4, 5]$. This is the same as traversing from the root node to the node via the edge labeled 'C'. Then, consider the second character, which is also 'C'. Using the suffix array, we look in the interval $[4, 5]$ where
the second character is ‘C’ and find the interval [5, 5]. This is the same as traversing over the next edge that is labeled ‘C’. When parsing the last character ‘A’, the final interval [5, 5] is found, similar to traversing the edge ‘A’, arriving at a leaf of the tree.

One drawback of storing a suffix tree in memory, even when optimizing it for genomes, still takes a great amount of memory, of around 15 bytes per base-pair [31]. This would amount to more than 50 GiB of data for a full human genome.

However, since the BWT is the last column of a sorted suffix array, and this suffix array represents a suffix tree, Ferragina and Manzini discovered a method to search the reference \( G \) for a substring \( R \) by using the BWT string \( B \) and some additional data structures, of which one is called the occurrence array.

### 3.2.1 The components of the FM-index

The occurrence array is generated from the BWT string, by counting the occurrence of each of the characters in the alphabet \( \Sigma \) up to each position of the BWT string \( B \) independently. The occurrence array of the example is seen in Figure 3.3.

![Figure 3.3: Generation of the occurrence array.](image-url)
One extra but small data structure that is needed is the array \( C \) the size of \( \Sigma \), that is the count of characters in the BWT (or reference) that are lexicographically smaller than the previous character of the alphabet \( \Sigma \) in the array (excluding the special character $).

All elements of the FM-index of the example where \( G = "GATGACCA$" \) are shown in Figure 3.4. These are:

- \( B \), the Burrows-Wheeler Transform of \( G \).
- \( O \), the occurrence array
- \( C \), where \( C[c'] \) is the number of characters in \( G \) that are lexograpically smaller than some character \('c' \) itself.
- \( S \), to be able to obtain the original position in \( G \) from a position in the suffix array.

![Table](image)

Figure 3.4: All components of the FM-index of the example.

The occurrence array shown in Figure 3.4 implies that the index will grow by the length of \( G \) times the number of characters in the alphabet, but a method to compress this array is shown further on in this chapter.

### 3.2.2 Exact matching using the FM-index

Instead of searching in a forward manner by traversing edges of the suffix tree or by finding the next suffix array interval as was done in the previous example, Ferragina and Manzini have proved that the FM-index allows for a backward search in the suffix array according to Algorithm 1.

For clarity, for the algorithms illustrated in this work, it is assumed that the read \( R \) has \(|R|\) characters where the first character has index 0 and the last character has
index \(|R| - 1\), *idem ditto* for the reference \(G\). Furthermore, any occurrence values with an index lower than zero are defined as zero.

### Algorithm 1 Exact matching using the FM-index

```plaintext
function ExactMatch\((R, C, O)\)
  
  \(i \leftarrow |R| - 1\)
  \(k \leftarrow 0\)
  \(l \leftarrow |G| - 1\)
  
  while \(k \leq l \cap i \geq 0\) do
    \(\sigma \leftarrow R[i]\)
    \(k \leftarrow C[\sigma] + O[\sigma, k - 1] + 1\)
    \(l \leftarrow C[\sigma] + O[\sigma, l]\)
    \(i \leftarrow i - 1\)
  end while
  
  if \(k \leq l\) then
    return \(\{k, l\}\)
  else
    return \(\{\phi\}\)
  end if

end function
```

If the algorithm is applied to the example genome, trying to map a read \(R = "CCA"\), the steps are as shown in Table 3.1. Again, this is conceptually similar to traversing the suffix tree shown in Figure 3.5.

### Table 3.1: Exact matching example using the FM-index. The short-read \(R = "CCA"\)

<table>
<thead>
<tr>
<th>Iteration</th>
<th>(i)</th>
<th>(\sigma)</th>
<th>(k)</th>
<th>(l)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Init</td>
<td>2</td>
<td>'A'</td>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>'A'</td>
<td>(C[^{\prime}A'] + O[^{\prime}A', -1] + 1 = 1)</td>
<td>(C[^{\prime}A'] + O[^{\prime}A', 8] = 3)</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>'C'</td>
<td>(C[^{\prime}C'] + O[^{\prime}C', 0] + 1 = 4)</td>
<td>(C[^{\prime}C'] + O[^{\prime}C', 3] = 4)</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>'C'</td>
<td>(C[^{\prime}C'] + O[^{\prime}C', 3] + 1 = 5)</td>
<td>(C[^{\prime}C'] + O[^{\prime}C', 4] = 5)</td>
</tr>
<tr>
<td>Return:</td>
<td></td>
<td></td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>

The following points are important to observe:

- The time-complexity of the algorithm is \(O(|R|)\). The fact that the time-complexity does not depend on the size of the reference \(G\) is extremely advantageous for searching large references such as the human genome.

- The path of the backward-search does not follow the edges of the suffix tree. It more or less jumps to regions in the suffix array for which the partial suffix exists. This has implications for the acceleration of the algorithm later on.
CHAPTER 3. ANALYSIS OF THE BURROWS-WHEELER ALIGNER

3.2.3 Keeping the index small

One drawback of the index shown in Figure 3.4 is the size. Even though the BWT implicitly holds a suffix array of the reference, the occurrence array is of size $|\Sigma||G|$. Suppose the integers are stored in a 32-bit format to allow a full human genome to be indexed, the size of just the occurrence array would be almost 400 GiB. Since Ferragina and Manzini were interested in achieving a high rate of compression, they compress the BWT string $B$, the occurrence array $O$ and the pointers to the start of the suffixes in the original reference, $S$, using several methods.

BWA diverges somewhat from the approach of Ferragina and Manzini on this point, because the author managed to fit the human genome in a reasonably sized (under 4GiB) data structure without the need for further compression. This is one of the reasons aligners such as BWA and Bowtie became popular. Their indexes fitted in regular desktop PC’s, and thus searching was much faster compared to methods that had to access a disk drive more often. This enabled researchers to initially use their regular workstations to align their short-reads within reasonable time.

The BWT string $B$ is stored in a 2-bits-per-base-pair format, and the special character $\$ is skipped in storage. For a human genome, the size of $B$ is therefore around 750 MiB. Note that the original purpose of the BWT was to make it easier to compress using run-length and move-to-front encoding, but these methods are not used in the adjusted FM-index of BWA at all.

The occurrence array $O$ is stored only in intervals of 128 base-pairs. We call this the reduced occurrence array or symbolically $O^*$. An example with an interval of 3 base-pairs

Figure 3.5: Exact matching example in a suffix tree representation. The backward search path is denoted by the green lines.
is given in Figure 3.6.

\[
\begin{array}{cccc}
| \text{c} | \text{i} | O^* & O^* \\
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>C</td>
<td>G</td>
<td>T</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>
\end{array}
\]

Figure 3.6: Reduced occurrence array

For the human genome \(O^*\) takes around 380 MiB of storage. Whenever an occurrence value is needed, the value of the corresponding interval in which the value resides is loaded, and the rest of the occurrence value is calculated on-the-fly by using the BWT string \(B\) itself.

For the example index used above, suppose we would like to know the value \(O^*[A',5]\) we would load \(O^*[A',3] = 1\) and check \(B[4]\) (which is \('C'\)) and \(B[5]\) (which is \('A'\)) for an occurrence of \('A'\). If there is an occurrence, we increase. In this case we end up with 2; the correct value for \(O^*[A',5]\).

There is also a method to keep the size of the \(S\) array small. It is stored in intervals of 32. With a method presented in \([32]\) it is possible to obtain the original value of \(S\) from the compressed version, \(S^*\). A full discussion of this method is outside the scope of this work, because this method is not used in the ‘aln’ step of BWA. Assuming the values of \(S\) are stored in 32-bits format, this gives a size of about 380 MiB for the human genome.

To summarize, the size of the index of a full human genome amounts to less than 1.6 GiB, as seen in Table 3.2.

Table 3.2: Index size of a full human genome

<table>
<thead>
<tr>
<th>Component</th>
<th>(B)</th>
<th>(O^*)</th>
<th>(S^*)</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size (MiB)</td>
<td>750</td>
<td>380</td>
<td>380</td>
<td>1510</td>
</tr>
</tbody>
</table>

Further on in this chapter, it will be explained that the actual index for BWA is twice as large as is shown in Table 3.2, but still well within 4 GiB.

Generating the index of the full human genome takes a considerable amount of time, mostly due to the generation of the BWT string. This involves sorting extremely large suffixes of the reference. On a modern PC platform, this can take many hours. However, the index only has to be generated whenever the reference changes. Researchers mostly use the aforementioned GRCh38 or HG19 or some other reference, but rarely change it.
The previous major version, GRCh37, was released four years before GRCh38. Therefore, this work will not continue to focus on the performance of the step where BWA generates the index.

3.3 Burrows-Wheeler Aligner

The FM-index that was discussed in the previous section allows for a backwards search in the suffix array according to Algorithm 1. However, the algorithm only allows for exact matching, while the reference genome $G$ and the reads may slightly differ at certain positions, such as is explained in Section 2.2.1 and 2.2.2. It is therefore needed that a short-read alignment algorithm supports inexact matching as well.

3.3.1 Inexact matching

Consider again the example where the reference genome $G = \text{"GATGACCA"}$. Suppose we want to align a short-read $R = \text{"GAA"}$. Assume that we allow the algorithm to find hits with at most one mismatch. The goal of the inexact matching algorithm should now be to find two hits, aligning "GAA" with both "GAT" at position 0 and with "GAC" at position 3. These are the only two hits containing one mismatch.

Suppose the base-pairs of the reads are parsed one by one in the backward-search manner of Algorithm 1, but now we allow one mismatch to occur in the whole read, with respect to the reference.

Now, as long as this mismatch did not occur yet, we should continue the search not only for the interval that corresponds with the current base-pair, but also for any base-pair for which a valid interval exists in the suffix array.

This means, while searching, as long as:

- a mismatch is still allowed
- the interval for each base-pair type is a valid interval in the suffix array (i.e. $k \leq l$)

we should continue to search within all the resulting intervals. This means that the search space is not of linear order, but increases exponentially as long as mismatches are still allowed.

Before explaining how BWA manages to decrease the search space, again, the example for $R = \text{"GAA"}$ is given in Table 3.3. Because an algorithm will be given later, the leftmost column shows the position of the read that is processed. The symbol $z$ denotes the number of differences that are still allowed. Wherever a cell is green, there is a match for that interval and base-pair. When a cell is orange, there is a mismatch, but we can still continue since $z \geq 0$. When a cell is red, either $z < 0$ or the interval is invalid (the substring does not exist in the reference).

In the table, in the first step where the number of differences is still $z = 1$, instead of looking at one interval for the corresponding base-pair 'A', the intervals for all base-pair types are considered. Thus, for four intervals it is checked whether they are valid. In the
Table 3.3: An example of the intervals to search when using inexact matching.

<table>
<thead>
<tr>
<th>Pos.</th>
<th>Intervals to check</th>
</tr>
</thead>
<tbody>
<tr>
<td>Init</td>
<td>$k = 0$, $l =</td>
</tr>
<tr>
<td>2</td>
<td>$k = 1$, $l = 3$, $z = 1$, $\sigma = 'A'$</td>
</tr>
<tr>
<td></td>
<td>$k = 4$, $l = 5$, $z = 0$, $\sigma = 'C'$</td>
</tr>
<tr>
<td></td>
<td>$k = 6$, $l = 7$, $z = 0$, $\sigma = 'G'$</td>
</tr>
<tr>
<td></td>
<td>$k = 8$, $l = 8$, $z = 0$, $\sigma = 'T'$</td>
</tr>
</tbody>
</table>

case of the example, all intervals are valid. Not coincidentally these are the four parts of the suffix array corresponding to each character in the alphabet $\Sigma$.

Continuing with each of those intervals, we can see that the first interval of the row for position 1 where $k = 2$ and $l = 1$ gives us an invalid interval, since the suffix "$AA..."" does not exist in the reference.

However, if we follow the path where in position 2, $k = 4$ and $l = 5$, then for the next position, a match is found for interval $k = 2$ and $l = 2$, since the suffix "$AC..."" does exist. However, the number of differences allowed is $z = 0$ after the first step, since a mismatch was allowed there. This means no more mismatched are allowed thereafter, and therefore at the following step, for position 0, when the previous interval had a mismatch, only trying the interval for $\sigma = 'A'$ is allowed.

Eventually two intervals are found, \{6,6\} and \{7,7\}. The corresponding search paths in the suffix tree representation are shown in Figure 3.7. Similar to the table, green arrows denote a match, orange a mismatch that is still allowed and red an invalid interval. A green node depicts a hit for the whole read.

It is important to notice that, in the worst case, the search space can grow exponentially with respect to the number of base-pairs that are processed. More formally, not taking insertions and deletions into account, allowing $z \rightarrow |R|$ and assuming all partial intervals are valid, the search space is of $O(|\Sigma|^{|R|})$. Luckily, for real short-reads $z$ can stay small. This mostly depends on the accuracy of the sequencing technology used. By default BWA allows $z = 5$ for 93 base-pair reads and $z = 4$ for 64 bp reads, and so on. Thus, the search space is bounded by not only $\Sigma$ and $|R|$ but also by $z$ such that the upper bound on the search space is more similar to $O(|\Sigma|^z|R|)$.

How BWA partially reduces the search space furthermore, and the algorithm that BWA utilizes will be given in the next section.
3.3.2 Limiting the search space

Because the search space may grow exponentially, processing one read may take a considerable amount of time, especially when the number of allowed mismatches is high. One of the main contributions of the original BWA implementation [4] is a clever method to reduce the search space.

BWA defines the array $D$, which at some point $i$ in a read $R$, gives $D[i]$; a lower bound on the number of differences in the reference $G$. Suppose that we are performing inexact matching. At some point $i$ in the read $R$, we arrive at a suffix array interval, and three differences are still allowed, i.e. $z = 3$. When the lower bound on the number of differences at this point is, for example $D[i] = 1$, we know that we can continue with this interval. To match the rest of the read we need to allow at least one difference, while we can still allow three. However, consider the case where these values are interchanged. Suppose we arrive at some interval with $z = 1$ and $D[i] = 3$. There is no need to continue searching in this interval, since there is no way to arrive at the end of the read without allowing at least three mismatches. Even though there might be valid intervals, we do not have to continue searching them, because $D$ guarantees us that we will not be able to find a hit that matches within our maximum number of differences. It is therefore pointless to continue the search for this interval.

To give a good estimation of $D$, the original work first shows an inefficient method of $O(|R|^2)$ time-complexity. This method starts by finding sequential sub-strings of the read, that exactly match the reference. For example, consider the read $R =$ "ATGTCC" in the reference $G =$ "GATGACCAS". We may first search for the substring "A" and
find that there is a match in the reference. Then, trying "AT", and "ATG" until we find that "ATGT" is not in the reference. We increase $D$ at this position to show that when backward searching, allowing one difference here is required to finish the search. Then we restart the search for substring "T" and continue to find that "TC" is not in the reference again, increasing $D$ at this position. Finally, "C" and "CC" will be found to be in the reference, and $D$ is not increased any further. Thus, in this example it is determined that $D = \{0, 0, 0, 1, 2, 2\}$.

Since the time-complexity of finding an exact match with the FM-index was shown to be $O(|R|)$ but in theory $R$ contains a worst-case of $|R|$ sub-strings, the worst-case time-complexity of using the regular FM-index is $O(|R|^2)$.

BWA shows a more time-efficient method of $O(|R|)$ to determine $D$. When a forward search would be allowed, whenever attempting to exact match, if an invalid interval is discovered, $D$ is increased and continue with the largest suffix array interval again, i.e. $\{0, |G|\}$, effectively restarting the exact matching. There are two methods to allow for a forward search. Either the read can be reversed, and a backwards search can be performed to mimic a forward search, or the reference can be reversed and a normal backward search can be performed. The latter seems expensive, especially for the size of the index. However, reversing the reference has several benefits.

When the DNA molecule strands separate in order to form a new DNA molecule, the new strand is synthesized starting from the 5'-end to the 3'-end. However, since the two separated strands are anti-parallel to each other, once synthesis begins, it can either continue in the direction of the so called forward strand or the reverse strand. Thus, a read may be taken from the forward or the reverse strand.

When searching for a read without having a reverse reference, the search has to be performed four times; one time to determine the $D$-value, one time to determine if it matches the forward reference, one time to determine the $D$ value for the reverse complement and one time to determine if the complemented read matches the reverse reference.

Initially BWA included the reverse reference while searching, so the complemented reads still had to be searched. This was later improved to make use of the so called FMD-index [33]. The FMD-index creates an FM-index not only of the reference, and then separately the reverse reference, but of the concatenation of the reference and the reverse complement of the reference.

Before indexing, the example reference $G =$" GATGACCA$\$" would become $G =$" GATGACCATGCTACATC$\$". As discussed in Chapter 2, 'T' is the complement of 'A' and 'G' is the complement of 'C'. When using $S$ to retrieve the absolute position in the reference from a suffix array interval, one may simply decide whether the location is in the original or the reverse complemented reference by looking if the position is over half the total size or not.

By using the FMD-index, BWA gains two advantages. First, when determining the size of $D$, one can simply complement the read and by backward-searching the reverse complemented reference, a forward-search is mimicked. Secondly, when aligning a read, both strands are searched simultaneously.
Thus, by including the reverse complement of the reference genome in the index, only two searches are needed. The algorithm taken from [4] to determine $D$ is as seen in Algorithm 2. In this representation it is assumed $O$ is built from the FMD-index and not the reverse index proposed in the original paper.

**Algorithm 2** BWA’s algorithm to calculate a lower bound on the number of differences, taken from [4]

```
function CalculateD(R, C, O)
    k ← 0
    l ← |G| − 1
    i ← 0
    b ← 0
    for i ≤ 0 do
        σ ← R[i]
        k ← C[σ] + O[σ, k − 1] + 1
        l ← C[σ] + O[σ, l]
        i ← i + 1
        if k > l then
            k ← 0
            l ← |G|
            b ← b + 1
        end if
        D[i] ← b
    end for
    return D
end function
```

### 3.3.3 The inexact matching algorithm of BWA

Combining the description of inexact matching with the FM-index shown in Section 3.3.1 and the array with the lower bounds on the number of differences $D$ from Section 3.3.2, the main alignment algorithm that BWA conceptually utilizes is seen in Algorithm 3.

From Algorithm 3 in general the working of BWA is as follows.

- Line 2: The bound array $D$ is calculated.
- Line 3: The recursion is started on the largest interval of the suffix array.

Then inside the recursion:

- Line 7: We check if enough errors are still allowable to find a hit by looking at the $D$ array and the number of difference still allowed.
- Line 10: When the last character of the read has been processed, i.e. $i = −1$, the current interval is a hit.
Algorithm 3 BWA’s inexact matching algorithm, taken from [4]

1: function InexactMatch($R, C, O$)  
2:    CalculateD($R, C, O$)  
3:    return InexRecur($R, C, O, |R| - 1, z, 0, |G| - 1$)  
4: end function

5: function InexRecur($R, C, O, i, z, k, l$)  
6:    if $z < D[i]$ then  
7:        return $\emptyset$  
8:    end if  
9:    if $i < 0$ then  
10:       return $\{k, l\}$  
11:    end if  
12:    $I \leftarrow \emptyset$  
13:    $I \leftarrow I \cup$ InexRecur($R, C, O, i - 1, z - 1, k, l$) ▶ Insertions  
14:    for all $\sigma \in \Sigma$ do  
15:        $k \leftarrow C[\sigma] + O[\sigma, k - 1] + 1$  
16:        $l \leftarrow C[\sigma] + O[\sigma, l]$  
17:        if $k \leq l$ then  
18:            $I \leftarrow I \cup$ InexRecur($R, C, O, i, z - 1, k, l$) ▶ Deletions  
19:                if $\sigma = R[i]$ then  
20:                    $I \leftarrow I \cup$ InexRecur($R, C, O, i - 1, z, k, l$) ▶ Matches  
21:                        else  
22:                            $I \leftarrow I \cup$ InexRecur($R, C, O, i - 1, z - 1, k, l$) ▶ Mismatches  
23:                    end if  
24:                end if  
25:        end if  
26:    end for  
27:    return $I$  
28: end function

• Line 14: If we allow insertions, then at any point of the read we should skip the current base-pair while staying in the same interval, and decrease the number of differences allowed $z$.

• Line 15: Then, for every base-pair type the following should be done:

• Line 16-18: Check if the interval is valid for this basepair type.

• Line 19: If we allow deletions from the reference, go to the new interval but keep processing the same base-pair (hence $i$ is not subtracted).

• Line 20-23: Check if we have a match or a mismatch, and decrease $z$ only when a mismatch occurred.

Even though Algorithm 3 is a conceptual representation of the functionality, the real BWA implementation somewhat differs, including but not limited to the following points:
• BWA does not only use $z$ but scores insertions, deletions and mismatches independently and with different values.

• BWA implements a software stack instead of using a recursion, so lines 14, 19, 21 and 23 actually push entries to a stack. The stack is a breadth-first-search stack, which is implemented as an array of pointers to sub-stacks of which each sub-stack holds all entries with the same score. The entries with the lowest scores are processed first, so that the algorithm very quickly converges on the best hit.

• In the real implementation a read is first loaded in reverse, then \texttt{CalculateD} does a backward-search on the original (not complemented) read (beginning with the last character of the original read) to determine the bounds. Then, \texttt{InexactMatch} and \texttt{InexRecur} effectively reverse and complement the read and perform a new backward search.

• For each type of mismatch, insertion and deletion, BWA allows for different limits, i.e. we can independently allow a maximum number of mismatches, insertions and deletions.

• BWA has an option to allow a variable amount of mismatches in a variable amount of first few base-pairs. This part of the read is called the \textit{seed}. The original work reports that it has a close to negligible effect on the accuracy of the algorithm.

• A multi-threaded implementation is available that uses the \texttt{pthreads} framework to create multiple threads.

• BWA’s source code contains almost no comments, but after analysis shows to contain many pieces of hand optimized code involving specific parts of the algorithm.

3.4 Performance analysis

To identify the run-time of different functions of the program, the program is profiled using \texttt{gprof}. The profiling results discussed here are generated by running the software on the platform which was chosen for implementation. This platform is the Convey HC-2 hybrid-core computing platform. It contains two Intel Xeon E5-2643 chips with 4 cores per chip running at 3.30GHz and has 128GB DDR3 SDRAM as physical memory. More information about the additional capabilities of this platform will follow in the next chapter.

3.4.1 BWA function profile

Two measurements have been performed. One measurement on an index of merely human chromosome 22 with synthetic reads generated using the program \textit{wgsim} [34] are seen in Figure 3.8. The synthetic reads are used because real short-reads are of very specific lengths and do not allow for a evenly spaced set of data points. Short-read batches of 100,000 reads were used.
3.4. PERFORMANCE ANALYSIS

The second measurement is done on real data on the full human genome. For this measurement, subsets of two real datasets were used; the same one as the original work with reads of 51 base-pairs long, and one set with reads of 100 base-pairs long. This is seen in Figure 3.9.

In both cases, BWA was run with default settings.

For clarity, the exact function names are not used in this overview. Also, several functions that have different hand-optimized implementations to used in very specific cases, but that exhibit the same functionality, are aggregated. For example, BWA has six functions to obtain values in the occurrence array. A function called \texttt{bwt\_2occ4} is used when eight occurrence array values at two positions for each of the four base-pair types are needed where they might reside in the same interval, and \texttt{bwt\_2occ} is used when only the value for one base-pair type is needed from two positions.

From the profile data several things are noticeable. First of all, for the synthetic benchmarks, when the read size grows, more time is relatively spent in obtaining occurrence array values. This is due to the search space becoming larger, because for larger reads more mismatches are allowed by default. On average, obtaining the values from the memory takes about 45% of the time.
Secondly, BWA still performs exact matching when the maximum number of differences still allowed is zero. Thus, it can be seen that for very small reads, where only one difference is allowed, BWA continues searching with *exact matching* a lot. When the read size grows, exact matching will only be used in the very last parts of the search. On average, exact matching takes about 25% of the time.

Third, when more differences are allowed, more time is relatively spent in *inexact matching*.

Finally, it can be seen that BWA spends a relatively small amount of time in the core loop, in loading reads from the disk, in calculating the D array (which is a sort of exact matching) and any other functions.

### 3.4.2 BWA total run-time

From the profile shown in the previous section it does not become clear how what type of input contributes to the *total* run-time of the program. Even though some points may be trivial, it is important to mention them. The most important input and settings that contribute to the total run-time are summed up:

- **Size of the index.**
  - If the index is larger, containing more combinations, then it is more likely that at some point in the backwards search, a partial hit is found, thus increasing the search space. For example, a real set of 250,000 reads aligned to the full human genome took 75 seconds, while the same set of reads took only 22 seconds to align to chromosome 22 only.

- **Size of the read.**
  - If the read is *very* small, then it is more likely that the number of alignments is larger, since the read has a higher chance to occur in several places and the value of D at the start of the search is not very high. This increases the run-time. However, weighing more heavily, since the read is shorter, less base-pairs have to be processed. Inversely, if the read is very large, then it is more likely that the number of alignments is smaller and the value of D may be higher at the start. However, since the read is larger, more base-pairs have to be processed, which generally outweighs the relatively smaller search space.

- **Number of differences allowed**
  - If more differences are allowed, the search space is in the worst-case increased exponentially as explained in Section 3.3.2.

- **Read quality**
  - If the read is of a low quality, while searching, it will sooner decrease the number of mismatches allowed. Thus, a batch of low-quality reads will be aligned faster (if alignments can be found at all). This motivates using the seeding strategy, to speed up the searching for high-quality reads.
3.4. PERFORMANCE ANALYSIS

3.4.3 Memory usage

Because BWA is a memory intensive application, the memory access pattern was analyzed as well. It was already shown that the memory access pattern is extremely random [26], such that BWA does not benefit much from a cache. Even though the work shows a method to achieve a higher hit-rate in the cache, the speed-up is small (1.2X).

One thing that was furthermore analyzed in this work was the amount of loads of the same occurrence value. By running a benchmark on merely chromosome 22 as index, it was measured that the top $2^{14}$ most accessed occurrence array values, which summed up to in total $5.98 \cdot 10^8$ accesses, only constitutes to just over 1% of the total accesses made, which were $5.56 \cdot 10^{10}$ accesses in total.

When a new short-read is being mapped, the first four intervals are always the same, because the first character $\sigma \in \Sigma$ is easy to find. However, as more characters are processed by the backward search, the intervals start to become relatively more unique, and not often the same interval will be accessed. This is visualized in Figure 3.10 where the top 80 occurrence array accesses are shown.

From the figure, it can be seen that the first four occurrence values are frequently loaded, and then the next 16 occurrence values are frequently loaded, then the next 64 occurrence values, and so forth. However, even though they are relatively accessed often, the total amount of locations accessed is so large, i.e. the graph should continue over the x-axis for so long, that accelerating the algorithm by storing the most frequently accessed occurrence values (or even whole intervals with corresponding BWT string) is not likely to benefit the run-time of the algorithm.

Apart from the memory access pattern, the size of the stack was analyzed as well. Over several runs it was measured that the stack, especially due to the Breadth-First-
Search approach of BWA, can grow up to 60 MiB in size, only counting the data in the stack, and not counting the data-structure to order the BFS stack.

3.4.4 Discussion

By looking at both the algorithms used in BWA and at the profiling data, several things are important to observe before moving on to a hardware implementation.

As most alignment tools, BWA is a memory intensive application. When looking at the profile, this quickly becomes obvious. Almost half the run-time is spent on obtaining occurrence array values.

Also, the exact matching (and consequently the inexact matching) algorithm exhibits practically no parallelism. The algorithm does not use any complex arithmetic. The only calculation that is done is where a new suffix array interval is determined. Consider again the step in the algorithms where \( k \leftarrow C[\sigma] + O[\sigma, k-1] + 1 \) and \( l \leftarrow C[\sigma] + O[\sigma, l] \). The new value for \( k \) and \( l \) depend on the old values for \( k \) and \( l \). More specifically, the addresses that must be loaded for the occurrence array value \( O \), are dependent on the previous iteration of the algorithm. Thus, using the backwards search of the FM-index, it is impossible to pre-fetch data or to make intermediate calculations in advance.

However, the inexact matching function in software contains 47 sequentially written branches inside one iteration of which the outcome depends on the partial hits but also on program parameters. A combinatorial hardware implementation of these branches may provide benefits over using a sequential circuit.

More importantly, because a typical data-set contains millions of short-reads, and each short-read can be aligned independent of the other, a lot of parallelism is exhibited at the input level. Therefore it is to be expected that a design where as many inexact matching units as memory bandwidth allows are implemented in parallel, a high speed-up may be gained.
FPGA design and implementation

In the previous chapter, several components of the BWA _aln_ were analyzed. This chapter will focus on discussing how the different components are mapped to hardware, starting at the top-level of the design in Section 4.1. In Section 4.3, one specific unit to obtain the occurrence values is presented. The unit that performs inexact matching is shown in Section 4.5. Finally, any specifics about the design with respect to the Convey computer implementation are presented in Section 4.6.

### 4.1 Architecture

Before proposing a hardware implementation for BWA, several requirements, goals and limitations are imposed on the design:

- The design should report similar results as possible to the software implementation of BWA.
  - The design should at least report the best hit, which should be the same as BWA’s best hit.

- The initial goal is to optimize for throughput; as much reads should be processed as possible within a certain amount of time. Area is important if it influences the maximum throughput. Latency and power are not of issue at the moment.

- The design should have the same command-line interface and options of BWA, so that for DNA researchers, the usage is unchanged.

It is important to discuss some limitations with respect to any hardware implementation. The components that mainly contribute to the run-time of the "aln" algorithm of BWA were analyzed in the previous chapter:

- Exact matching
- Inexact matching
- Obtaining the occurrence array value

The major drawback with respect to a possible parallelized hardware implementation is that the iterations of exact- and inexact matching depend on previous iterations. The FMD-index backwards search algorithm and the conceptual suffix tree do not allow for intermediate calculations to be done in advance. It is impossible to pre-calculate the branch of the tree, or the interval of the suffix array, in which a search will end up, in advance of doing the search itself.
The inexact matching function of the BWA software implementation contains 47 (mostly independent) branches, which can likely be optimized by using combinatorial circuits for the many different decisions, feeding the selection lines of multiplexers that have the different outcomes as inputs.

One additional consideration to make for implementing the inexact matching function is the stack. Preventing the stack to be placed in the main memory would advantageous, since many loads and stores can be prevented. Lacking a cache, typically RAM blocks that are available in many FPGA’s would offer a good solution to hold the stack. Although the priority Breadth-First-Search stack can grow up to 60 MiB, which is too large for most block RAM in FPGA’s, a Depth-First-Search may also be implemented to achieve the same results. The drawback is that the algorithm does not converge to the best hit as fast.

Exact matching will most likely be hard to optimize, since this is a very simple function, very similar in implementation to the description of Algorithm 1.

The software implementation of obtaining the occurrence array value can be somewhat optimized in hardware, since it involves counting a number of base-pairs and adding it to a reduced occurrence array value. Since the base-pairs are stored in 2-bit per base-pair format, the software implements a hand optimized but undocumented method of converting the BWT string of interest to count the occurrences in the interval of interest. This method requires many bit-wise operations and some loops with a small number of iterations. In hardware, it is possible to do this fully combinatorial and with a limited number of gate levels.

While calculating the $D$ array (with the lower bound on the number of differences at some position in the read) does not constitute to a lot of run-time, it is more efficient to implement this in hardware as well and store the result in block RAM. In this way, the values of $D$ do not have to constantly travel through a memory interface while inexact matching. Also, the $\text{CALCULATE}D$ algorithm makes use of occurrence array values, so that the unit that is used in inexact matching may be reused.

A major advantage (with respect to a hardware implementation) of BWA is that, at the input level, there is a high level of parallelism. Each short-read could be processed independent of the other short-read. This will allow multiple cores to run in parallel. To save area, it will also allow for a pipeline to process multiple short-reads in parallel through different stages.

Since each iteration of a stack-based implementation of Algorithm 3 generates one or more entries, but (except for insertions) each new entry advances the search to the next character in the read, if there is a theoretically unlimited number of cores that can simultaneously access memory to processing such a new entry, then the time-complexity is only bounded by $|R|$. In fact, assume $|R|$ is constant over all short-reads in a batch and insertions are not allowed. On an unbounded PRAM model, the time-complexity of the algorithm would be merely $O(|R|)$. However, since the search space can grow exponentially and generally memory can only process accesses sequentially, the parallelism is harder to exploit at this level.
Therefore, the proposed design will focus on having several cores processing reads independently. Furthermore, a real implementation would have both of the following (trivial) restrictions:

- There can be only a limited number of cores.
- There is a limited memory bandwidth.

As long as the number of cores can grow such that the memory bandwidth is fully utilized, then the design will be optimized for the given platform. However, when the memory bandwidth is more than sufficient, but the number of cores is the limit, then also the memory latency will be of importance to the total run-time.

### 4.1.1 Implementation platform

The platform on which the design is implemented is a Convey Computer Hybrid-Core 2 (HC-2). This platform contains two Intel Xeon E5-2643 chips that can serve as the host processor, that has 128 GiB DDR3 SDRAM available. In addition, the platform offers four Xilinx Virtex-6 LX760 FPGA’s that have uniform access to a separate memory of up to 64 GiB. In the case of full utilization of the memory interfaces, it can have a maximum throughput of 80 GiB/s [35]. To give a more generic overview of the design, the Convey HC-2 specifics of the design will be discussed later, in Section 4.6. First this work will focus on explaining the design in a more conceptual manner, and only mention HC-2 specifics if this had an important impact on the design.

### 4.1.2 Top level architecture

Taking the previous points into consideration, at the top-level, the architecture seen in Figure 4.1 is proposed.

Since BWA does not spend a lot of the total run-time on pre- and post-processing reads, this work does not invest time into rewriting or remapping these functions. BWA is open-source, so most of the pre- and post-processing functions are reused. This includes loading the reads in batches, converting them to two-bit per base-pair format, and some other trivial tasks. Thus, a **host processor** for which the BWA software can be compiled is used for these tasks. Also **disk access** will be dealt with by this **host processor**. This approach saves a significant amount of development time over creating some hardware-only design. Furthermore, the design is fully integrated into the original BWA source, so that all the command-line parameters stay the same, as was required.

The **aln core** in Figure 4.1 (the name is after the command-line argument for the algorithm this work is attempting to accelerate) will implement all functions that constitute mostly to the run-time: exact matching, inexact matching and obtaining the occurrence array values.

The functionality of this level of the design is now as follows. The **host processor** will load and pre-process the short-reads from memory. The FMD-index and all short-reads ready to be processed by the cores will also be copied to the **core memory**
(available to the ALN CORES). Then, the parameters and the base address of the index are passed to the ALN CORES and they are started independently.

The reason that each core must be controlled independently is that the run-time of processing the short-read on each core depends on the data, which makes it unknown at design-time. Thus, the cores start and finish asynchronously.

After an ALN CORE is finished processing the short-read, it will pass the suffix array interval to which the read is aligned back to the HOST PROCESSOR, so that it can be written to disk. As long as the batch of short-reads is not empty, the host processor will load the new short-read into the core memory and start the first core that is idle.

Because the source code of BWA contains many manual low-level optimizations, macro’s and other exotic C constructs (especially for sequential processing on the x86 platform) and implements a BFS method for which the stack does not fit in some block-RAM of an FPGA, the code is unsuitable for a C-to-HDL synthesis tool. Such a tool will not detect the high-level meaning of several hand optimized functions and will not recognize that the algorithm can also be implemented in a DFS manner, so that the stack will fit in some block-RAM. Furthermore, because BWA is internally undocumented, it is expected that before fully understanding the code and rewriting it such that it becomes suitable for a C-to-HDL tool, even more time will be spent than manually implementing the HDL design. The ALN CORE is therefore manually implemented in VHDL.

### 4.2 ALN Core architecture

The ALN CORE consists of several components, shown in Figure 4.2.

This core contains some registers to hold parameters and base addresses that the
4.2. ALN CORE ARCHITECTURE

core must use during alignment. When the command to start is passed via a control register, the core units are reset and started.

The **short-read loader** obtains the short-read from memory and holds it locally so that whenever a certain base-pair at some position is requested, it is almost immediately available without having to wait for memory.

The short-read is then processed first by the **Calculate D unit**. This unit performs the function of Algorithm 2. From the algorithm it can be seen that for each iteration some occurrence array value at \( k \) and \( l \) for a specific base-pair type are needed. The **Occurrence Obtain Unit** that is attached to the **Calculate D unit** obtains these two values in an efficient manner, which is discussed in section 4.3.

When the bounds in \( D \) are known, the read and \( D \) are accessed by the **Inexact Matcher**, which starts a Depth-First-Search according to Algorithm 3. In each iteration of this algorithm, eight occurrence array values are required; for each base-pair type at position \( k \) and \( l \). The **Occurrence Obtain Unit** that is attached to the **Inexact Matcher** is used for this. Partial hits and score-keeping information are stored in the **Stack**. The **Inexact Matcher** may push up to nine new entries to the stack each round; a match, three mismatches, an insertion and four types of deletions. The **Inexact Matcher** is discussed in Section 4.5.
Figure 4.3: Occurrence Obtain Unit

When the Inexact Matcher cannot find any more items on the stack, the result is passed back to the registers and a status register is updated, indicating that the core is done processing all entries of the stack.

4.3 Occurrence Obtain Unit

Even though there are two variants of the Occurrence Obtain Unit, only the one that is connected to the Inexact Matcher is discussed here, since the one that is connected to the Calculate D unit is a simplified version of this.

The function of the unit is to obtain eight values given two positions \( k, l \), all occurrence array values of each base-pair type at position \( k \) and \( l \). As described in Section 3.2.3, not the whole occurrence array is stored, but only intervals of it. More specifically, the interval for BWA is 128 base-pairs. In this work, the data-structure is called the reduced occurrence array or \( O^* \). Since the occurrence array counts the number of occurrences in the BWT string \( B \) up to some position, to get an occurrence array value, the following must be done, given some position \( k \):

- Calculate in which interval \( k \) resides
- Load the four corresponding \( O^* \) values.
- Load the BWT string in that interval.
- Count the occurrences of each character up to \( k \) in the interval.
- Add this to \( O^* \).

In this way \( O(\sigma, k), \forall \sigma \in \Sigma \) will be known.
The design of the **Occurrence Obtain Unit** is shown in Figure 4.3. A controller receives the request with \( k \) and \( l \) from the Inexact Matcher. Then, it will calculate the interval and load the \( O^* \) values in the registers shown in red. Then, the BWT strings are loaded.

A small example will be given to illustrate how the **Base-pair counter** is controlled. Suppose some value of \( k \) is given, such that in the occurrence array interval of interest, the base-pair up to which the occurrences have to be counted is the 75th.

In the FMD-index of BWA, the BWT strings in the intervals are stored as 32-bit words, that contain 16 base-pairs in 2-bit per base-pair format. Thus, there are eight 32-bit words in an interval, but the implemented design has 64-bit word memory accesses. Therefore, two 32-bit words are loaded simultaneously and are reordered so that they are properly aligned to the rest of the circuit, and 32 base-pairs are obtained in one load. Effectively, in this design, one interval contains four 64-bit words that contain the BWT string.

In the case of the example, to accumulate the proper value of \( O(\sigma, k), \forall \sigma \in \Sigma \), the 64-bit word that contains the last base-pair is the third word. Therefore, for the first two words, the position of interest that controls up to which base-pair the total count is added, is 31; the position of the last base-pair in the word. This is because all base-pairs in the word contribute to the total count.

However, when arriving at the last word, in the example: word 2, the position of interest is now 10, the 11th base-pair in the word, since \( 75 - 32 - 32 = 11 \). The **Base-pair counter** should stop counting there, so that the occurrence value is correct. Thus, this unit should be able to count up to any base-pair position in a word. The example is furthermore shown in Table 4.1.

### Table 4.1: Example of how the position of interest is controlled for the occurrence obtain unit when the 75th base-pair in the interval is required.

<table>
<thead>
<tr>
<th>Word</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base-pairs</td>
<td>0-31</td>
<td>32-63</td>
<td>64-95</td>
<td>96-127</td>
</tr>
<tr>
<td>Position of interest</td>
<td>31</td>
<td>31</td>
<td>10</td>
<td>-</td>
</tr>
</tbody>
</table>

One optimization that also exist in the software implementation is that, very often, the interval in which \( k \) and \( l \) reside are the same. The registers for \( k \) and the Base-pair counter for \( k \) are therefore duplicated for \( l \) is well. Whenever they are in the same interval, the controller loads the \( O^* \) values in both registers, and loads the highest BWT string word required (which is always determined by \( l \), otherwise the SA interval we’re searching in would be invalid). If \( k \) and \( l \) are not in the same interval, the intervals are loaded in sequence.

The design for the **Base-pair counter** is shown in Figure 4.4.

In this design, when a BWT string arrives, it is first filtered for each specific base-pair type (in the figure, only 'A' is shown). The filter simply converts the 64-bit BWT string word to a 32-bit representation, where whenever at some position the base-pair of
interest occurs, the bit is set to 1, otherwise to 0. This single-bit representation is fed into an AND circuit, and AND-ed with a run of ones up to the position of interest, effectively masking off the base-pairs that should not be counted. From this point onward, the problem is reduced to a population count of the resulting bits.

Because the FPGA’s that are used for implementation are Virtex-6 FPGA’s, and make optimal use of a Configurable Logic Block when the number of inputs to a function is eight bits [36], the design therefore implements 8:4 compressors (note that to count the number of ones in an eight-bit string, the result could range from zero to eight itself, therefore requiring four bits at the output) and a few adders to arrive at the final result which is represented by six bits.

4.4 Calculate D Unit

The Calculate D unit, as seen in Figure 4.5, is a relatively simple unit that performs the algorithm seen in Algorithm 2. It works as follows.

When the unit is started, it iterates over a read according to the algorithm. In each iteration it obtains a base-pair from the Short-Read Loader. In combination with the current suffix-array interval $k, l$ the two occurrence values for that base-pair type are
obtained. The resulting interval is checked for validity, and if it is valid, the registers that hold the \(k, l\) interval are updated, otherwise they are reset. In the meanwhile, the value \(b\) is updated and written to a True Dual-Port RAM (TDPR) block in the FPGA. For the current design to support read lengths up to 128 base-pairs, the TPDR requires 1 RAMB36E1 and 1 RAMB18E1 resource in the Virtex-6 LX760.

This TPDR is required because due to the fact that the Inexact Matcher not only requires one value of \(D\), but also the previous value in the array. Thus, to save cycles when \(D\) is accessed by the Inexact Matcher, a TPDR is used. There are never write conflicts as the controller will only write to the TPDR on one port.

When the Calculate D unit is done generating the D array, the Inexact Matcher can request values from the D-array. These values will be obtained by the controller from the TPDR and passed on to the Inexact Matcher.

### 4.5 Inexact Matcher

The design of the Inexact Matcher is shown in Figure 4.6. To keep the discussion of this unit to a reasonable length, note that this figure is an extremely simplified version of the actual implementation, showing the most important data-paths only.

The Inexact Matcher functions according to Algorithm 3 and works as follows. The stack is initialized to contain an entry with the largest interval of the suffix array, i.e. \(k = 0, l = |G| - 1\). Also the current base-pair to process is set to the last base-pair, i.e. \(i = |R| - 1\), and \(z\) is set to whatever amount of differences the user wants to allow. Note that just like the BWA software, the hardware implementation keeps track of mismatches and insertions and deletions separately to determine the score of an entry.
Passing the interval $k, l$ to the **Occurrence Obtain Unit**, when an answer is received, in the **Interval Generation** unit, all the possible intervals for each base-pair type are calculated according to

$$k \leftarrow C[\sigma] + O[\sigma, k - 1] + 1$$

and

$$l \leftarrow C[\sigma] + O[\sigma, l]$$

resulting in eight new intervals. These intervals are then passed to the **Interval Validation** unit that determines whether they are valid intervals to push to the stack, also taking into account the search parameters, the value of $D$ at the current and previous position, and the allowed number of differences $z$.

The result is a vector where each potentially new type of entry has its own bit; one for a match, three for mismatches, one for an insertion and four for all types of deletions. This vector is fed to a controller that sequentially processes only the set bits of the vector and steers the **Entry Generator** to generate the designated stack entry to push onto the stack.

Next to also controlling all handshaking to all other units on the higher level of the design, the controller also keeps track of many other things, such as whether a hit is found. A hit is found whenever $i$ reaches zero. The result is stored in a high-score register only when the score of the hit is better than the hit that is already stored. This is a drawback of the proposed design; a best hit will always be found (note that multiple...
4.5. INEXACT MATCHER

hits may have the same score), but all other hits are discarded. A bit of accuracy is
lost, but a lot of speed is gained, because unlike the BWA software, any partial hits
that already have a higher (worse) score than the high-score, can never become the new
high-score, thus they are immediately discarded. The software implementation of BWA
somewhat applies a similar strategy, but still allows suboptimal partial hits to continue
as long as the score stays within a certain threshold. Furthermore, when new entries are
generated, the entries with the best score are pushed last, such that they are popped
first again, somewhat converging on the best hit faster.

To support very large genomes, the values for \( k \) and \( l \) in the software implementation
are stored as 64-bit integers. However, only 33-bit integers are required to store the
largest suffix array position. This saves memory in the stack and area for the many
multiplexers in the design, and timing issues are less likely to occur.

Seeding is supported, but the current version does not support mismatches in the
seed itself, where the software implementation can be parametrized to allow a small
number of mismatches in the seed.

4.5.1 Stack

The stack is implemented in the block RAM of the Virtex-6 FPGA. For a Depth-First-
Search, a depth of 256 entries was chosen. Note that when the user set \( z \) very high,
this might not be sufficient when the reads are mapped to a large index, since many
intervals will be valid and many branches of the search will be entered because a high
number of difference is allowed. However, with default settings for \( z \), staying typically
below 5, it is more than sufficient. Suppose all intervals are valid while searching (which
is highly unlikely) when all nine possible types of entries are pushed and one popped,
effectively growing the stack by 8 per position, for a read length of 32 base-pairs the
stack will just be large enough. Still, the stack has a specific signal to indicate that it is
full. If this happens the search will terminate and the read will not be aligned.

Each stack entry is 120-bits wide, containing data similar to the software imple-
m entation. It requiring less bits (the software uses 256 bits), since in hardware it is easy
to optimize data sizes. The data an entry contains is:

- The position in the read \( i \).
- The number of mismatches.
- The number of insertions.
- The number of extensions of an insertion.
- The number of deletions.
- The number of extensions of a deletion.
- Whether the entry is currently in an insertion or deletion or normal alignment with
  matches and mismatches.
• The current suffix array interval, $k, l$

The number of block RAM resources of the Virtex-6 that are used by this stack design is 2 (out of 720 total available).

4.6 Convey HC-2 implementation

The previously described design was implemented on the Convey HC-2 platform. Each of the four FPGA’s on the Convey machine can implement a so called Application Engine (AE). Inside the AE, one or multiple units such as the ALN Core may be placed. The core can use the infrastructure of Convey’s development environment, so that a developer can quickly map his/her application onto the system. This way, the developer does not have to bother with the memory infrastructure or the connection to the host processor.

An AE can communicate with the host processor in two ways. This typically depends on how much data must be transferred. For instructions and small control- and status-like signals the so called Dispatch Interface may be used. Specific AE instructions can be created, or AE register moves can be made from/to the host processor. For large sets of data, memory copy functions are available to copy from the host memory to the memory available to the AE, called the coprocessor memory. An interface called the Memory Controller (MC) Interface is available to an AE so that it can access the coprocessor memory. Each AE can have access to two Memory Controllers.

To copy the data (parameters, short-reads and FMD-index) to the Application Engine, the design presented in this work takes the following approach:

• The FMD-index is copied to the coprocessor memory.

• A batch of short-reads is loaded in the host processor memory.

• The ALN COREs are reset and initialized with the alignment parameters.

Then, as long as reads are still unaligned:

• Memory is allocated in the coprocessor memory for a new short-read.

• The short read is copied to the coprocessor memory.

• The base-address of the short-read is passed to the first idle core through an AE register.

• A bit corresponding to the idle core is set in an AE control register which will start alignment on the core.

In the current state of the design, each ALN Core uses one Memory Controller port to connect it to the coprocessor memory, so that it may perform memory loads from the FMD-index. Each Memory Controller contains two ports, which are effectively just two interfaces to the custom logic in the AE core that are later multiplexed onto one interface to the rest of the memory architecture.

The AE interface and the memory controllers themselves run at 300 MHz, but the designs at the AE side of the two ports, typically run at 150 MHz. The implemented design also runs at this clock speed.
4.6. CONVEY HC-2 IMPLEMENTATION

4.6.1 Memory bandwidth and area usage

An iteration during inexact matching of the ALN CORE takes a worst case of 29 cycles plus memory latency, divided over the tasks seen in Table 4.2.

Table 4.2: Cycle times of a typical aln core iteration.

<table>
<thead>
<tr>
<th>Cycles</th>
<th>Task</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Popping an entry from the stack.</td>
</tr>
<tr>
<td>3</td>
<td>Determine what to do with the entry.</td>
</tr>
<tr>
<td>4</td>
<td>Requesting a new occurrence value and calculating addresses.</td>
</tr>
<tr>
<td>5-8 or 10-16</td>
<td>Load requests to the memory controller (depends on the data).</td>
</tr>
<tr>
<td>35-500</td>
<td>Memory latency (typically 100 cycles).</td>
</tr>
<tr>
<td>4</td>
<td>Calculating the real occurrence value and passing it back.</td>
</tr>
<tr>
<td>1-9</td>
<td>Pushing new entries to the stack (depends on the data).</td>
</tr>
</tbody>
</table>

In the worst-case, the amount of data loaded are eight 64-bit words. If we take a memory latency of typically 100 cycles, then the memory interface processes 512 bits per 137 cycles. At a clock speed of 150 MHz, this means that almost 0.92 microseconds are spent on one iteration processing 512 bits, and that the bandwidth that is used on one port is almost 67 MiB/s. However, the peak memory bandwidth of one port is 1.25 GiB/s.

This already indicates that a more optimal design would implement either a pipelined version of the design (which will be shown in the next section), and/or more cores to achieve a higher bandwidth, aligning more reads in parallel. However, the device area usage must be considered first.

The design was mapped to the Virtex-6 LX760 with one, two, four and eight ALN CORES. The area usage can be seen in Table 4.3. The relative area usage compared to the available resources of the LX760 is seen in Figure 4.7.

Table 4.3: Area statistics for the design (grey is extrapolated data)

<table>
<thead>
<tr>
<th>Resource type</th>
<th>LX670</th>
<th>1 core</th>
<th>2 cores</th>
<th>4 cores</th>
<th>8 cores</th>
</tr>
</thead>
<tbody>
<tr>
<td>Slices</td>
<td>118560</td>
<td>21845</td>
<td>26492</td>
<td>30102</td>
<td>40125</td>
</tr>
<tr>
<td>LUT Flip Flop pairs</td>
<td>67648</td>
<td>81854</td>
<td>94787</td>
<td>124004</td>
<td></td>
</tr>
<tr>
<td>Fully used LUT-FF pairs</td>
<td>32729</td>
<td>38983</td>
<td>44203</td>
<td>55796</td>
<td></td>
</tr>
<tr>
<td>Slice register</td>
<td>948480</td>
<td>51777</td>
<td>63024</td>
<td>69250</td>
<td>84700</td>
</tr>
<tr>
<td>Slice LUTs</td>
<td>474240</td>
<td>53112</td>
<td>64069</td>
<td>76322</td>
<td>103176</td>
</tr>
<tr>
<td>Logic</td>
<td>46240</td>
<td>54856</td>
<td>65454</td>
<td>91039</td>
<td></td>
</tr>
<tr>
<td>Memory</td>
<td>132480</td>
<td>4798</td>
<td>6866</td>
<td>7594</td>
<td>10026</td>
</tr>
<tr>
<td>Routing</td>
<td>2074</td>
<td>2357</td>
<td>2183</td>
<td>2111</td>
<td></td>
</tr>
<tr>
<td>RAMB36E1</td>
<td>720</td>
<td>69</td>
<td>90</td>
<td>100</td>
<td>120</td>
</tr>
<tr>
<td>RAMB18E1</td>
<td>1440</td>
<td>17</td>
<td>18</td>
<td>20</td>
<td>24</td>
</tr>
</tbody>
</table>

From these figures, it can be seen that the area footprint of the design is relatively small. The Stack and the Calculate D unit take up 2 RAMB36E1’s and
1 RAMB36E1 + 1 RAMB18E1 respectively. However, as more cores are attached to Convey Memory Controller ports, the Convey memory and dispatch infrastructure introduce most of the area overhead. The FIFO’s in the Memory Controllers actually take up more block RAM space than the design itself. The 8 core implementation uses half the memory infrastructure available to one Application Engine (4 Memory Controllers), thus the overhead is expected to grow even more when implementing more cores or when using more of the memory controllers.

When using 8 cores and consequently 8 memory ports on 4 memory controllers, the memory bandwidth per AE is $8 \times 67 = 536$ MiB/s. The maximum memory bandwidth of the Convey HC-2 can be 80 GiB/s, but this is only achieved in the specific case where all four FPGA’s contain an AE, and where each AE uses eight memory controllers.

From the area statistics it is safe to estimate that 16 cores will fit as well, so that all memory ports are used, achieving double the throughput. The throughput for one AE then becomes 1072 MiB/s. However, the peak bandwidth for one AE is 20 GiB. This means that the memory bandwidth usage is merely 5.2% of the available bandwidth. The current design, with respect to utilizing memory bandwidth, is therefore far from optimal.

It is not expected that much more than 16 cores may fit in one FPGA. Thus, adding more cores that share a memory controller port would not be a feasible solution. A better solution would be to introduce pipelining to the design. Since only about a third of the FPGA area is used for the design where eight cores are implemented, it is expected that a pipelined design with eight cores is more feasible. Such a design would be able to achieve a higher memory throughput by processing multiple reads in parallel in a pipeline. The goal of the design would be to maximize memory throughput by loading a data-word on every cycle of every memory port for every Application Engine. In the next section, a pipelined design and estimation for its performance will be presented.
4.7 Pipelined design

One major improvement over the current design would be to apply pipelining to the inexact matching unit, such that the design is able to hide the memory latency and make constant use of all functional units.

While inexact matching, during one iteration, multiple entries may be pushed onto the stack. These entries may be processed independently. One approach with respect to a pipelined approach will therefore be that the pipeline could process multiple stack entries for the same read at a time.

However, this approach has several drawbacks. The original design pushes the best scoring entry last. In this way, the implementation converges upon the best hit quicker than when less optimal entries are pushed last (and therefore popped first). If a hit is found quicker, any lower scoring entries are not processed anymore.

Consider the case where a pipelined implementation would pop entries from the stack that are less likely to become best hits (while the best entry so far is further on in the pipeline already). When these less optimal entries are processed, it is likely that multiple even less optimal entries are pushed back again to the stack, as a result from processing the suboptimal entry. Thus, the stack will grow extremely fast with most likely suboptimal entries. Therefore, a larger stack is needed as well. Furthermore, the stack may be empty, resulting in an empty pipeline, especially at the beginning and the end of the search.

A better approach would be to have multiple stacks, where each stack holds entries for a different short-read. This way, in a round-robin fashion, entries are popped from the stack and fed into the pipeline. The drawbacks, concerning a rapidly growing stack discussed previously, are not present. However, it would still require more block RAM to hold all the stacks.

While this design is not implemented, for further work on this subject, a pipelined design is proposed in Figure 4.8.

The pipelined design would have the following stages:

- **EF: Entry Fetch**: Load an item from the stack, and load the $D$ value and the base-pair from the short-read.
- **AG: Address Generation**: Calculate the occurrence interval and generate the addresses to load the data from.
- **OLI: Occurrence Load Interval $k$ & $l$**: Pass load requests to the memory controller for the reduced occurrence array interval for the current $k$ and $l$.
- **OLB: Occurrence Load BWT $k$ & $l$**: Pass load requests to the memory controller for the BWT string in the occurrence array interval for the current $k$ and $l$.
- **OWx: Occurrence Wait stage $x$**: Pipeline stages to hide the memory latency.
46

CHAPTER 4. FPGA DESIGN AND IMPLEMENTATION

Figure 4.8: Design for a pipelined Inexact Matcher.

- **OC: Occurrence Calculate**: Calculate the occurrence values from the reduced occurrence value and the BWT string.
- **PD: Push Deletions**: Push four possible deletions.
- **PIMM: Push Insertion & Mismatches**: Push a possible insertion and three possible mismatches.
- **PM: Push Match**: Push a possible match, or a possible mismatch that was not pushed in the previous stage.

Each stage will take 4 cycles, which balances the pipeline according to the amount of cycles different actions need, as seen in Table 4.2. Most of the functional units of the previous design can be quickly reused to implement the stages of a pipelined design.

### 4.7.1 Performance estimation

The performance of the pipelined design will depend heavily on the optimal number of memory latency hiding stages, which is subject for further investigation. If the average memory latency is taken, which is 100 cycles on the 150 MHz clock, then there should be 25 memory hiding stages of 4 cycles each. Since there are ten other stages, there will be 35 stages in total. The total latency will be 140 cycles, instead of 137 cycles for the non-pipelined design. The stages are therefore well balanced, only introducing a minimal overhead of 3 cycles.

Using the average memory latency, the best-case performance of the single-core implementation is that one inexact matching iteration is performed every 118 cycles in the best case, and 137 cycles in the worst case. Thus, the number of iterations per cycle is between 0.0073 and 0.0085. The pipelined design is estimated to use 140 cycles per
iteration, but performs 35 iterations simultaneously. The number of iterations per cycle is 0.25 for the pipelined design (since there are 4 cycles per stage). By pipelining the design, the performance is estimated to be increased by between 29.5 in the worst-case, and 34.25 times in the most optimal case. In reality, it will most likely be slightly lower due to pipeline stalls occurring when the memory latency is sometimes exceeding the typical value. Thus, a rough estimation of the speed-up of the pipelined design over the non-pipelined design is 30X.

One drawback to the design with respect to using the optimal memory bandwidth, is that, if a load takes longer than 100 cycles, the pipeline will have to stall to allow the load to complete before the corresponding Occurrence Calculate stage can begin. This may be solved by not using the average latency cycles to determine the number of Occurrence Wait stages, but the average plus one standard deviation, for example. This will require more stages, thus more stacks and more area. Effectively, the speed-up will be increased even more, since more reads are processed in the pipeline in parallel, but is limited by the available bandwidth and will require deeper FIFOs.

Also, if a load takes shorter than the number of latency cycles, it is required to buffer the load until the Occurrence Calculate stage begins. This will require more registers or block RAM to be used for a FIFO. The amount of data that needs to be stored depends on the minimum memory latency and the number of stages.

To achieve the maximum bandwidth on the Convey HC-2, it is required that all four Application Engines read from all eight memory controllers on every cycle on each of the two ports [35]. Therefore, one AE will have to make sixteen loads each 150MHz cycle. In both the Occurrence Load Interval stage, there will always be four loads over four cycles. In the Occurrence Load BWT stage in the best case (with respect to memory bandwidth) also four loads are performed. Thus, one core will load eight words over four cycles or two words per single cycle. This means that to achieve the maximum memory bandwidth, one AE should implement eight pipelined cores, such that one AE loads 16 words per cycle. In conclusion, eight cores per AE/FPGA is an upper bound to the number of cores required to achieve maximum memory bandwidth. Note that this is independent of the pipeline depth.

The figures of 30X speedup and eight cores per AE fit well with calculations on the available memory bandwidth. Suppose 32 cores are implemented. The current core uses 69 MiB/s. 32 non-pipelined cores times 69 MiB/s use 2.16 GiB/s, which is 37X less memory bandwidth than available.

### 4.7.2 Area estimation

As discussed in Section 4.5.1, one stack requires 2 RAMB36E1 units in the Virtex-6 LX760. Therefore having 35 stacks, so that in the typical memory latency case, the pipeline is completely filled, would require 70 of these units. Furthermore, each short-read would need the D array to be available as well, taking another 35 RAMB36E1’s and 35 RAMB16E1’s. It is therefore estimated that there is more than enough block RAM to implement eight cores.

It is much harder to estimate the area usage of the pipelined design, since the additional control logic and the pipeline registers are not present in the current design.
Currently, for eight cores only about a third of the resources are used. Although speculative, it is feasible that the pipelined design will be able to fit eight times as well in one FPGA.
In this chapter, several measurements of correctness and performance will be presented. First, the correctness of the design is evaluated in Section 5.1. Secondly, the performance measurements are presented in Section 5.2. In Section 5.3, real data benchmarks are shown. To conclude, the scalability is discussed, and an estimation of the speed-up for a pipelined design is given in Section 5.4.

In all measurements of this chapter, all times are reported as wall clock times for the alignment step, which is the step that was mapped to hardware. Allocating and moving the index from the host processor to the coprocessor memory, which is performed only at the start of the program, is not included in these measurements (this takes about 2 seconds for the full human genome). Allocating and moving short-reads while reads are dispatched to the coprocessor are included in these measurements. All software measurements are made by running BWA on the host processor only (which is a Intel Xeon E5-2643 with 128 GiB DDR3 SDRAM). All hardware measurements were done with the host processor only dispatching reads and collecting results.

5.1 Functional verification

To validate the correctness of the alignments of the hardware implementation, the suffix array intervals of hits are compared to those of the software implementation. From Section 4.5, it is important to repeat that the Inexact Matching Unit differs from the original BWA implementation in such a way that it will never continue with sub-optimal entries at all, i.e. stack entries that already have a lower score than a found hit are immediately discarded.

Thus, to evaluate correctness, first of all, the same amount of reads should be aligned as in the software implementation. Secondly, the best hit found by the hardware implementation should have the same score as the best hits found by the software.

For example, in Listing 5.1 and 5.2, consider the (for this purpose) customized output to show the suffix-array intervals of the alignments of both the software and hardware of a certain data-set. The listing shows a short-read number in the batch and to which suffix array intervals it is aligned. If there is any alignment, the hardware will always find all the best alignments, but the current design only reports the first one found.

In the case of the example, for read 17, the software finds 19 alignments, where two alignments have the best score (which is 3). The breadth-first-search method of the original work finds a different alignment first. The second alignment it finds, is the same as the alignment found in the depth-first-search method of the hardware first, and has the same score. It is yet to be considered how reporting only one best alignment affects the downstream tools, but the design can easily be adjusted to report the alignments in
a similar way to BWA as well.

Listing 5.1: Software alignments

\[
\begin{align*}
[15] \text{aligned to 1 SA itv:} & \{83340839,83340839\}[3] \\
[16] \text{aligned to 1 SA itv:} & \{79102512,79102512\}[3] \\
[17] \text{aligned to 19 SA itv:} & \{64966822,64966825\}[3] \\
& \{64966766,64966766\}[3] \\
& \{64967050,64967050\}[6] \\
& \{64966826,64966826\}[6] \\
& \ldots
\end{align*}
\]

Listing 5.2: Hardware alignments

\[
\begin{align*}
[15] \text{aligned to 1 SA itv:} & \{83340839,83340839\}[3] \\
[16] \text{aligned to 1 SA itv:} & \{79102512,79102512\}[3] \\
[17] \text{aligned to 1 SA itv:} & \{64966676,64966666\}[3] \\
[18] \text{aligned to 1 SA itv:} & \{40033953,40033955\}[0] \\
[19] \text{aligned to 1 SA itv:} & \{3061453,3061453\}[6]
\end{align*}
\]

In Table 5.1, several results are shown. The table can be read as follows: \textit{s32x100k} means that 100,000 simulated short-reads of 32 base-pairs long were used. Batches prefixed with ‘r’ are real reads. The options shown are BWA specific, where -n means total number of differences allowed (including gaps), -o means gap opens allowed (indels) and -e means gap extensions allowed. BH stands for Best Hit, and shows whether the hardware best-hit was also found by the software.

In the table, the type of short-read batch, the reference used, and the options are shown. Furthermore, for each combination, it was measured whether the same amount of reads are aligned, and the total number of alignment is measured. Because of the behavior described previously, the hardware implementation reports only one hit per read. However, the software also reports suboptimal alignments within a specific score range of the best hit. Also, note that when zero differences are allowed, inexact matching is essentially the same as exact matching, and there can be only one suffix array interval with a hit.

<table>
<thead>
<tr>
<th>Batch</th>
<th>Reference</th>
<th>Options</th>
<th>Reads aligned</th>
<th>Total alignments</th>
<th>BH</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>SW</td>
<td>HW</td>
<td></td>
</tr>
<tr>
<td>s32x100k</td>
<td>CHR22</td>
<td>-n 0</td>
<td>51836</td>
<td>51836</td>
<td>Yes</td>
</tr>
<tr>
<td>s32x100k</td>
<td>CHR22</td>
<td>-n 2</td>
<td>97104</td>
<td>97104</td>
<td>191904</td>
</tr>
<tr>
<td>s32x1k</td>
<td>GRCh37</td>
<td>-e 2 -o 1 -n 4</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
</tr>
<tr>
<td>s56x100k</td>
<td>GRCh37</td>
<td>-n 3</td>
<td>96832</td>
<td>96832</td>
<td>188798</td>
</tr>
<tr>
<td>s64x100k</td>
<td>GRCh37</td>
<td>-n 0</td>
<td>26613</td>
<td>26613</td>
<td>26613</td>
</tr>
<tr>
<td>s72x5k</td>
<td>GRCh37</td>
<td>-e 1 -o 1 -n 3</td>
<td>4695</td>
<td>4695</td>
<td>6589</td>
</tr>
<tr>
<td>r51x1k</td>
<td>GRCh37</td>
<td>-n 4</td>
<td>995</td>
<td>995</td>
<td>2134</td>
</tr>
<tr>
<td>r51x255k</td>
<td>GRCh37</td>
<td>-n 3</td>
<td>240800</td>
<td>240800</td>
<td>456034</td>
</tr>
</tbody>
</table>
5.2 Performance

The current implementation of the design contains eight non-pipelined cores. Thus, this chapter will show the performance results for single-, double-, quad- and octa-core performance. Several types of data-sets are used. To get a fine range of read lengths, the program wgsim was used (with default settings) to generate simulated reads of the full human genome. The source of the reference is human genome version GRCh37.

Also, real data-sets of short-reads are used to evaluate the performance. One data-set is the same as in the original work, read batch ERR000589, which contains reads of size 51. To keep the run-times to a reasonable length, a subset of the data-set is used; about 255k short-reads. The other data-set is, read batch SRR062634, consisting of reads of length 100, of which about 200k are used. Both read batches are available from the online databases of the 1000 Genomes Project

The following measurements are performed:

- Multiple size simulated short-reads on a full human genome index.
- Multiple size real short-reads on a full human genome.

The following setups are used:

- Two-core performance versus two-thread performance.
- Four-core performance versus four-thread performance.
- Eight-core performance versus eight-thread performance.

5.2.1 Simulated short-reads

To obtain short-reads of several lengths, the program wgsim was used to generate reads in a range from 8 to 64 base-pairs with steps of 8.

5.2.1.1 Ungapped alignment

In Figure 5.2 to 5.4, different sizes of simulated reads were aligned to the human reference genome GRCh37. No gaps were allowed, and seeding is disabled. For these figures, the legend of Figure 5.1 is used. SWx stands for software implementation, x threads. HWx stands for hardware implementation, x cores. SUx stands for speedup of x hardware cores over x software threads. The x-axis of all figures shows the read lengths used.

The speed-up is slightly higher than for short-reads aligned to a small index, most likely due to the much larger reference genome causing more cache misses for the software implementation on the host processor. The index for chromosome 22 is about 50 MiB large, and the cache of the host-processor is 25 MiB, thus a large part of the reference will be available in the cache for many loads.

1http://www.1000genomes.org/
CHAPTER 5. RESULTS

Figure 5.1: Graph legend

Figure 5.2: Ungapped alignment, zero differences allowed.

Figure 5.3: Ungapped alignment, two differences allowed.

Figure 5.4: Ungapped alignment, four differences allowed.
In Figure 5.2, it can be seen that the speed-up is slightly below 0.09 in the best cases, but drops to just below 0.04 when more cores are used. The reason for this is the overhead of the dispatch interface. When zero differences allowed, the search space is very small, so not a lot of time is spent aligning the short-reads. Thus, the overhead of the dispatch interface becomes more dominant in the total run-time.

In Figure 5.3 and 5.4, the Dispatch Interface overhead becomes relatively less dominant, because the search space increases as the number of differences allowed is greater. Per unit of time, less reads are dispatched, so the relative overhead becomes smaller.

Especially in Figure 5.4, it can be seen that, as stated previously, multiple cores in parallel exhibit great scalability, since every read may be processed in parallel. The run-times of the one-core hardware implementation are almost exactly double that of the two-core implementation, and the two core-implementation run-time is almost exactly double of the four-core implementation, and so forth.

In this measurement, the speed-up of the eight core implementation slightly overtakes the speed-up of the one-, two- and four-core implementation, which shows that the scalability is slightly better for the hardware implementation, as long as the search space is large enough. Four differences is a realistic number of differences for reads of length 60 and beyond.

For the measurements in this section, the average speed-up was 0.082, making the current design about 12 times slower than the software implementation when performing ungapped alignment.

5.2.1.2 Seeding

In Figure 5.5 to 5.7, different sizes of simulated reads were aligned to the human reference genome GRCh37. Here, seeding was enabled at different lengths. The legend of Figure 5.1 is used again.

In these figures, it can be seen that both the hardware and software implementation benefit from seeding in specific cases. For example, in Figure 5.6, the average speed-up for only seeding the first 10 base-pairs is increased to nearly 0.10, compared to not seeding at all in Figure 5.4, where the speed-up always stayed below 0.05.

However, when the seed gets longer and less suffix-array intervals are likely to be found when continuing with inexact matching after the seed, the search space becomes very small again. Thus, for larger seeds, such as in Figure 5.7, the overhead of using the Dispatch Interface becomes more evident again.

For the measurements in this section, the average speed-up was 0.072, 0.085 and 0.080 for a seed length of 10, 20 and 30, respectively. This makes the hardware implementation about 13 times slower than the software implementation when performing ungapped alignment with seeding.
Chapter 5. Results

Figure 5.5: Seed length of 10, two differences allowed.

Figure 5.6: Seed length of 10, four differences allowed.

Figure 5.7: Seed length of 30, four differences allowed.
5.2. PERFORMANCE

5.2.1.3 Gapped alignment

In Figure 5.8 to 5.10, different sizes of simulated reads were aligned to the human reference genome GRCh37, where one gap was allowed. This means that for each iteration, not only matches or mismatches are pushed to the stack, but also insertions and deletions, doubling the amount of stack entries an iteration can generate.

The drawback of using Depth-First-Search is even more prominent from these measurements. Since the search space grows twice as fast when allowing insertions and deletions, the hardware implementation performs very poorly, again due to the Depth-First-Search not converging on the best hit as quickly as the software implementation.

For the measurements in this section, the average speed-up was 0.045. This means that when performing gapped alignment, the hardware implementation is about 22 times slower than the software implementation.
CHAPTER 5. RESULTS

Figure 5.8: One gap open allowed, one difference allowed.

Figure 5.9: One gap open allowed, two differences allowed.

Figure 5.10: One gap open allowed, three differences allowed.
5.3 Realistic reads and parameters

While the previous section has shown results of simulated reads, this section will evaluate performance for real short-reads. Furthermore, realistic parameters are used as well.

![Figure 5.11](image1.png) Performance of aligning a subset of ERR000589 (51 bp reads), seed length 20, 3 mismatches, 1 open and 6 extensions allowed.

![Figure 5.12](image2.png) Performance of aligning a subset of SRR062634 (100 bp reads), seed length 30, 5 mismatches, 1 open and 6 extensions allowed.

In Figure 5.11, a batch of reads of length 51 is aligned with close to default settings of BWA (the hardware does not yet allow mismatches in the seed). The speed-up obtained is somewhat similar to that which is seen in the simulated data-sets. Because there is a smaller amount of measurements, the X-axis will now show the number of threads or cores used to align the read batch to the reference.

In Figure 5.12, a batch of reads of length 100 is aligned, also using the default settings of BWA. The average speed-up over these measurements is 0.96, making the hardware implementation almost 11 times slower when aligning with realistic data-sets and settings.
5.4 Estimations for a pipelined design

In this section, the potential speed-up of a pipelined design with full resource utilization of the platform is considered.

First the scalability of the design should be taken into account. For many of the parameter settings used in the previously shown measurements, the parameters are not very realistic. This causes the overhead of the Dispatch Interface to seem of great influence. However, for data-sets with more realistic settings, where the search space for each read is significant, the overhead becomes relatively small. Furthermore, it is possible to implement the design in such a way that the cores load unprocessed reads themselves, effectively making the Dispatch Interface obsolete, canceling its overhead.

Table 5.2 shows different data-sets with realistic settings, the number of cores or threads used, the run-time of software and hardware (in the columns SW (s) and HW (s)), the speed-up relative to the previous row for each implementation (in the columns denoted by SU prev.), the speed-up of the software implementation over the hardware implementation (in SW/HW). Furthermore, because in Section 4.7 it was estimated that a pipelined implementation could be around 30 times faster, the run-time was divided by that number (and shown in the column Pipel. HW (s)). In the last column, the pipelined speed-up is shown. All cells that are grey contain estimated values.

Table 5.2: Estimations for a pipelined implementation.

<table>
<thead>
<tr>
<th>Cores / threads</th>
<th>SW (s)</th>
<th>SU prev.</th>
<th>HW (s)</th>
<th>SU prev.</th>
<th>SW/HW</th>
<th>Pipel. HW (s)</th>
<th>Pipel. SU</th>
</tr>
</thead>
<tbody>
<tr>
<td>ERR005589 3 mismatches, 1 gap, 6 extensions, length 20 seed</td>
<td>1 10.45</td>
<td>115.20</td>
<td>0.09</td>
<td>3.84</td>
<td>2.72</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 6.46</td>
<td>57.97</td>
<td>1.99</td>
<td>0.11</td>
<td>1.93</td>
<td>3.34</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4 3.28</td>
<td>30.11</td>
<td>1.92</td>
<td>0.11</td>
<td>1.00</td>
<td>3.27</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8 1.71</td>
<td>19.90</td>
<td>1.51</td>
<td>0.09</td>
<td>0.66</td>
<td>2.57</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16 2.29</td>
<td>11.00</td>
<td>1.81</td>
<td>0.21</td>
<td>0.37</td>
<td>6.24</td>
<td></td>
<td></td>
</tr>
<tr>
<td>32 2.17</td>
<td>6.08</td>
<td>1.81</td>
<td>0.36</td>
<td>0.20</td>
<td>8.42</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SRR062634 5 mismatches, 1 gap, 6 extensions, length 30 seed</td>
<td>1 28.29</td>
<td>355.75</td>
<td>0.08</td>
<td>11.86</td>
<td>2.39</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 15.58</td>
<td>175.42</td>
<td>2.03</td>
<td>0.09</td>
<td>5.85</td>
<td>2.66</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4 8.31</td>
<td>87.93</td>
<td>2.00</td>
<td>0.09</td>
<td>2.93</td>
<td>2.83</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8 4.95</td>
<td>44.72</td>
<td>1.97</td>
<td>0.11</td>
<td>1.49</td>
<td>3.32</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16 5.80</td>
<td>22.40</td>
<td>2.00</td>
<td>0.26</td>
<td>0.75</td>
<td>7.77</td>
<td></td>
<td></td>
</tr>
<tr>
<td>32 5.92</td>
<td>11.22</td>
<td>2.00</td>
<td>0.53</td>
<td>0.37</td>
<td>13.22</td>
<td></td>
<td></td>
</tr>
<tr>
<td>64 bp simulated, 4 mismatches, ungapped, length 10 seed</td>
<td>1 6.43</td>
<td>66.75</td>
<td>0.10</td>
<td>2.23</td>
<td>2.89</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 2.89</td>
<td>33.12</td>
<td>2.02</td>
<td>0.09</td>
<td>1.10</td>
<td>2.62</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4 1.40</td>
<td>16.68</td>
<td>1.99</td>
<td>0.08</td>
<td>0.56</td>
<td>2.52</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8 0.79</td>
<td>8.37</td>
<td>1.99</td>
<td>0.09</td>
<td>0.28</td>
<td>2.83</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16 0.78</td>
<td>4.19</td>
<td>2.00</td>
<td>0.19</td>
<td>0.14</td>
<td>5.56</td>
<td></td>
<td></td>
</tr>
<tr>
<td>32 0.78</td>
<td>2.09</td>
<td>2.00</td>
<td>0.37</td>
<td>0.07</td>
<td>11.13</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Because there are no measurements for 16 or 32 cores, the average speed-ups of 2 over 1 core, 4 over 2 cores, etc., has been used to estimate the run-time. While it seems rather optimistic, there is no reason to assume that the hardware scales as bad as the software multi-threaded implementation, which is limited by the number of cores of the host. The Intel Xeon processor of the host contains four cores that support 2 threads. The optimal number of threads for BWA on the host processor is eight (as seen in the measurements). Adding more threads does generally not increase performance. To keep the estimation as fair as possible, the lowest run-time was selected whenever adding more threads did not cause a speedup. When replacing the Dispatch Interface to start with a new alignment per core, and letting the cores load a new short-read to align themselves, it is highly likely that the scaling of the hardware will stay near linear.

From the table, the prognosis is made that if the following assumptions hold:

- there are enough resources to fit 32 cores
- the scaling stays near linear also for 16 and 32 cores
- pipelining will cause a speed-up of about 30X over the non-pipelined implementation

... the hardware implementation will indeed be able to achieve a speed-up of (for these data-sets) between 2X and 13X
In this report, the problem of the alignment of millions of short DNA reads to a reference genome was discussed. To accommodate personal genomics in the future, it is desirable that aligning short-reads is done as quickly as possible, so that clinical applications become faster and more effective.

It was shown that many software tools exist for short-read alignment, but that there are only a few attempts at mapping short-read alignment algorithms to hardware. Therefore it was decided to investigate the potential speed-up of a hardware implementation of a component of the well known tool called the Burrows-Wheeler Aligner. This work solely focused on the `aln` algorithm.

The Burrows-Wheeler Aligner is a versatile tool that makes extensive use of the FMD-index, which allows for a backwards search in a reference genome and its reverse complement at the same time. The method uses a conceptual suffix array, where the search converges on a specific interval of the array. The original work on the Burrows-Wheeler Aligner showed that, by using the relatively small FMD-index and several methods to reduce the search space, the tool becomes very efficient to run on normal desktop computers.

In this report it was determined that the parallelism of the algorithm itself is very low, but since determining the suffix-array intervals of a short-read may be done independently of the other short-read, the parallelism at the input is very high; i.e. many short-read aligners may operate in parallel. Also some of the software functions may be accelerated by simply creating combinatorial logic for the many independent branches that are implemented in the original code.

The work continued to focus on the implementation of a single short-read alignment core on the Convey Computer Hybrid-Core 2 platform, containing a high-end memory infrastructure and four Virtex-6 LX760 FPGA’s, that allow for a high data throughput. The core is functionally similar to the Burrows-Wheeler Aligner, but implements a Depth-First-Search in block RAM instead of a Breadth-First-Search, which makes the algorithm converge on the best hit less efficiently. This is done to save area such that as many cores may be mapped to the platform as possible and to prevent many memory transfers. Furthermore, units to obtain occurrence values are implemented in such a way that they are optimized for the Virtex-6.

Several data-sets with simulated reads from the human genome were aligned using both the hardware and software aligner. The hardware aligner reports the same best hits as the software aligner, thus the core is verified to be functionally correct. However, different performance measurements have shown that the hardware implementation currently does not provide a speed-up over using the host-processor of the Convey HC-2 only. For ungapped alignment without seeding, the design is still about 12 times slower. With seeding the design is about 13 times slower. For gapped alignment, where the
search space is even larger, the design performs about 22 times slower.

There are two main reasons the design is not achieving a speed-up. Most importantly, the algorithm is using Depth-First-Search instead of Breadth-First-Search, which causes the best hit to be found much slower. This often causes many sub-optimal paths of the search tree to be traversed, that would otherwise be discarded. Secondly, the Convey HC-2 platform has a high memory latency, which causes the current implementation to use only a fraction of the available memory bandwidth. This latency may be hidden by implementing the proposed pipelined design, such that on every clock cycle a load request is sent to all the memory controllers of the platform.

The pipelined design is estimated to achieve a 30X speed-up over the current hardware implementation. Also, the hardware implementation scales practically linearly for realistic alignment settings, while the software implementation does so only up to eight threads. It is estimated that a pipelined hardware core will fit 32 times on the whole platform. The prognosis for the speed-up of a 32 pipelined core design running on the coprocessors over 8 threads running on the host processor of the Convey HC-2 is between 2X and 13X.

6.1 Recommendations for future work

There are several recommendations for future work that will be discussed in this section.

For the design presented in this work, the following improvements can be made:

- **Allow sub-optimal hits.** The design can easily be adjusted so that it continues and reports sub-optimal hits within a certain threshold, like the software implementation.

- **Fill up the platform’s resources.** For now only one Application Engine is implemented, which contains eight ALN CORES. The total platform can have four AE’s with 32 cores.

- **Implement the pipelined design.** Because only a fraction of the available bandwidth of the system is used, and because there are still enough resources to extend the design, it would be worthwhile to implement the pipelined version of the design (which was presented in Section 4.7. In this way, by processing more short-reads in parallel and loading on every memory controller port on every clock cycle, the maximum memory bandwidth of the Convey HC-2 is utilized.

- **Load short-reads and write back the results directly to memory, independent of the dispatch interface.** In the current state of the design, every time a short-read is aligned, the base address of the next short-read has to be passed through the Dispatch Interface which is stalled when there are memory loads. This is very ineffective because the Dispatch Interface stalls when memory loads are being performed. It would be better to pass a base address of a large short-read batch to each core and let the core calculate the location of the next read itself, and write back the result as well through the memory interface.
6.1. RECOMMENDATIONS FOR FUTURE WORK

- **Implement Breadth-First-Search.** Initially it was estimated that a high amount of overhead was required to place all stack entries in memory. However, when allowing gapped alignment it was measured that the Depth-First-Search method become very ineffective. It may be possible to implement the a BFS in such a way that very good scores will still be kept in the block RAM of the FPGA, and that only bad scoring entries will spill to the main memory. During inexact matching, the core does not write anything at all, so there may be no overhead while spilling these entries to the main memory. Depending on the size of the local stack, there is also a probability high enough probability that the best hit will be found in the local stack. This will cause many of the entries spilled to main memory to be discarded because there may be many that do not fall within the best-hit threshold anymore.

More general recommendations for future research into accelerated versions of short-read alignment are:

- **Use seed-and-extend algorithms.** The algorithm for searching the FMD-index is very effective for exact matching, but becomes less effective for inexact matching, since the search space may grow exponentially, as was shown before. The BWA software already offers a seed-and-extend algorithm which is called `bwa mem`. Although the algorithm is more dynamic in nature than `bwa aln`, after seeding, it applies Smith-Waterman. For this algorithm, a lot of data is loaded at once, in contrast to `bwa aln` loading very small chunks of data in each iteration. It requires more computational complexity, making it more suitable for acceleration in FPGA. Implementations of Smith-Waterman on the Convey platform are already available [37].

- **Investigate to different types of indexes.** The FMD-index of BWA was especially designed to allow it to fit in regular desktop computers. However, a machine such as the Convey HC-2 has a lot more resources available. It is worthwhile to investigate tools using indexes that are larger, but allow for less time to be spent in finding alignments. In this way, time is traded for memory footprint size. One good candidate would be the recent `segemehl` short-read alignment tool.
Bibliography


BIBLIOGRAPHY


