

Using an MPSoC to implement DNA sequence alignment

Implementation and acceleration of the Smith-Waterman
algorithm

Robin NOLLET

Promotor: Prof. dr. ir. Davy Pissoot
Co-promotor: Ing. Václav Šimek
Begeleider: Ing. Jonas Lannoo

Masterproef ingediend tot het behalen van
de graad van Master of Science in de
Industriële wetenschappen: Elektronica-ICT
afstudeerrichting Elektronica

Academiejaar 2019 - 2020

©Copyright KU Leuven

Without written permission of the supervisor(s) and the author(s) it is forbidden to reproduce or adapt in any form or by any means any part of this publication. Requests for obtaining the right to reproduce or utilise parts of this publication should be addressed to KU Leuven Campus Brugge, Spoorwegstraat 12, B-8000 Brugge, +32 50 66 48 00 or via e-mail iiw.brugge@kuleuven.be.

Zonder voorafgaande schriftelijke toestemming van zowel de promotor(en) als de auteur(s) is overnemen, kopiëren, gebruiken of realiseren van deze uitgave of gedeelten ervan verboden. Voor aanvragen i.v.m. het overnemen en/of gebruik en/of realisatie van gedeelten uit deze publicatie, kan u zich richten tot KU Leuven Campus Brugge, Spoorwegstraat 12, B-8000 Brugge, +32 50 66 48 00 of via e-mail iiw.brugge@kuleuven.be.

Acknowledgements

From the Brno University of Technology (BUT) I would like to thank the following people:

- **Ing. Václav Šimek** as my supervisor for providing feedback on the process of executing this thesis, as well as helping with the administrative side of the Erasmus exchange and recommending courses at BUT.
- **Ing. Tomáš Martínek, Ph.D.** for providing the learning materials towards HLS and to point out the idea behind the BFAST algorithm.
- **Ing. Vojtěch Mrázek, Ph.D.** for providing the ZCU 104 evaluation kit hardware.

From KU Leuven I would like to acknowledge the following people:

- **Ing. Jonas Lannoo, Ph.D. Student** as a supervisor for his support during my Erasmus towards the thesis.
- **Ing. Sammy Verslype** for the original idea to try and pick a subject where I try to accelerate an algorithm using an FPGA equipped platform.

I would also want to thank my father **Friedel Nollet, Ph.D.** (molecular biologist at the AZ Sint-Jan Hospital in Bruges) for the original idea of implementing the genome mapping problem, as well as fact-checking my thesis for the biology part and providing me with sample data.

This thesis was executed during my time as an Erasmus exchange student at the Brno University of Technology. I would like to acknowledge the BUT for accepting me as an incoming Erasmus student.

Furthermore, thanks to **ir. Joan Peuteman, Ph.D.**, **ir. Hilde Bonte** and **Ing. Michaela Studena** for helping me in the administration needed for the Erasmus exchange.



Summary

Tijdens de laatste decennia hebben biologen grote stappen gezet in het begrijpen van het leven: dieren, mensen en planten. Sinds het opkomen van DNA sequencing technieken is genetica een onderdeel geworden van de biologie. Maar, door de hoeveelheid DNA sequencing data die verwerkt moet worden, is het analyseren van genetische toepassingen erg rekenintensief voor computers, bijvoorbeeld bij analyse van het menselijk genoom, wat bestaat uit 3 miljard baseparen. Een grote fundamentele applicatie binnen de genetica is het "short read genome mapping" of het "short read alignment", wat probeert de locatie van een kort stukje DNA terug te vinden in het volledige genoom. Als er genoeg van deze "reads" gealigneerd kunnen worden, kan hier veel interessante info uit worden afgeleid. Bijvoorbeeld, bij voldoende aligneringen kunnen we het volledige genoom afleiden uit ons bemonsterd DNA. Zo kan men op basis van het aantal reads die aligneren in één genomische regio (wat men ook wel de "reading depth" noemt) te weten komen of er een trisomie van chromosoom-21 (Down syndroom) aanwezig is bij een foetus.

Typisch wordt een "read" vergeleken met het volledige genoom via het Smith-Waterman algoritme. Als resultaat krijgen we dan de positie van deze read terug in het genoom.

Een volledige genoomsequentie in één keer bepalen kan niet, vanwege de technieken die de sequencing machines gebruiken. Deze machines kunnen enkel reads van een korte lengte aan, met een maximum van een paar honderd baseparen ineens. Tegenwoordig wordt er meer en meer DNA gesequenced, en dit groeit exponentieel. Gezien er meer vraag is naar goede aligneringstechnieken, moeten deze ook worden verbeterd om bij te blijven met de stijgende vraag naar DNA sequencing. Maar, meestal worden deze nieuwe technieken enkel geprogrammeerd op de standaardprocessoren van een computer.

In deze thesis zullen we het Smith-Waterman algoritme bestuderen, en uit deze studie leren we dat een implementatie op een "gewone" processor niet de beste optie is. Er bestaan andere elektronische technologieën om dit algoritme op uit te voeren, zoals bijvoorbeeld een FPGA. We zullen een MPSoC gebruiken, die beide een stuk ARM (de "gewone" processor) en FPGA (de gespecialiseerde hardware) aan boord heeft.

In de meeste klinische toepassingen waar dit soort alignering gebruikt wordt, is het aantal reads die gealigneerd moeten worden in de miljoenen. Als ultiem resultaat willen we de "time-to-result" van een klinische test verkleinen, zodat de rekenkracht van de computers niet de bottleneck wordt van de tests.

Eerst werd een softwareversie van het algoritme geïmplementeerd op de ARM processor. Deze

hebben we getest met een sequencing van het SARS-CoV-2 (coronavirus) als een dataset, en vergeleken met resultaten bekomen via een gekend bio-informatische software (Galaxy). Als we de resultaten van beide methoden, de eigen implementatie en deze via Galaxy, vergelijken met elkaar, dan kunnen we dezelfde conclusies trekken. Verder zijn we ook in staat om mutaties te ontdekken in het gemonsterde genoom in vergelijking met onze referentie.

Nadat we dit ook geïmplementeerd hebben op de FPGA gespecialiseerde hardware, hebben we een implementatie die dezelfde resultaten toont als onze softwareversie, maar deze resultaten worden vier maal sneller bereikt.

Abstract

Short read genome mapping is an important application in genomics. It is an algorithm to locate a short read of DNA on the full genome. If enough of such short reads are mapped, some interesting results can be achieved. For example, if enough reads are mapped, we can extrapolate the full genome of the organism and all DNA mutations can be detected. Moreover, the amount of reads in one place (referred to as the reading depth) can give useful information, e.g. the presence of extra DNA like a trisomy of chromosome-21 in Down syndrome patients.

Typically, each read is compared with the whole genome in a local alignment, for example with the Smith-Waterman algorithm (S-W). As an output, we get the position in the human genome and alignment with its score (how well the sequence fits in that spot). This practice is commonly referred to as *Mapping to reference genome*.

A full genome cannot be determined immediately, because the machines that determine this sequence can only handle short reads consisting of a few hundred bases. DNA sequencing machines are currently capable of producing millions of reads per day, and their throughput is growing at an exponential rate. This exponential growth should be accompanied by an improvement in genome mapping techniques, to keep up with the throughput of these machines. However, most of the current software tools used for genome mapping are run on traditional CPUs.

If we analyze the S-W algorithm, we can see that the value of each cell in the matrix only depends on the left-upmost 3 cells. Therefore, it leads us to believe that this algorithm can be accelerated on other hardware solutions such as an FPGA since the S-W algorithm is heavily parallelizable.

In most clinical applications where mapping to a human reference genome is used, the number of reads to be compared with the genome is in the millions, which further increases the demand to speedup the process of mapping the reads in the genome, to decrease the time-to-result.

In literature accelerating short read alignment on FPGAs was described to be 28 times faster in respect to CPUs and 9 times to GPU [20]. showed a speedup of 5.6x to 71.3x dependent on the accuracy of the alignment [24]).

The idea of this thesis was to implement the Smith-Waterman alignment algorithm on an MPSoC, which contains both programmable FPGA hardware and an ARM processor in 1 chip. As a target board, the ZCU 104 evaluation kit was chosen.

As a starting point, a software implementation was implemented on the ARM processor. After running the software implementation with the unmapped sequences from the SARS-CoV-2 (coronavirus)

as a sample set, we obtained a dataset of mapped reads. The reads were also mapped by using the Galaxy (Bowtie 2) online tool. We observed that our implementation is working correctly since the reading depth graphs are approximately the same. Furthermore, we were able to detect and identify several DNA mutations in the coronavirus DNA sequenced by the University of Washington in respect to the reference sequence from Wuhan (China).

Some reads showed different results in comparison with the Galaxy online tool, but this will probably be because the parameters used are a bit different. However, the important part is that the reading depths are the same, as well as the consistently mutated bases marked in the genome that were detected.

After implementing the Smith-waterman matrix fill-in in the FPGA hardware, we achieved speedup of 4.41 in comparison with an implementation running fully on the ARM processor.

Keywords: Smith-Waterman algorithm, short read genome mapping, accelerator, MPSoC, FPGA, HLS

Contents

Acknowledgements	iii
Summary	v
Abstract	vii
Table of contents	x
List of figures	xii
List of tables	xiii
List of abbreviations	xiv
1 Introduction	1
1.1 Introduction	1
1.2 Organization of the chapters	2
2 Background information on DNA and DNA sequencing	3
2.1 Biology and DNA	3
2.1.1 History of genetics and DNA	3
2.1.2 Structure of DNA	3
2.1.3 DNA in the human body	6
2.2 The Human Genome Project	7
2.3 Sequencing	7
2.3.1 The sequencing technology	7
2.3.2 The FASTQ file format	10
3 Methods for DNA sequence alignment	11
3.1 DNA sequence aligning	11
3.1.1 Alignment in general	11

3.2	Local versus global alignment	12
3.3	Commonly used algorithms	12
3.3.1	Needleman-Wunsch	13
3.3.2	Smith-Waterman	13
3.4	Problem definition	17
3.4.1	Mapping to a reference genome	17
3.4.2	The SAM and BAM file format	18
3.4.3	Clinical application	20
4	Platforms for accelerating the Smith-Waterman algorithm	22
4.1	Overview of possible hardware	22
4.1.1	CPU	23
4.1.2	GPU	23
4.1.3	FPGA	23
4.2	Hardware selection	24
4.2.1	Recent advances in High-Level Synthesis	25
4.2.2	Platform communication	26
4.3	HLS and SDSoC learning tools	26
4.3.1	Using Vivado HLS and Xilinx SDK	26
4.3.2	SDSoC	26
5	Software implementation of the Smith-Waterman algorithm	28
5.1	The concept	28
5.2	General overview of the implementation	29
5.2.1	Parameters and Types	29
5.2.2	The code structure	29
5.3	Details of the implementation	30
5.3.1	File interfaces	31
5.3.2	Memory management	33
5.3.3	The alignment	34
5.4	Implementation results	36
5.4.1	The IGV software	36
5.4.2	Sample data	37
5.4.3	Results	37
6	Accelerating the software implementation using HLS	39

6.1	Analysing software performance	39
6.2	Recoding parts of the software to be more hardware friendly	39
6.2.1	Recoding the Cell generation layer	39
6.2.2	Recoding the FillIn layer	40
6.3	Hardware acceleration	42
6.3.1	DMA	42
6.3.2	Unrolling pragmas	44
6.3.3	Comparison with the software	45
7	Conclusion and future research	46
A	Implementation code	52
A.1	Filesystem organization	52
A.2	Code	53

List of Figures

1.1	Megabase means 1 million nucleotide bases. In this graph we can see that the cost of sequencing decreases significantly over time, hence the demand of sequencing increases. This increases is more than Moore's law, which is the reason that mapping techniques should improve to keep up with the demand [22].	1
2.1	The structure of one nucleotide [8](modified from source).	4
2.2	The famous double helix [14].	4
2.3	A short part of a DNA molecule containing a $[5' - ACTG - 3']$ (left) and a complementary strand of $[3' - TGAC - 5']$ (right). These 2 strands are interconnected by hydrogen bonds [12].	5
2.4	the 46 human chromosomes [6].	6
2.5	The order and the sizes of the chromosomes of the human genome as depicted in the IGV software [7].	7
2.6	The enzymatic copying of a string of DNA. The original is unzipped, thus allowing new nucleotide bases to attach to the exposed bases [27].	8
2.7	Sequencing technology used by Illumina attaches a nucleotide with a fluorescent tag to the next base in the read, then it captures a picture to determine the base, and removes the fluorescent tag so a new nucleotide group can bind in the next iteration [27].	9
2.8	From left to right are the pictures taken at each iteration in the flowcell. The color at that specific spot marks which nucleotide has been bound. With the use of some image processing techniques the exact sequence in that spot can be identified [26]. .	9
3.1	Data dependencies in the H matrix	15
3.2	Mapping to a reference genome. The direction of the read is represented by arrows .	17
3.3	Chromosomes of a patient with trisomy 21 (Down Syndrome). The trisomy of chromosome-21 is encircled [21].	20
3.4	A schematic overview of the NIPT test [19].	21
4.1	An overview of the different semiconductor technologies [11].	22

4.2	A cpu processes instructions sequentially. It could be accelerated using pipelining, but the maximum stays 1 instruction per clock cycle [10].	23
4.3	The basic layout of an FPGA [4].	24
4.4	An overview of the hardware available on the ZCU 104 MPSoC evaluation kit	25
5.1	The concept of the implementation: the sequences get mapped to the whole genome	28
5.2	The created types to store the sequence and the reference	29
5.3	An organisation chart of the split up functionalities	31
5.4	The type created to store the genome information.	31
5.5	The type created to store the read information	31
5.6	The type created to store the mapping information, together with the read	32
5.7	The type created to store the information for every cell in the alignment matrix	34
5.8	The layout of the IGV analysing software	36
5.9	Close-up of a few reads in the IGV analysing software. Notice how the color coded bases are substitutions, whereas the matched bases are not shown. Moreover, a gap and insertion is also visible	37
5.10	A comparison of the data obtained by Galaxy (top) and the data obtained by the implementation discussed in this chapter (bottom)	38
6.1	a TCF profile of the software implementation	39
6.2	The optimal processing found by Vermij E. [34]	40
6.3	Data dependencies for generating a cell in the alignment matrix	41
6.4	Schematic of a system using DMA	42
6.5	Latency analyzer in SDSoC. We can see the number of clock cycles spent in the software and the hardware variant of the implementation, as well as the speedup. The speedup is calculated 2 times, since we run the fillIn function twice, once for the forward and once for the reverse sequence. The resource utilization of the FPGA hardware is also visible in this picture	45
7.1	A comparison of the data obtained by Galaxy (top) and the data obtained by the implementation discussed in this chapter (bottom)	47
7.2	The latency analyzer in SDSoC. We can see the number of clock cycles spent in the software and the hardware variant of the implementation, as well as the speedup. The speedup is calculated 2 times, since we run the fillIn function twice, once for the forward and once for the reverse sequence. The resource utilization of the FPGA hardware is also visible in this picture	48
A.1	The used directory structure in the implementation. Directories are colored in yellow.	52

List of Tables

3.1	Classification of DNA alignment algorithms	13
3.2	Similarity matrix example	14
3.3	Example of the initialization of the scoring matrix	14
3.4	Example of a populated scoring matrix	16
3.5	A traceback in S-W. For example, the 13 in the blue colored cell is based on the cell with value 10 in on the left-up diagonal (match). Therefore, the path will jump to the cell with value 10. This is repeated until a 0 cell is hit.	16
5.1	Encoding for the nucleotide bases	29
6.1	The 3 newly created arrays to house the data needed for generating the next diagonal of cells. The '?' in the cD array (red) represents cells currently being generated. Notice that all the information needed to generate the new cell is present in the other 2 arrays	41

List of abbreviations

- A** - *Adenine* - One of the 4 nitrogen bases present in DNA
- ASIC** - *Application Specific Integrated Circuit* - An integrated circuit especially designed for a specific application
- BAM** - *Binary Alignment Map* - The file format used for storing mapped reads, binary
- BLAST** - *Basic Local Alignment Search Tool* - A heuristic algorithm for finding local alignments
- bp** - *base pairs* - 2 nitrogen bases that are connected with hydrogen bonds in DNA. Adenine is connected with Thymine, Guanine with Cytosine
- C** - *Cytosine* - One of the 4 nitrogen bases present in DNA
- cfDNA** - *cell-free DNA* - cell free DNA, which is found in the blood plasma
- CIGAR** - *Concise Idiosyncratic Gapped Alignment Report* - A string that indicates where matches, insertions and deletions occur in a mapped sequence
- CLB** - *Complex Logic Block* - The basic element in an FPGA
- CPU** - *Central Processing Unit* - The integrated circuit present in every computer which is easily reprogrammable.
- DNA** - *DesoxyriboNucleic Acid* - A molecule present in the nucleus of a cell that stores the genetic information for all living organisms
- FAT** - *File Allocation Table* - A technology for organizing file systems
- FPGA** - *Field Programmable Gate Array* - An integrated circuit consisting of programmable logic components
- G** - *Guanine* - One of the 4 nitrogen bases present in DNA
- GPU** - *Graphical Processing Unit* - A semiconductor technology that is specialized in video encoding and decoding
- HDL** - *Hardware Description Language* - A set of commands that can be used to describe how the hardware in an FPGA should be programmed
- hg** - *human genome* - A reference for the full DNA sequence found in the nucleus of the cells from every human
- HLS** - *High Level Synthesis* - A compiler used to translate C code into HDL code
- IDE** - *Integrated development environment* - A software application that provides utilities to a programmer when programming an application
- Indel** - *Insertion or Deletion* - A single term to describe an insertion or deletion in DNA
- MPSoC** - *Multi-Processor System on Chip* - An integrated circuit that contains multiple microprocessors and/or programmable hardware

NGS - *Next Generation Sequencing* - The technique used most often to determine the sequence DNA

NIPT - *Non-Invasive Prenatal Testing* - A test for detecting genetic defects in a foetus

N-W - *Needleman-Wunch algorithm* - An algorithm used for global alignment, which is similar to Smith-Waterman

SAM - *Sequence Alignment Map* - The file format used for storing mapped reads, text-based

SIMD - *Single Instruction Multiple Data* - A technology in CPUs that can manipulate more than 1 attribute with a single instruction

S-W - *Smith-Waterman algorithm* - A variation of the N-W algorithm, adapted for local alignment. It is a dynamic programming technique

sWGS - *shallow Whole-Genome Sequencing* - An experiment to detect gains and losses in DNA material

T - *Thymine* - One of the 4 nitrogen bases present in DNA

USB - *Universal Serial Bus* - An industry standard for connections between a computer and peripherals

Chapter 1

Introduction

1.1 Introduction

Over the last few decades, biologists have made major steps in trying to understand life: animals, humans, and plants. Since the rise of DNA sequencing techniques, genomics has become an emerging field within biology. However, genomic applications are often very computationally demanding, due to the size of the involved datasets, for example when analyzing the 3 billion base pairs of the human genome.

One fundamental application in genomics is short read genome mapping, which attempts to locate a short read of DNA on the full genome. If enough of such short reads are mapped, some interesting results can be achieved. For example, if enough reads are mapped, we can extrapolate the full genome of the organism. Even the amount of reads in one place (referred to as the reading depth) can give useful information, e.g. the presence of a trisomy of chromosome-21 in Down syndrome patients.

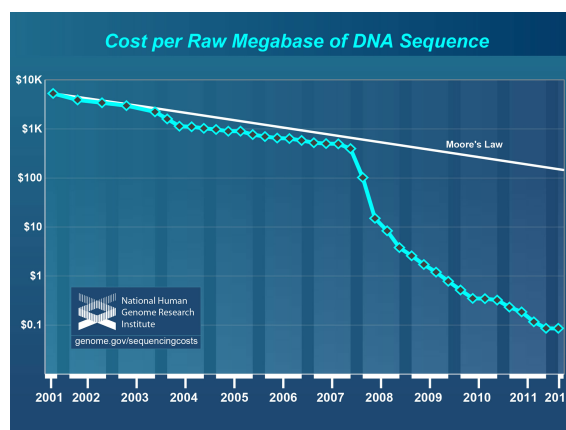


Figure 1.1: Megabase means 1 million nucleotide bases. In this graph we can see that the cost of sequencing decreases significantly over time, hence the demand of sequencing increases. This increases is more than Moore's law, which is the reason that mapping techniques should improve to keep up with the demand [22].

A full genome cannot be determined immediately, because the machines that determine this sequence can only handle short reads consisting of a few hundred bases. These sequencing machines are currently capable of producing millions of reads per day, and their throughput is growing at an exponential rate [22]. This exponential growth should be accompanied by an improvement in genome mapping techniques, to keep up with the throughput of these machines. However, most of the current software tools used for genome mapping are run on traditional CPUs.

This thesis will implement genome mapping on an MPSoC (Multi-Processor System on Chip), which has an amount of programmable logic available to accelerate some aspects of the currently used algorithms. As a sample set, DNA sequencing reads of the PhiX and SARS-CoV-2 viruses will be explored. However, the algorithms discussed in this thesis can be expanded to the full human genome, where there is a high need for accelerating the algorithm and thus reducing the time-to-result of clinical tests.

1.2 Organization of the chapters

The further chapters of this thesis are organized as follows:

Chapter 2: The theoretical background in genetics, molecular biology, and DNA, as well as the sequencing technology.

Chapter 3: The theoretical background regarding alignment, existing alignment algorithms, as well as the problem statement with some example clinical applications.

Chapter 4: The theoretical background on existing implementations of the Smith-Waterman algorithm on different kinds of hardware, as well as the hardware selection process. This chapter also covers the difficulties I had when learning the required programming techniques, covering High-Level Synthesis and SDSoc.

Chapter 5: A detailed description of the developed software implementation, as well as the results achieved with this implementation.

Chapter 6: The analysis of the software implementation and where to accelerate using FPGA hardware. Also, the comparison between the accelerated version and the software version.

Chapter 7: Conclusion and future work.

Chapter 2

Background information on DNA and DNA sequencing

2.1 Biology and DNA

2.1.1 History of genetics and DNA

Genetics For thousands of years, humans have observed the effects of heredity and implemented their knowledge to domesticate plants and animals. However, the science behind heredity was only started to be understood since 1859 with the publication of *on the origin of species* by Charles Darwin.

Around 1865, an Austrian monk and botanist Gregor Mendel, who studied at the university in Brno in the current Czech Republic, published his results on the hybridization studies of pea plants. He is often credited as being the father of modern genetics. In his findings, he implemented the role of *factors* that influence the expression of traits. These factors later became known as *genes* [29].



Gregor Mendel

DNA In 1869, Swiss physician Friedrich Miescher discovered a microscopic substance in the pus of discarded surgical bandages. Later, in 1909, Phoebus Levene named this substance DeoxyriboNucleic Acid (DNA) since it is found in the nucleus of a cell and has acidic properties [18]. The full structure of DNA was discovered by Francis Crick and James Watson at the Cavendish Laboratory at the University of Cambridge [28].

2.1.2 Structure of DNA

DNA, or Deoxyribonucleic Acid, is the molecule that stores the genetic information of all living organisms. It is the information that programs all of the activities in a cell.

Structurally, DNA is a polymer, which means each molecule is built up out of small repeating molecular units. In DNA, these units are called *nucleotides*.

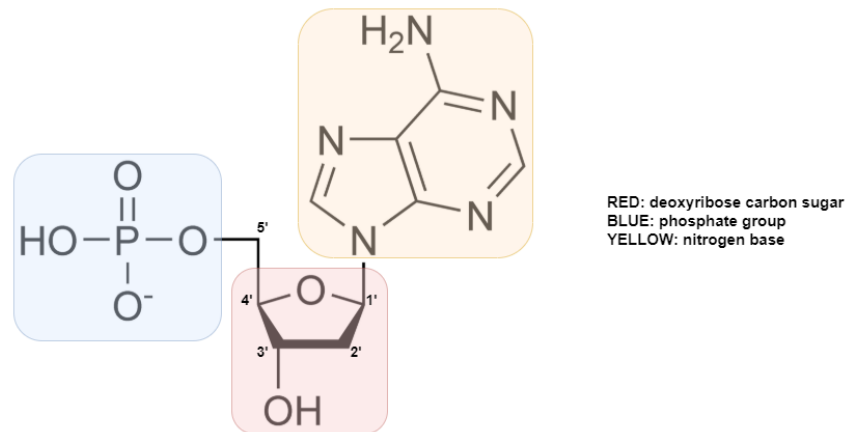


Figure 2.1: The structure of one nucleotide [8](modified from source).

Each nucleotide consists of 3 parts:

1. A carbon sugar molecule called *Deoxyribose*.
2. A phosphate group to connect the Deoxyribose molecules.
3. One of four possible nitrogen bases: Adenine (*A*), Thymine (*T*), Cytosine (*C*) or Guanine (*G*).

It is important to note that in most living organisms DNA does not exist as a single polymer, but rather a pair of molecules that are held tightly together. This is the famous *double helix*.

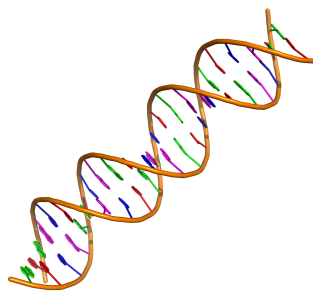


Figure 2.2: The famous double helix [14].

Like in any good structure, there is a need for the main support. In DNA, the sugars and phosphates bond together to form twin backbones. These sugar-phosphate bonds run down each side of the helix, but chemically in opposite directions.

The first phosphate group, at the start of the molecule, connects to the sugar group's 5th carbon (5'). At the end of the structure, the 3rd carbon (3') of the sugar group is unconnected. This makes

a pattern typically noted as $[5' \rightarrow 3']$. Now, since the other molecule in the helix goes in the opposite direction, the pattern of the other backbone is typically noted as $[3' \rightarrow 5']$.

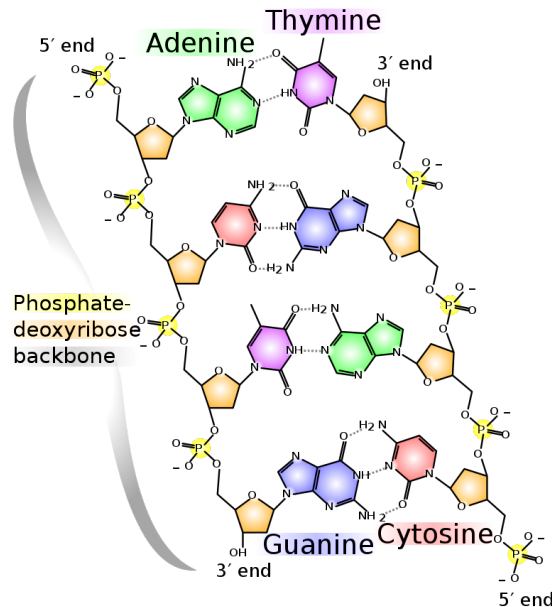


Figure 2.3: A short part of a DNA molecule containing a $[5' - ACTG - 3']$ (left) and a complementary strand of $[3' - TGAC - 5']$ (right). These 2 strands are interconnected by hydrogen bonds [12].

These two long chains are linked together by the nitrogen bases via their relatively weak hydrogen bonds, but there can't just be any pair of nitrogen bases. Adenine can only make hydrogen bonds with Thymine. Likewise, Guanine can only bond with Cytosine. These bonded nitrogen bases are called *base pairs*.

It is the order of these bases, which is also called the *sequence*, that allows this DNA to store useful information. In this way, e.g. *AGGTCCATG* means something completely different as a base sequence than e.g. *TTCCAGATC*.

Since each of the bases in the sequence has only one possible counterpart, you can predict what its matching counterpart will be in the opposite string. For example:

If the following sequence is known



we can deduce the sequence in the other direction as



2.1.3 DNA in the human body

In human cells, DNA molecules can be found in the nucleus of all cells in the body. It consists of 46 very long molecules, which during cell division condense in what we call *chromosomes*. The only exception is in reproductive cells (the egg cell and the sperm cell), which only have 23 chromosomes.

The 23 chromosomes, which make up our whole DNA, are always present in pairs in the cells, making a total of 46 chromosomes. Each time, the pair consists of one chromosome from the father and the other one from the mother.

The 23 chromosome pairs are classified in:

- 22 pairs of autosomal chromosomes, marked 1 to 22 according to the length of the sequence. The longest chromosome (chromosome number-1) is 248,956,422 bases long. The shortest (chromosome number-22) is 50,818,468 bases long.
- In each cell, there is also an X chromosome plus an X or Y chromosome, dependent on the gender (XY for male, XX for female).

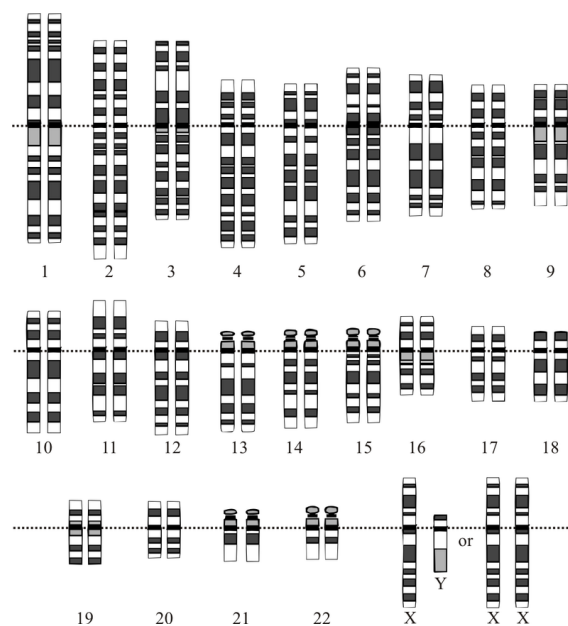


Figure 2.4: the 46 human chromosomes [6].

These chromosomes are packed tightly together in the nucleus of the cell. If all of these 46 chromosomes are put together, this makes about two times 3 billion base pairs. These 3 billion base pairs provide the assembly instructions for pretty much everything inside the cell.

2.2 The Human Genome Project

In the field of Bioinformatics, an important dataset is the *Human Genome*. This is the full DNA sequence found in the Nucleus, ordered from chromosome 1 to 22, followed by the X and Y chromosome.

In October 1990, biologists in the relatively new field of molecular biology started the Human Genome Project. The goal of this project was to determine the sequence of the 3 billion base pairs that make up human DNA. This project was completed and published in 2003. So, nowadays we have a good idea of how the human genome is built up.

The Human Genome is easily found on the internet since it is publically available. One of the most often used assembly is *hg19*, which was published in 2009. Since DNA has only 4 possible bases (A, T, C or G), this can be encoded in a 2-bit representation. If this encoding is used, ideally the Human Genome is approximately 750 megabytes.

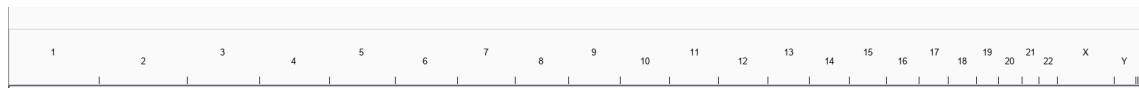


Figure 2.5: The order and the sizes of the chromosomes of the human genome as depicted in the IGV software [7].

2.3 Sequencing

2.3.1 The sequencing technology

The term *Sequencing* is used for all techniques to read and decipher the DNA code from a given snippet of DNA. During the last years, the techniques that sequence human DNA has changed quite a lot. For about 15 years the *Next Generation Sequencing (NGS)* is the technique most often used. The biggest advantage of NGS, in comparison with other techniques, is the speed of the sequencing since it can sequence billions of short DNA molecules in parallel. In practice, this sequencing is most often done by the instruments of the company Illumina, which dominates the market (around 90% market share).

How whole-genome NGS works

1. The DNA to sequence is isolated from the cells. Most often this is the whole genome.
2. In some cases, the isolated DNA can now be copied enzymatically. This step is repeated until there are enough copies of the same DNA. Usually, this is in the millions or billions of copies.

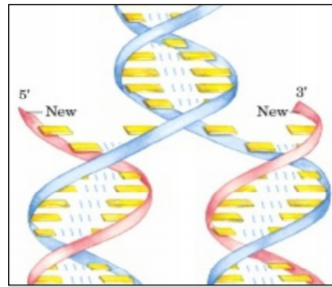


Figure 2.6: The enzymatic copying of a string of DNA. The original is unzipped, thus allowing new nucleotide bases to attach to the exposed bases [27].

3. The full DNA sequence is now broken apart into small DNA molecules (100 to 1000 bases long). This is done using enzymes or high-frequency sound waves.
4. Now the sequencing can start: a *flow cell* is used where these small DNA molecules can bind to a glass surface.
5. Different enzymatic and chemical reactions can now be done on this flow cell through an automatic flow of reagents. The following steps are iterated until the full read has been filled in:
 - (a) The entire flowcell is filled with nucleotides, all with different nitrogen bases. Important is that at each of these nucleotides there is a fluorescent group attached. This also makes sure no other nucleotide can bind.
 - (b) The fluorescent groups have a different color, dependent on the nitrogen base attached (A, G, T or C). At this time a camera picture of the flowcell is taken and stored.
 - (c) After the flowcell is emptied of the loose nucleotides, another reagent flows in this flowcell. This reagent splits the fluorescent group so that in the next iteration a new nucleotide group can bind with the read.

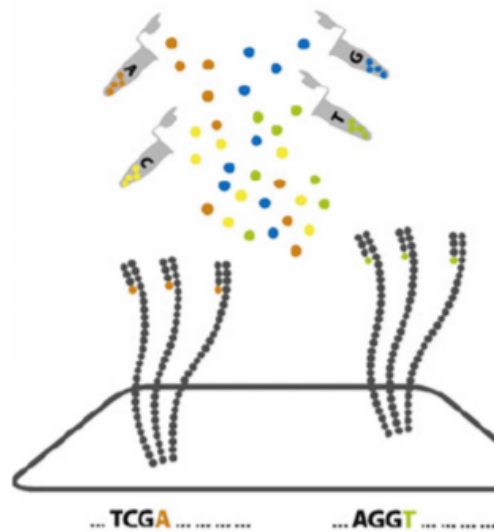


Figure 2.7: Sequencing technology used by Illumina attaches a nucleotide with a fluorescent tag to the next base in the read, then it captures a picture to determine the base, and removes the fluorescent tag so a new nucleotide group can bind in the next iteration [27].

6. After the whole DNA snippets have been filled in, the machine deduces the sequence in the DNA snippet. The pictures that were taken in order during the operation show the colors released in a specific spot, and by extent the attached nitrogen base. By the means of some image processing techniques, it is quite easy to get all the sequences from all the molecules bound on the flowcell. This is called the *Primary processing*.

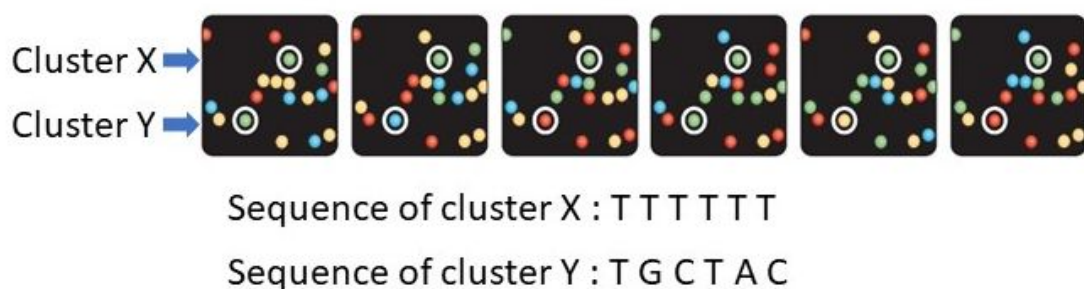


Figure 2.8: From left to right are the pictures taken at each iteration in the flowcell. The color at that specific spot marks which nucleotide has been bound. With the use of some image processing techniques the exact sequence in that spot can be identified [26].

7. In the *secondary processing*, the sequence is trimmed by quality, etc. The operations that are done on the read in this step are outside the scope of this thesis.

As a result of the NGS, we get a (large) file in the FASTQ format.

2.3.2 The FASTQ file format

Since the color of each spot observed in the camera pictures in the primary processing can have a light shift, there is a specific "uncertainty" about the correct base is in that spot. This is called the *quality* of the base.

The *FASTQ* file format has become the de-facto standard as output from sequencing instruments. It is a text-based format for storing both the bases in the sequence and their corresponding quality. A FASTQ file uses four lines per sequenced DNA fragment:

1. A '@' character followed by a sequence ID, plus an optional description. This description mostly contains the coordinate of the spot on the flowcell.
2. The sequence of DNA bases identified by the machine. This is either *A*, *G*, *C*, *T*, or *n* when the base cannot be identified with a specific threshold certainty.
3. A '+' character, optionally followed by the sequence ID (again) and an optional description.
4. The quality values for each respective base in line 2. The length of this line must be the same as the number of bases in line 2.

The quality score in memory is a value in the range 0x21 (lowest quality) to 0x7e (highest quality). Since this value is represented in ASCII in the file format, this ranges from the '!' character to the '~' character. A complete list of the possible values for the quality score can be found below:

```
!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNPOQRSTUVWXYZ
[\\]^_`abcdefghijklmnopqrstuvwxyz{|}~
```

Important to note is that this quality score is logarithmic. Also, the '@' and '+' character are a possible value for the quality, so this will be something to look out for when implementing the interpreter for this file.

A FASTQ file containing a single sequence of a DNA fragment of 55 bases might look like this:

```
@P3018:1114
GCTACTTCCCAAGAAGCTGTTTCAGAATCAGAATGAGCCGCAACTTCGGGATGAAA
+
AAAAAEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEE
EEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEE
```

Keep in mind that a FASTQ file consists of multiple of these sequences, all stacked under each other.

Chapter 3

Methods for DNA sequence alignment

3.1 DNA sequence aligning

The human genome is used as a reference genome for all sequenced human DNA (*hg19* is the most widely used). However, the genetic code of all humans is slightly different, which also holds true for all other organisms. Genetic sequence alignment is the science where you try to align 2 sequences with each other so that the amount of differences is minimal. In this chapter, the most frequently used algorithms are discussed.

3.1.1 Alignment in general

In genetic codes, there are 3 types of differences between the given sequence and the reference:

- Insertion: one or more bases have been added in the genetic code in a specific spot.
- Deletion: one or more bases have been removed from the genetic code in a specific spot.
- Substitution: one or more bases have been substituted by other bases.

Inserts and deletions are often described by a single term, *indel*.

For example: if we want to align the following sequences:

```
Seq1: ATATCGGC
Seq2: ATCG
```

The alignment itself can now be done in different ways. Possible alignments are:

```
Alignment 1
Seq1: AtaTCgGc
Seq2: A--TC-G-
Alignment 2
Seq1: atATCGgc
Seq2: --ATCG--
```

Which alignment that is the actual output, depends on the algorithm and the given parameters (penalty and similarity scores). The '-' character represents a base that is not present.

Keep in mind, there is no one "correct" alignment. The core of the alignment algorithms is the same each time, but the parameters of these algorithms are changed depending on the application.

3.2 Local versus global alignment

To explain the difference between local and global alignment, we can take a look at the following example:

```
The 2 DNA sequences:
Seq1: TCCCAGTTTGTGTCAGGGGACACGAG
Seq2: CGCCTCGTTTTCAGCAGTTATGTGCAGATC

Alignment 1 :
Seq1: -----tccCAGTT-TGTGTCAGgggacacgag
Seq2: cgcctcgttttcagCAGTTATGTG-CAGatc-----

Alignment 2 :
Seq1 : tcCCa-GTTTgt-GtCAGggg-acaC-GA-g
Seq2 : cgCCtcGTTTtcaG-CAGttatgtgCaGAtc
```

Both alignments are valid but different. The first alignment is *locally aligned*. This means that the similarities are prioritized in the same region, with the similarity as high as possible. On the other hand, the second alignment is *globally aligned*. Here the similarities over the full length of the sequences are used for the alignment.

In practice, the local alignment is used most often, since it can give you information of 2 sequences that do not have (approximately) the same length.

3.3 Commonly used algorithms

In this section, we will take a look at some algorithms that are used most often for DNA sequence alignment.

The most used algorithms are often categorized in 2 ways:

- local alignment versus global alignments (see Section 3.2)
- dynamic algorithms versus heuristic algorithms: dynamic algorithms are exact but slow and computationally demanding, whereas heuristic algorithms are faster but are approximations and the best alignment is not guaranteed.

A schematic view of some algorithms that are used in practice can be found below:

	Dynamic programming	Heuristic programming
Local alignment	Smith-waterman	FASTA, BLAST
Global alignment	Needleman-Wunsch	X

Table 3.1 Classification of DNA alignment algorithms

Keep in mind, a lot of other claimed "algorithms" (for example BFAST, ...), are accelerated versions of the Smith-Waterman algorithm.

3.3.1 Needleman-Wunsch

Needleman and Wunch proposed a new algorithm for genetic sequence alignment in 1970, now known as the *Needleman-Wunsch* (N-W) algorithm [25]. Since this algorithm is meant for global alignment, which is rarely used in practice, further discussion of the algorithm will not be done. However, N-W has a lot of similarities with the Smith-Waterman algorithm, discussed in the next section.

3.3.2 Smith-Waterman

The *Smith-Waterman* (S-W) algorithm was first proposed by Temple F. Smith and Michael S. Waterman in 1981[31]. It is a variation on the N-W algorithm, adapted for local alignment. It is a dynamic programming technique, so an optimal local alignment is guaranteed.

The core of the algorithm is a matrix fillup with data dependencies on the previous cells. An analysis of the algorithm can be found below:

1. Symbols used in the analysis:

Let sequences $A = a_1a_2a_3 \dots a_n$ and $B = b_1b_2b_3 \dots b_m$ be the sequences that need to be locally aligned. Here n and m are the lengths of the sequences A and B

2. Define the parameters:

- Define $s(a, b)$ be the *similarity matrix* (sometimes also called the *substitution matrix*) for the two sequences. It is used for "rewarding" when $a_i = b_j$ and "punishing" when $a_i \neq b_j$.

In the most general way, we define the similarity score as a matrix of values, e.g.:

	A	C	G	T
A	3	-3	-3	-3
C	-3	3	-3	-3
G	-3	-3	3	-3
T	-3	-3	-3	3

Table 3.2 Similarity matrix example

Often, there are only 2 scores used (equal or not equal). In this case, the similarity matrix can be condensed as follows:

$$s(a_i, b_j) = \begin{cases} +3, & a_i = b_j \\ -3, & a_i \neq b_j \end{cases}$$

- Define d as the *gap penalty* which regulates the score for an insertion or a deletion. This parameter can be:

- *Linear*: The penalty is constant. So, in this case, it doesn't matter if the previous was also a gap or not.
- *Affine*: An affine gap penalty considers gap opening and extension separately. For the sake of simplicity, my further implementation will not include this refinement of the algorithm. The algorithm can be extended to include this affine gap penalty, but this would make the algorithm more complex and we would limit our ability to develop possible accelerations. It is also expected to affect the DNA mapping on a reference genome. However, if we assume the size of the gaps are small, there won't be much difference in result in comparison with the linear penalty score.

3. The initialization: We construct a scoring matrix H with size $(n + 1) \times (m + 1)$. The first column and first row are initialized with 0.

For example: if we want to align the sequences A = *GGTTGACTA* and B = *TGTTACGG*:

		T	G	T	T	A	C	G	G
	0	0	0	0	0	0	0	0	0
G	0								
G	0								
T	0								
T	0								
G	0								
A	0								
C	0								
T	0								
A	0								

Table 3.3 Example of the initialization of the scoring matrix

4. Matrix fill in: We fill in the matrix using the following formula:

$$H_{ij} = \max \begin{cases} H_{i-1,j-1} + s(a_i, b_j), \\ H_{i-1,j} - d, \\ H_{i,j-1} - d, \\ 0 \end{cases}$$

If we keep in mind that the value of a cell may never be lower than 0, we can represent the data dependencies in the following schematic:

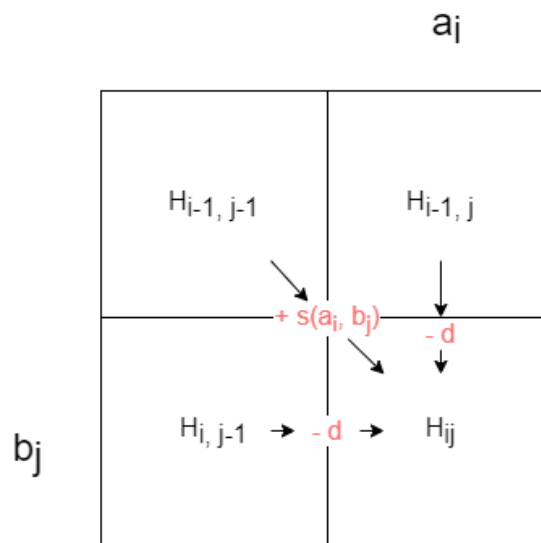


Figure 3.1: Data dependencies in the H matrix

Where $s(a, b)$ and d are the parameters of the algorithm. If we use the following values as an example:

$$s(a_i, b_j) = \begin{cases} +3, & a_i = b_j \\ -3, & a_i \neq b_j \end{cases} \quad \text{and} \quad d = 2$$

We can now fill up the scoring matrix H :

		T	G	T	T	A	C	G	G
		0	0	0	0	0	0	0	0
G		0	0	3	1	0	0	3	3
G		0	0	3	1	0	0	3	6
T		0	3	1	6	4	2	0	4
T		0	3	1	4	9	7	5	3
G		0	1	6	4	7	6	4	8
A		0	0	4	3	5	10	8	6
C		0	0	2	1	3	8	13	11
T		0	3	1	5	4	6	11	10
A		0	1	0	3	2	7	9	8

Table 3.4 Example of a populated scoring matrix

5. Traceback: We start at the cell with the highest score in the matrix H . Starting here we only move left, up or diagonally (left-up) to the cell on which the value in the cell was based until we hit a cell with value 0.

		T	G	T	T	A	C	G	G
		0	0	0	0	0	0	0	0
G		0	0	3	1	0	0	3	3
G		0	0	3	1	0	0	3	6
T		0	3	1	6	4	2	0	4
T		0	3	1	4	9	7	5	3
G		0	1	6	4	7	6	4	8
A		0	0	4	3	5	10	8	6
C		0	0	2	1	3	8	13	11
T		0	3	1	5	4	6	11	10
A		0	1	0	3	2	7	9	8

Table 3.5 A traceback in S-W. For example, the 13 in the blue colored cell is based on the cell with value 10 in on the left-up diagonal (match). Therefore, the path will jump to the cell with value 10. This is repeated until a 0 cell is hit.

From this traceback we can now deduce the following alignment:

```
GTT-AC
|||||
GTTGAC
```

This alignment is the output of our algorithm.

3.4 Problem definition

3.4.1 Mapping to a reference genome

From the DNA sequencing machines, we get a big amount of reads in the FASTQ format. We should note that all these reads are worthless without a proper interpretation.

In most cases, the first step in the analysis of the reads is knowing from which part of the genome it's derived. Typically, the read is compared with the whole genome in a local alignment, for example using the Smith-Waterman algorithm. As an output, we would get the position in the human genome and an alignment with its score (how well the sequence fits in that spot). This practice is commonly referred to as *Mapping to reference genome*.

Since the reads from DNA sequencing machines are 75 to 300 bases long, and the whole human genome is approximately 3 billion bases, this comparison is computationally a very intensive task. If we analyze the S-W algorithm (as we have done in Subsection 3.3.2), we can see that the value of each cell in the matrix is only dependent on the left-upmost 3 cells. Therefore, it leads us to believe that this algorithm can be accelerated on other hardware solutions such as an FPGA (which will be discussed in Chapter 4) since S-W is heavily parallelizable.

In most clinical applications where mapping to a human reference genome is used, the number of reads to be compared with the genome is in the millions.

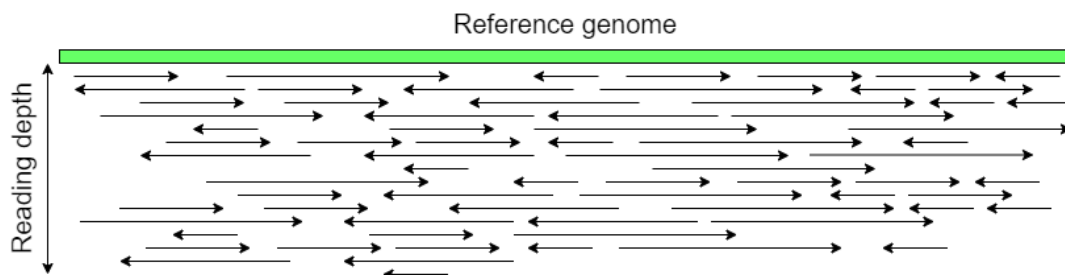


Figure 3.2: Mapping to a reference genome. The direction of the read is represented by arrows

Since the read can be from the complementary DNA molecule in the double helix, the sequences can be in forward $[5' \rightarrow 3']$ direction, or in complementary $[3' \rightarrow 5']$ direction. To transform that read to the used reference genome direction we need to perform the following changes to the read to make its reverse complementary:

1. The bases should be changed to their corresponding base in the base pair;
2. The sequence should be reversed.

We have no way of knowing in which direction the read is taken, both the forward and the backward possibility should be compared with the reference and the best-aligned version of these two should be chosen.

Please note, in most normal cases we can assume the distribution of reads is practically uniform.

Therefore, each base in the human genome will be covered by a statistically expected amount of reads. This amount is referred to as the *reading depth*.

3.4.2 The SAM and BAM file format

As a convention, the output of mapping algorithms is in a *SAM* (Sequence Alignment Map) or *BAM* (Binary Alignment Map) file format. The BAM file format is just a compressed version of the SAM format, but the SAM format is more readable, which makes it easier for troubleshooting. There are many tools to transform a SAM to BAM file already, so in this thesis, we will focus on the SAM format. A SAM file consists of two sections: a header and an alignment section [13].

3.4.2.1 Header

The header is used for information that is independent of the alignments, such as the name of the used algorithm, reference genome, used commands during generation, etc. The header must be at the beginning of the file, before the alignment section. Each line of the header field must start with an '@' character, so these lines are easily distinguished from lines in the alignment section.

3.4.2.2 Alignment Section

Each line in the alignment section represents one mapped sequence and consists of 11 mandatory and some optional fields, which are separated with a tab.

The 11 mandatory fields are:

1. **Qname**: this is the name of the query (the sequence to match) and can be found on the first line of the FASTQ file, which is the input to the mapping algorithm.
2. **FLAG**: a combination of bitwise flags where each bit has a specific interpretation. It consists of 11 bits, but for basic alignments only 2 fields are important:
 - bit 2 (binary: 00000000X00, integer value 4) is set to 1 if the sequence is unmapped or the map is not found
 - bit 4 (binary: 000000X0000, integer value 16) is set to 1 if the sequence has been mapped as its reverse complement.
3. **Rname**: the name of the reference genome. This can be found on the first line of the FASTA file, where the reference genome is stored. If the read is unmapped, this field contains a '*' character
4. **Pos**: the position of the leftmost base in the alignment. Keep in mind that the indexing of the reference starts with 1 for the first base.

5. **MapQ**: the mapping quality, which indicates how good the sequence fits in the specific position. This can be any value between 0 and 254. Value 255 is a reserved value to represent an unavailable quality.
6. **CIGAR**, which stands for *Concise Idiosyncratic Gapped Alignment Report*. It is a string that indicates where the matches (M), insertions (I), and deletions (D) occur. For example, if the CIGAR states *3M1I6M2D10M* this means from left to right: 3 matches, then an insertion, followed by 6 matches, 2 deletions, and finally 10 matches. In case the sequence is unmapped, this field should be filled with a '*' character
7. **Rnext**: reference sequence name of the primary alignment of the text read in the template. For this thesis, we will fill this field with a '*' character.
8. **Pnext**: the position of the primary alignment of the next read in the template. For this thesis, we will fill this field with a '0' character.
9. **Tlen**: the observed template length, from the first till last mapped base. For this thesis, we will fill this field with a '0' character.
10. **Seq**: the full sequence. This can be found in the FASTQ file on the second line.
11. **Qual**: the qualities of the sequence, also given by the FASTQ file on the fourth line.

An example of one line in a SAM file:

SRR11	0	MN98	25	254	7M	*	0	0	GTAAAG	BBBBBCB
-------	---	------	----	-----	----	---	---	---	--------	---------

In this example:

- **Qname**: the name of the sequence: *SRR11*.
- **FLAG**: this read is matched in the $[5' - > 3']$ direction.
- **Rname**: the name of the reference genome is *MN98*.
- **Pos**: the match is found at the 25th base of the reference genome.
- **MapQ**: the mapping quality is 254.
- **CIGAR**: *7M*, so a perfect match.
- **Rnext, Pnext and Tlen**: data not provided.
- **Seq and Qual**: the sequence is *GTAAAG* with quality *BBBBBCB*.

3.4.3 Clinical application

In virtually all sequencing applications DNA alignment and reference mapping is needed. Applications which involve whole genome sequencing in particular are hampered by a long computational analysis time.

We will discuss two clinical applications as examples to genome mapping.

1. **NIPT (non-invasive prenatal testing), a test for detecting genetic defects in a foetus.**

During or after conception, DNA can be lost or gained in the fertilized egg cell. This can result in a severe syndrome of the child. For example, Down syndrome is caused by a trisomy of chromosome 21. Normally all chromosomes are present twice in each cell, one from the mother and the other from the father. In Down syndrome patients something went wrong during cell division at the very early stage of development, and the fetus has in its cells three times chromosome 21. Because chromosome 21 is quite small and does not contain that many genes, the child can survive, though with typical mental and clinical problems [15].

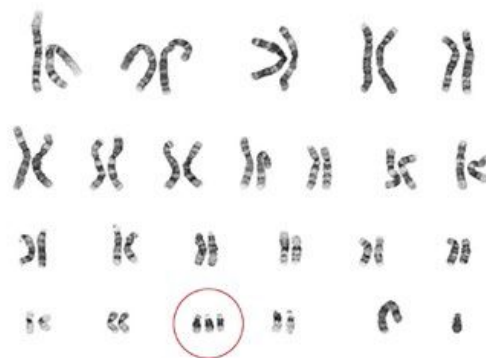


Figure 3.3: Chromosomes of a patient with trisomy 21 (Down Syndrome). The trisomy of chromosome-21 is encircled [21].

Before high throughput DNA sequencing technologies were available like they are today, testing if a fetus has a trisomy-21 could only be done by taking a small amount of amniotic fluid (fluid around the fetus). However, to obtain this fluid there was a need for a risky invasive procedure (called *amniocentesis*) leading sometimes to the termination of the pregnancy.

It is known that small amounts of DNA of the fetus are present in the blood of the mother, in the cell-free DNA (*cfDNA*) which we find in the blood plasma (the clear, aqueous part of the blood). The blood plasma is used by our body to transport 'waste', including DNA from cells that were broken down. When fetal cells die, which is a normal process, the building blocks of these cells are transported in the plasma of the blood from the mother, included small DNA fragments from the fetus [19].

NIPT (non-invasive prenatal testing) is used to analyze DNA derived from the mother's blood. A large number of short *cfDNA* fragments are sequenced at random. Then, each sequence is mapped to the whole human genome to find out where it comes from. Finally, the distribution

of these reads is calculated. If we observe a higher frequency of reads as compared to normal individuals coming from chromosome-21, it is almost certain that the fetus has Down's syndrome [19].

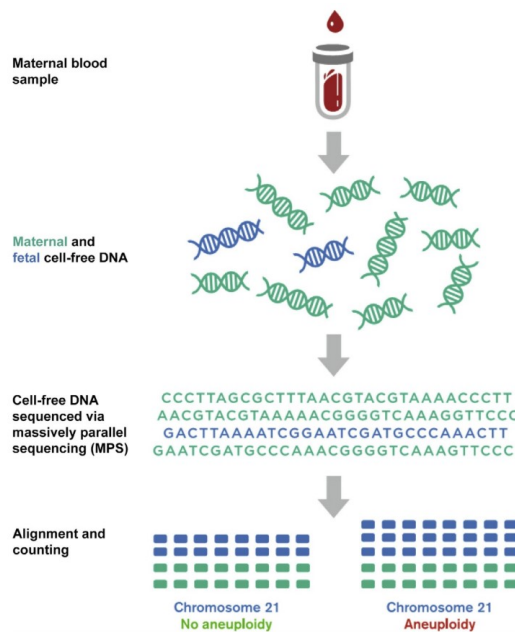


Figure 3.4: A schematic overview of the NIPT test [19].

Using the same method, we can also find other defects in the number of chromosomes. For example trisomy 18 (Edwards syndrome), trisomy 13 (Patau syndrome), or even in the sex chromosomes, such as XXY (Klinefelter syndrome) or lack of a second X or a Y chromosome (Turner syndrome) [19].

2. Shallow whole-genome sequencing of tumor DNA.

It is a known fact that damaged DNA can lead to tumor development. This damage can be single bases changes but can also be a loss or gain of large DNA sequences where important genes are located. When someone is diagnosed with cancer, the knowledge of which DNA regions are lost or gained can be important to decide on treatment.

A relatively new technique to detect all gains and losses of DNA material in one single experiment is shallow whole-genome sequencing. The technique is performed as follows: DNA from the tumor is fragmented (it is broken in small pieces, eg. by a fragmentase enzyme or by high-frequency sound). These pieces are sequenced randomly, and with a mapping algorithm to the reference genome, the over- or underrepresentation of reads (as compared with a normal sample) indicates if regions of the DNA have changed, and which regions these are [30].

This technique is currently being optimized and validated in the laboratory of the AZ Sint-Jan hospital (in Bruges, Belgium) for analysis of DNA gains or losses important for diagnosis, follow-up and therapy of leukemia's, lymphoma's and some solid tumor types.

Chapter 4

Platforms for accelerating the Smith-Waterman algorithm

As we have discussed in Subsection 3.3.2 (analysis of the Smith-Waterman algorithm) the value of each cell in the matrix is only dependent on the left-upmost 3 cells. Therefore, it leads us to believe that this algorithm can be accelerated on other hardware solutions that are better equipped for parallelism than a normal CPU.

4.1 Overview of possible hardware

When designing a processing unit based on the semiconductor technology, it is always a tradeoff between efficiency and flexibility. For example, the CPU in a PC is highly flexible so that it can run any set of instructions without a lot of intervention between context changes. On the other hand, if a certain algorithm or instruction set should be executed as fast as possible, the FPGA's and ASICs are the best choices. At the extreme, an ASIC or *Application Specific Integrated Circuit* can be chosen, but this means a complete chip should be designed from scratch just for this specific application. However, an ASIC is extremely expensive to design and build, especially for small production quantities. In figure 4.1 an overview of the different options can be found.



Figure 4.1: An overview of the different semiconductor technologies [11].

4.1.1 CPU

The power of the CPU lies in its flexibility: it can be very easily programmed with new instructions in a short time. However, if the CPU would only run one specific algorithm for its whole lifetime, it would be terribly inefficient. Also, CPUs work sequentially, which is less suitable for algorithms that demand a large number of computations which could be done in parallel.

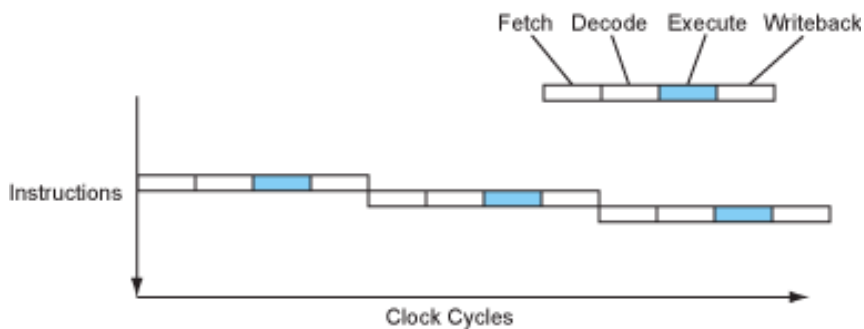


Figure 4.2: A cpu processes instructions sequentially. It could be accelerated using pipelining, but the maximum stays 1 instruction per clock cycle [10].

Since the original implementations of the S-W algorithm are on a CPU, it became clear that a CPU was not the most suitable platform since it consists of a lot of operations that can be done in parallel. However, since the rise of SIMD, CPU's speed for parallel operations have improved substantially. *SIMD* stands for *Single Instruction Multiple Data* and makes it possible to manipulate more than 1 attribute of data with a single instruction, although it is the same instruction on all the data. CLC bio, a Danish company specialized in bioinformatics, has been able to achieve impressive speedups with a software implementation using SIMD, closing in on 200x [1].

4.1.2 GPU

A GPU or *Graphical Processing Unit* is a semiconductor technology which is specialized in video encoding and decoding. Since video encoding and decoding are actually glorified matrix manipulations, a GPU could also be used to manipulate all kinds of matrices. Since the Smith-Waterman algorithm is a big matrix fill in, it leads researchers to believe that it could be accelerated on a GPU. In 1997, an implementation of S-W on a GPU was published which achieved a speedup of 2x over all previous software implementations [16].

4.1.3 FPGA

An FPGA, or *Field Programmable Gate Array*, is an integrated circuit consisting of programmable logic components. These logic components can be programmed as any logic function, such as an AND, XOR, etc. In most FPGA other elements are also found, such as memory blocks, DSP blocks, etc.

Most basic FPGA's, contain the following items:

1. CLB's, or *Complex Logic Blocks*, consisting of a *LookUp Table* (LUT) and a flipflop. A LUT can be programmed so that it contains any type of logic function.
2. Programmable Interconnects have to connect the CLB's into a bigger circuit, which is also called *routing*. This routing has the most influence on delays, and are also responsible for most errors.
3. I/O blocks, which connect the internal logic inside with the outside pins of the FPGA. Most can be configured as input, output, or bidirectional.

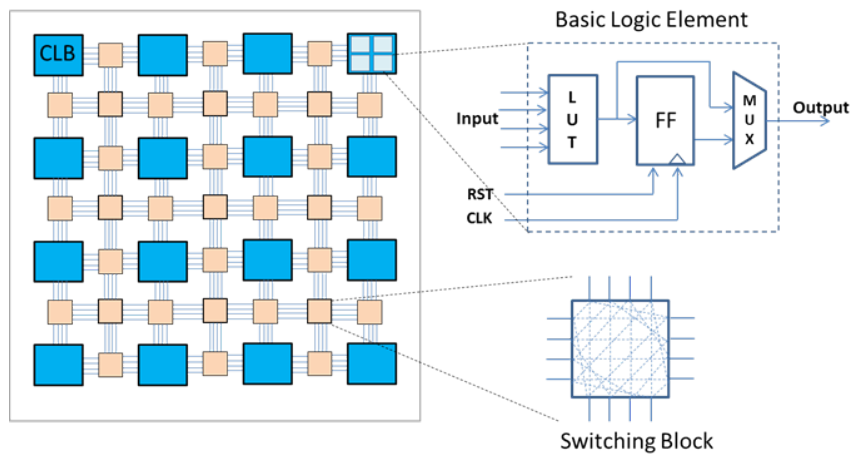


Figure 4.3: The basic layout of an FPGA [4].

For an implementation or design of a circuit which should be loaded in an FPGA, a *Hardware Description Language* (HDL) is often the only practical choice to implement such a system, since drawing the circuits with a CAD program or by hand would take a very long time.

In a paper from 2007, an implementation of S-W with an FPGA (Virtex-4) achieved a speedup of up to 100x in comparison with a 2.2GHz Opteron processor [32]. A few companies have also made some implementations on FPGA in the past, e.g. Cray Inc. [2], TimeLogics [3], ...

A master's thesis from 2011 by Vermij E. [34] has a detailed analysis of an FPGA based Smith-Waterman implementation. In other [17] papers, it was found that the performance per Watt level for an FPGA implementation is better than a GPU or CPU by a factor of 12-21 times.

4.2 Hardware selection

As the hardware the ZCU 104 evaluation kit was selected, which has an MPSoC onboard. An MPSoC (or *Multi-Processor System on Chip*) is an integrated circuit that contains multiple microprocessors. This means that both a processor and programmable hardware is available.

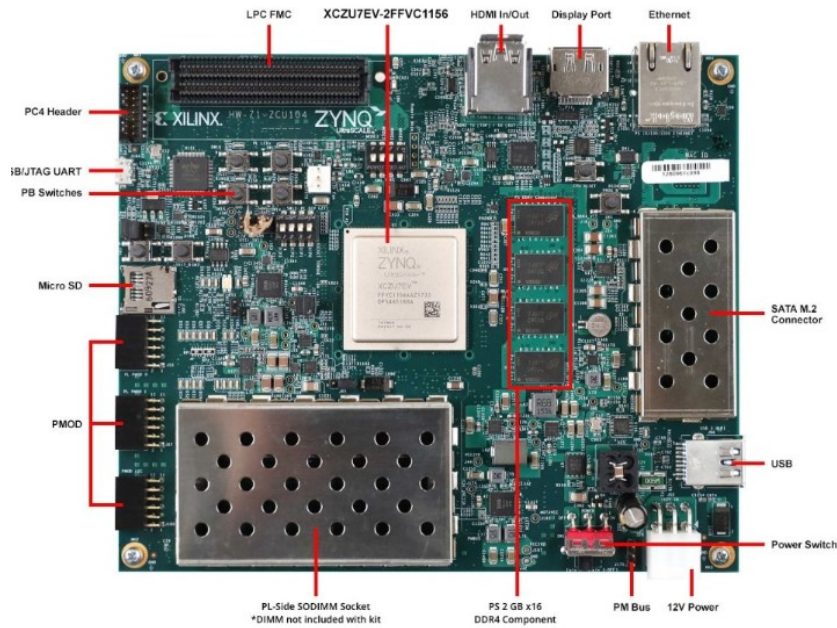


Figure 4.4: An overview of the hardware available on the ZCU 104 MPSoC evaluation kit

Reasons for choosing the ZCU 104 evaluation kit:

1. An MPSoC is present, which will be used to the host application;
2. Both a USB and Ethernet interface is available, which makes it easy to communicate with the target board;
3. An operating system can be run on the board (a Red Hat Linux distribution). This means we won't have to worry about implementing FAT or Ethernet stacks;
4. 16 times 2 gigabytes of RAM is available, which can be useful when implementing big structures, like the alignment matrix;
5. It was available at the Brno University of Technology.

4.2.1 Recent advances in High-Level Synthesis

When programming an FPGA, an HDL is often used. However, if a specific implementation already exists in a programming language, it often has to be reimplemented from scratch in HDL to load it on the FPGA. Therefore people have been trying to make a compiler that compiles normal C code into HDL. This "compiler" is called HLS, or *High-Level Synthesis*. In recent years, HLS became popular as an alternative for designing and implementing a complex system that should be run on an FPGA.

During research in the known literature, no implementation using HLS for the Smith-Waterman algorithm was found. That's why we believe that this is still an unexplored area.

4.2.2 Platform communication

Initially, it seemed important that there is good communication between the board and the host to load and unload sequences since we didn't want the communication to become the bottleneck. However, when using S-W for genome mapping, this won't be an issue since the time it takes to map a read takes a lot longer than the time to transfer it.

In the end, no direct communication with the board was implemented. It seemed sufficient to just load the reads with the genome on an SD card, and insert this in the board. After the mapping process, the mapped reads are available on the same SD card. However, if this program would be used in practice, it doesn't seem practical to constantly change out the SD card. Therefore, another solution should be found. A possible solution could be using the Ethernet stack available as part of the OS, so we could easily use FTP for on and off loading from and to the board.

4.3 HLS and SDSoC learning tools

Disclaimer: The purpose of this section is to show the learning tools I used to achieve a suitable implementation for the genome mapping problem. New programming techniques such as HLS and SDSoC were familiarized. This section might be less applicable if the reader is interested in the science and implementation approach of the genome mapping problem, but can be interesting if the reader is also new to the mentioned programming techniques.

4.3.1 Using Vivado HLS and Xilinx SDK

Learning HLS in examples To learn HLS, I used learning materials sent to me by Tomáš Martínek, who works at the Brno University of Technology [23]. It contains a theory part, and also hands-on lab examples. These labs were worth exploring in this thesis since they contain most concepts of HLS.

Learning how to program target board At first, the Xilinx SDK was used to program the board. However, it was found to be unpractical to use, because after programming a "Hello World" application it became clear that the easiest way to program it this way would be bare-metal. Therefore, a FAT or Ethernet stack ourselves would need to be implemented. Therefore the SDSoC IDE was used for programming the target board, which structures the application on top of an operating system. This operating system provides the Ethernet and FAT stacks.

4.3.2 SDSoC

SDSoC is an IDE developed by Xilinx which is specialized in programming MPSoCs. It is based on the Eclipse IDE, so most of its features are known by most programmers.

The strength of SDSoC lays in the ability to easily transfer functions from software to programmable logic. It can be done with the click of a button. Then, the functions marked for hardware (written

in C) will be fitted in the programmable logic using the HLS compiler. However, the syntax is not always accepted since HLS cannot implement every possible programming technique in C yet.

As mentioned earlier, in this implementation we will work on a Linux distribution.

Learning process on matrix multiplication example in SDSoC To learn MPSoC and the SDSoC IDE, Xilinx' online available materials were used, available at their GitHub page, which includes some hands-on assignments. The assignments use a matrix multiplication example, preinstalled with SDSoC. A weblink to this GitHub page can be found in the references [33].

Chapter 5

Software implementation of the Smith-Waterman algorithm

This chapter will cover an approach to genome mapping using the Smith-Waterman algorithm. The algorithm is a very arduous task for the computer, but it is more precise than heuristic methods such as FASTA and BLAST.

5.1 The concept

For genome mapping a sequence of 75 to 300 bases in length should be mapped to the reference genome. The idea behind this mapping and the applications is thoroughly described in chapter 3.

In some implementations of the genome mapping problem, there is often a specific algorithm that can find some candidate locations. A small S-W is then performed on these candidate locations to find the best fitting one. However, in this implementation, it was decided to compare the sequences with the whole genome instead of only a few candidate locations. This will yield more precise results, but it is also more computationally demanding. However, the current implementation is also applicable for read alignment to a selected region of a genome or a set of genomes from different organisms.

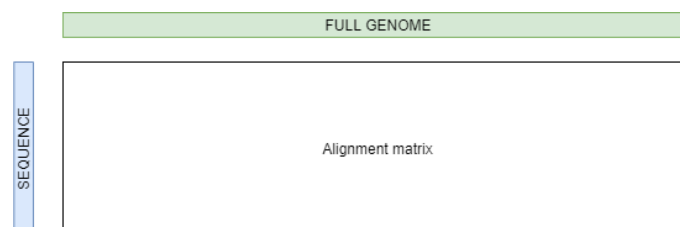


Figure 5.1: The concept of the implementation: the sequences get mapped to the whole genome

5.2 General overview of the implementation

5.2.1 Parameters and Types

Nucleotide base First of all, it seemed important to define the nucleotide bases. There are only 4 possible bases (*A*, *C*, *G* and *T*), so 2 bits are enough. The following coding was chosen:

Base	A	C	G	T
Code	00	01	10	11

Table 5.1 Encoding for the nucleotide bases

To store these bases the `uint8_t` type from the `stdint` library was used. It is 1 byte in size which is the smallest available type in the C language. If more time would have been available, and an implementation with the full human genome would have been made, it might be a good idea to define a specific type consisting of only 2 bits, which would be a lot more memory efficient.

DNA sequence type To easily keep track of a sequence in the program, a type was created which can hold a sequence of DNA. It consists of 2 parts: the length of the sequence and a pointer to the first base in memory. Since the genome can have a length of more than 3 billion bases, and the sequence is only a maximum of 300 bases in length, it made sense to use a separate type for the reference and the sequence.

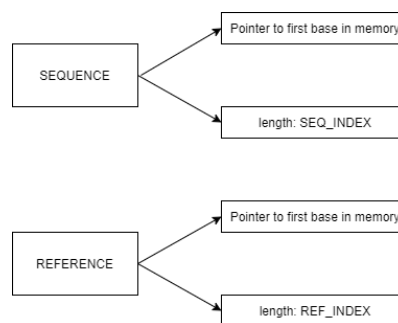


Figure 5.2: The created types to store the sequence and the reference

To index these "arrays" of bases, a new type was created for both the sequence indexing (*SEQ_INDEX*) and the reference indexing (*REF_INDEX*). Since an index to the sequence can be as high as the *length* − 1, it made sense to make the length attribute out of these newly created index types.

5.2.2 The code structure

1. Allocate some memory for the following data:
 - (a) The reference genome, which can be quite large;

- (b) current read;
 - (c) reverse complementary of current read;
 - (d) matrix for during the alignment. The amount of memory that should be allocated to this matrix is equal to the size of the sequence times the size of the reference.
2. Initialize the first row and first column of the matrix on 0, to prevent edge cases when performing the Smith-Waterman Algorithm;
 3. Load the reference genome from the FASTA file;
 4. Open the FASTQ file for loading unmapped reads, and the SAM file to store the reads once they are mapped;
 5. For every read in the FASTQ file, we perform the following operations:
 - (a) Load the next read from FASTQ file;
 - (b) Perform the alignment. (see Subsection 5.3.3);
 - (c) write the mapped read to the SAM file;
 6. Close the FASTQ and SAM files;
 7. Free the reserved memory again.

5.3 Details of the implementation

As in most implementations in software, it was decided to split up the functionality of the program in multiple blocks. We have 3 big blocks:

1. Interfacing with the memory: reserving memory for the reference, the sequence, and the alignment matrix.
2. Interfacing with the files and the filesystem: Since the FASTA, FASTQ and SAM files are in a specific format, it made sense to build interpreters from and to these formats.
3. The alignment itself, which is the core of the program.

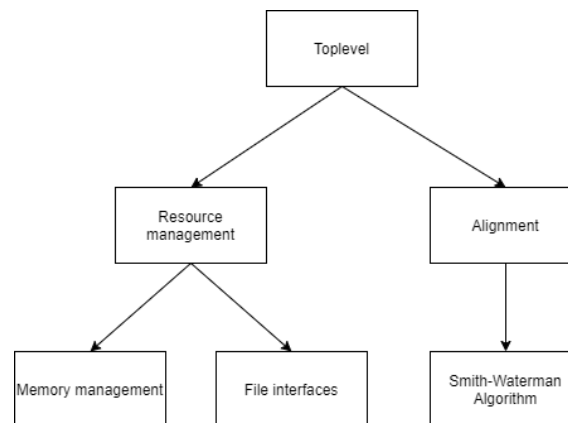


Figure 5.3: An organisation chart of the split up functionalities

5.3.1 File interfaces

5.3.1.1 Parameters and types

Since we need to interface FASTA, FASTQ, and SAM files, it seemed appropriate to create 3 custom types. For the FASTA file, which stores the genome, the type *GENOME* was created. In the case of the FASTQ file and the SAM file, the respective types *READ* and *MAPPED_READ* were created. Notice that the attributes of the types match the information to interface the files.

The information stored in the *GENOME* type is the name of the genome (*Rname*) and the reference.

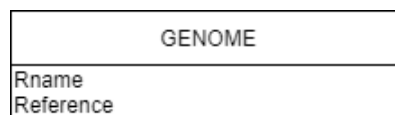


Figure 5.4: The type created to store the genome information.

In the *READ* type, the current sequence is stored together with its quality string and its name.

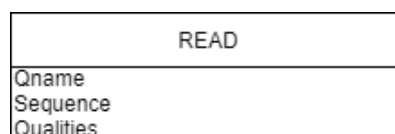


Figure 5.5: The type created to store the read information

The *MAPPED_READ* type contains all the information needed to write a full line in the SAM file, which is the output of the program. Mark, the read itself is also stored in this type since all the information in the read will also be written to the SAM file.

MAPPED_READ
Read Rname Flag Position MapQ CIGAR Rnext Pnext Tlen

Figure 5.6: The type created to store the mapping information, together with the read

5.3.1.2 Code structure

First of all, the representation of the bases in the files (which is in the ASCII character format) should be transformable to the BASE type in the program. Because of this, conversion functions were created to change the character to base or vice versa.

FASTA interface The function of this part is to load the genome from the FASTA file into the created *GENOME* type. The FASTA file format is a simple text-based way of storing a genome. On the first line is a '*^*' character followed by the name of the genome (*Rname*). Starting from the second line to the end of the file the genome is stored. Often, this genome is split up using spaces or in multiple lines, so when coding the interpreter we need to make sure to skip these whitespace characters.

For better code readability, the genome loading is split into 2 functions that are executed in order:

1. loading the genome info
2. loading the genome into the REFERENCE type.

FASTQ interface this part of the program is responsible for loading the next read as a stream. The buildup of a FASTQ file is thoroughly described in Subsection 2.3.2.

In this case, the code was also split up into functions for better readability:

1. Loading the QName, which is found on the first line for every read, behind a '@' character
2. Loading the sequence, which is stored in a text-based format in the FASTQ file, so it should be transformed to the SEQUENCE type. In a sequence, a base may be suddenly marked with an 'N' character, which means after the primary processing the process was unable to identify the base. Because of the way the sequencing machines work, this mostly happens at the end of the sequence. So, it was decided to only cut the sequence short at the moment an 'N' character is registered.

For example, if the sequence were *ACGGCGCATTACNNAN*, the interface will only store *ACGGCGCATTAC*. This is justified by the fact that we are statistically certain that this

sequence will be matched correctly from the moment the read is more than 15-20 bases long. The statistical proof itself will not be covered in this thesis.

3. Loading the qualities. Found on the fourth line for each read, the qualities for each base are stored. The information is stored in an array with the same length as the sequence.

SAM interface The SAM interface will write the current mapped read to a SAM file, one line for one read. The buildup of a SAM file is thoroughly described in Subsection 3.4.2.

For this implementation, some values of the SAM format are not important and should be set to a default value. So it came naturally to create an "init" function, in which these default values are assigned.

- RNext should be set to the '*' character;
- Pnext should be set to 0;
- TLen should be set to 0;

Then, to write the line in the output SAM file, a function was created which accepts an attribute of the *MAPPED_READ* format and writes it as a line in the file.

5.3.2 Memory management

5.3.2.1 malloc and sds_alloc

The allocation of memory in a C program is usually done using the "malloc" function. However, on the used SoC, a distribution of Linux is running. Like in most operating systems, this Linux distribution uses a virtual address space to enlarge the available RAM virtually.

The original idea for the project was to be able to accelerate certain functions using the programmable hardware available in the SoC chip. Since this hardware should be able to access the data in the memory, there two options:

1. First option: build a copy of the address translator on the hardware. This would require a lot of work and would slow the whole process of looking up things in memory.
2. Second option: let the software interface the physical memory directly, so that we don't use virtual addresses. This solution was used in the final implementation.

Luckily, Xilinx has published a library with an *sds_alloc* function. This does the same as the malloc function in the C language but in physical memory instead of the virtual memory.

5.3.2.2 Allocating memory

To allocate memory, the following formula was used:


```
memory_pointer = (TYPE*) sds_alloc( length * sizeof(TYPE) );
```

The `sds_alloc` function returns a pointer to the first address in memory that has been reserved. It turns out that this pointer should be cast explicitly to the type used.

- In case of reserving space for the sequence and the reference, the "TYPE" part in the formula should be filled in by the `BASE` type. For the length part of the formula, parameters were created named `seqMax` and `refMax` respectively.
- When space for the alignment matrix will be reserved, the "TYPE" part in the formula should be filled in by the `CELL` type (for more explanation on the `CELL`-type, see Subsection 5.3.3.1). As for the length of the space to reserve, this should be equal to `refMax` times `seqMax`.

5.3.3 The alignment

5.3.3.1 Parameters and types

A naive approach to implementing the Smith-Waterman algorithm might be to have the alignment matrix filled in by the values only. If we would use this approach, it would quickly become clear that the backtracking is impossible, since we also need to know where the value originates from. Also, when generating the CIGAR-string, it is good to know which direction it originates from, as it implies a match (M), insertion (I), or a deletion (D). Since all this information should be stored in the alignment matrix, it seemed a good idea to create a new type. This type was called *CELL*.

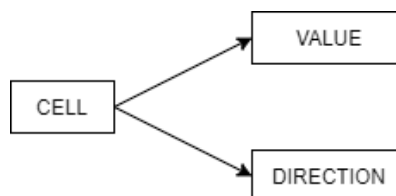


Figure 5.7: The type created to store the information for every cell in the alignment matrix

The `CELL` type stores the place it originates from, using a *DIRECTION* attribute. There are 4 possible directions a value can originate from: zero (coded as 0), diagonal (1), up (2), left (3). In case the value from 2 directions is equal, the following priorities have been followed:

1. If the maximum value is zero, the direction will always be (0).
2. If the maximum from the diagonal is equal to up or left, the diagonal will be chosen (1), which indicates a match.
3. If the values up and left are equal, the priority will go to the "up" direction (2).

Furthermore, it stores the value using a special type *CELL_VALUE*. The size of values this type can store should be greater than the length of the sequence times the similarity score. Keep in mind that to calculate the maximum, this value-type should be able to go negative.

5.3.3.2 Code structure

The alignment layer The alignment layer does everything in the alignment of the sequence that has nothing to do with the matrix fill in. That functionality is outsourced to a separate function.

Also, for the CIGAR string (which indicates the matches, insertions and deletions), a certain threshold was programmed in. The CIGAR string of a matched read consists of multiple parts. For example: *6M1I7M* consists of 3 parts; *6M*, *1I* and *7M*. By limiting the number of parts that are allowed, we can define a threshold by which a sequence is considered as "aligned". If both the forward and reverse sequences go over this limit, the sequence will be marked as "unmatched".

1. Fill in the matrix using the forward direction of the sequence.
2. Store the maximum value of the matrix.
3. Generate the CIGAR string for this forward direction. We have to do this so early because you need the full matrix for the CIGAR generation, and we want to reuse the memory for the matrix according to the reverse sequence.
4. Check if the CIGAR limit has been exceeded. If not, then store the position of the map.
5. Reverse the sequence.
6. Repeat for the reverse sequence.
7. Check for the limit of the CIGAR. If it was exceeded in both forward and reverse sequence, mark it as unmatched.
8. Check which read fit best (the forward or reverse one) by comparing the maximum values in the mapping. Assign the best one to the MAPPED.READ type.

The fill in layer This is probably the easiest layer of them all, but it will also be one of the most computationally demanding ones. It skims over every CELL in the matrix, except for the first row and first column, and uses the CELL generation layer to generate the cell on that specific spot. Also, while having an iteration that goes over every CELL, it is a good idea to use this iteration to find the maximum CELL in the matrix, which is the starting point for the backtracking.

The CELL generation layer This layer consists of 3 parts:

1. Generate the 3 values originating from diagonal, up, and left cells, using the formulas described by the S-W algorithm. For this step, the parameters s (for the similarity score) and gp (for the gap penalty) were created. For testing, These were mostly kept on $s = 3$ and $gp = 2$, but can be easily changed according to the application.
2. Determine which of these three values is the greatest. If all negative, choose 0.

3. Assign the correct value and direction of the newly generated cell.

This layer must be efficiently coded, and since it runs for every CELL (and there are a lot of CELLS), it can be expected to be the core of the program.

5.4 Implementation results

5.4.1 The IGV software

The free and easy to download program IGV[7] (or Integrative Genomics Viewer) is a visualization tool for exploring genomic datasets. In practice, it is used to examine the results from mapped reads.

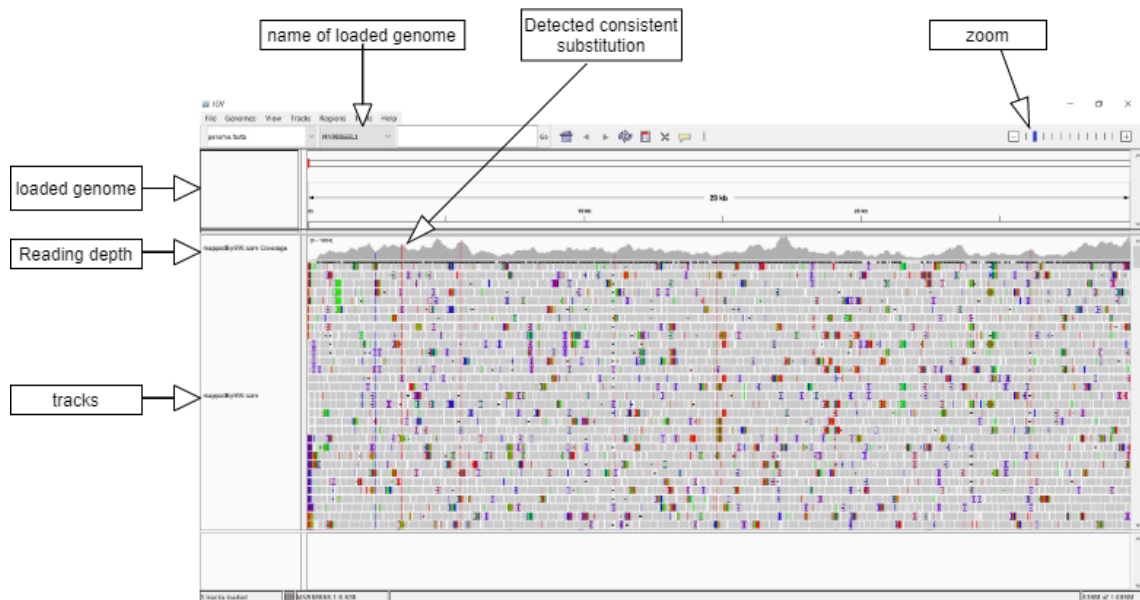


Figure 5.8: The layout of the IGV analysing software

The most important functions in IGV for this thesis are:

- It color-codes insertion, deletions or substitutions, so it is easy to see "from a distance";
- It can display the read depth for every base in the genome;
- If the read depth is sufficient, it can also determine if an indel or substitution is part of the read, or is consistent enough throughout all the reads to show it in the full genome;
- The sequence direction is indicated by an arrow when examining a specific read.

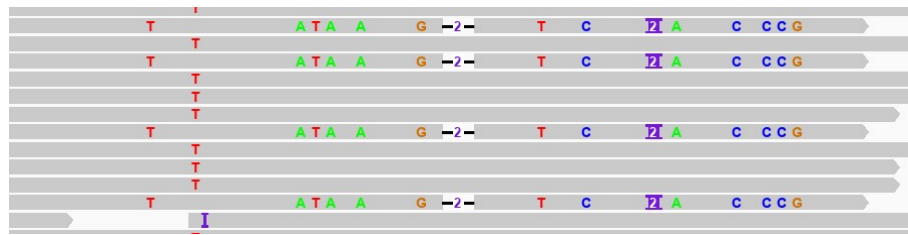


Figure 5.9: Close-up of a few reads in the IGV analysing software. Notice how the color coded bases are substitutions, whereas the matched bases are not shown. Moreover, a gap and insertion is also visible

5.4.2 Sample data

To try the implementation on the human genome directly seemed way too ambitious. Therefore, it seemed appropriate to try the implementation first on an organism that has a genome of only a few thousand bases in length. Since this thesis is performed around the spring of 2020, the "coronavirus" (or SARS-CoV-2) seemed like an appropriate candidate. After a quick google search, it turned out that this virus had a genome consisting of merely 29 thousand bases.

The genome of the coronavirus was found easily online with reference number MN988668.1 in the genome bank of NCBI [5].

The FASTQ files obtained from a DNA sequence for the coronavirus used in this thesis were found in the SRA (sequence read archive) of the NCBI, submitted by the University of Washington.[5]

Of course, when fitting these fastQ files through the implementation from this thesis, we have to be able to compare it with an implementation which is considered as correct. Therefore, an online tool (Galaxy) was used where a lot of existing bioinformatical algorithms and programs can be performed on given sample data using cloud computing [9]. By converting the fastQ files (given by the SRA of the NCBI) using one of these available applications, we can get a pretty good reference for matching against our results. As the application to compare to, the *Bowtie* algorithm was chosen, which is based on the S-W algorithm.

5.4.3 Results

We will compare the results of our implementation with the sample data, using the Galaxy online tool, where the *Bowtie* algorithm was used for the genome mapping.

After running the software implementation overnight with the unmapped sequences from the coronavirus, we obtained a dataset of mapped reads. Both of these mapped reads and the dataset obtained by using the Galaxy online tool, were imported into IGV for comparison, which can be seen in figure 5.10

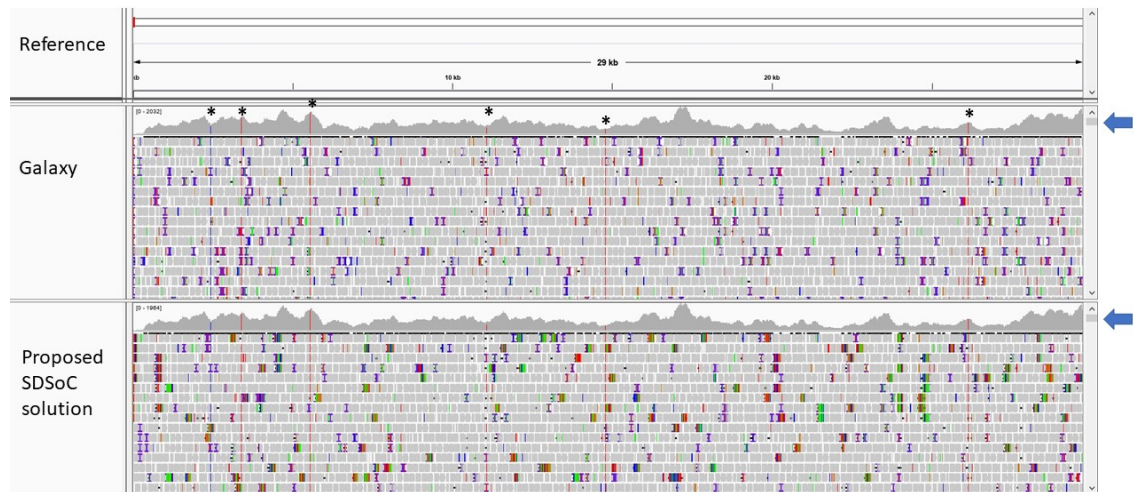


Figure 5.10: A comparison of the data obtained by Galaxy (top) and the data obtained by the implementation discussed in this chapter (bottom)

It can be observed that the implementation is working correctly since the reading depth graphs are approximately the same (indicated by an arrow). Furthermore, IGV was able to detect and identify consistent substitutions or indels in the reads.

If you look closely to figure 5.10, some small differences between the exact reads are visible. This will probably be because both the algorithm and the parameters were a bit different, such as gap penalty and similarity score. However, the important part is that the reading depths are the same, as well as the consistently mutated bases marked in the genome (indicated by an asterisk).

The implications for the coronavirus data The coronavirus is constantly changing because of the mutations of some bases in the genome over time, just like the flu. In this way, it is possible to trace where infections originate. For example, the corona in China can have slightly different bases than from Italy or Spain. We can identify these changes by sequencing the whole genome and looking for consistent substitutions or indels over the whole dataset.

From these results shown by IGV 5.10, some mutations are shown, so we can assume that the coronavirus which was sequenced in Washington (taken from a US patient, the FASTQ file used) has some slight differences with the original Chinese sequenced genome (the reference sequence).

Chapter 6

Accelerating the software implementation using HLS

6.1 Analysing software performance

In the SDSoC environment, it is possible to analyze the software running on the SoC chip using a TCF profiler. After running this analysis, the TCF profiler returns an overview of the functions, sorted on the amount of time spent when running. The analysis is shown in figure 6.1

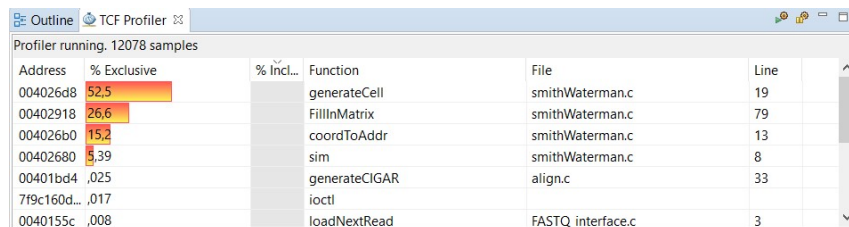


Figure 6.1: a TCF profile of the software implementation

When examining this analysis, we should keep in mind that the generateCell, coordToAddr, and sim functions are inline functions used in the FillInMatrix method. Just as suspected the software spends almost all of its time in these methods, so it's worth trying to accelerate these functions.

6.2 Recoding parts of the software to be more hardware friendly

6.2.1 Recoding the Cell generation layer

In 2011, Vermij E. (Delft University of Technology, The Netherlands) studied RVE (recursive variable expansion) [34]. He discusses the most efficient ways to program a processing element to generate one value in the alignment matrix. His results can be found in figure 6.2.

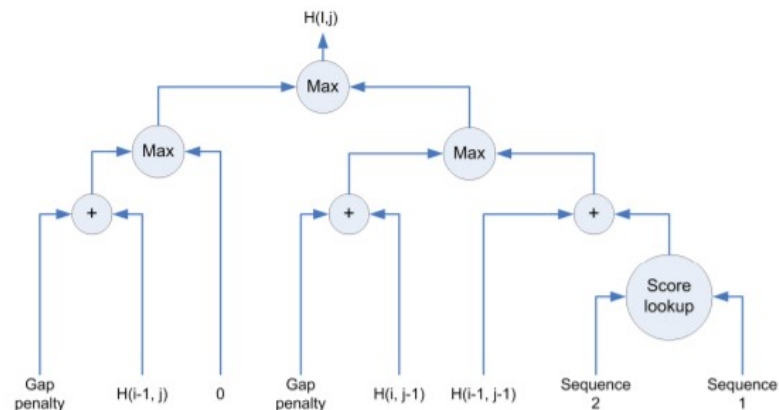


Figure 6.2: The optimal processing found by Vermij E. [34]

It seemed like a good idea to reimplement the generatecell function, using this newly found scheme. However, it is also important to keep track of where the value comes from. Therefore, the following (new) code was adopted for generating a CELL:

```
//calculate the possible values
CELL diagonalCELL = { diagonal.value + sim(refVal, seqVal), 1 };
CELL leftCELL = { left.value - gp, 2 };
CELL upCELL = { up.value - gp, 3 };
CELL zeroCELL = { 0, 0 };

CELL upstreamA = (leftCELL.value > upCELL.value) ? leftCELL : upCELL;
CELL upstreamB = (diagonalCELL.value > zeroCELL.value) ?
diagonalCELL : zeroCELL;

CELL newCell = (upstreamA.value > upstreamB.value) ? upstreamA : upstreamB;

//Return the cell:
return newCell;
```

Where the second attribute in the CELL type is the direction.

6.2.2 Recoding the FillIn layer

HLS does not support input and output from the same memory locations in hardware. Therefore, the FillIn layer also has to be recoded. We will take a look at the data dependencies again:

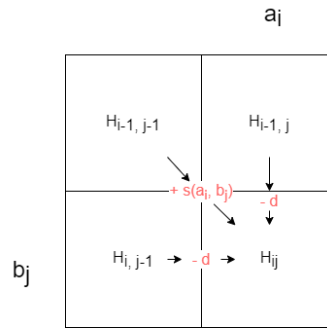


Figure 6.3: Data dependencies for generating a cell in the alignment matrix

Since the current cell only depends on the left-up 3 cells, we can compute every cell on the diagonal in parallel. Therefore, 3 arrays were created: one contains the current diagonal being generated, one contains the previous diagonal for the cell above and the cell to the left of the current generated cell, and one for the diagonal before that. This last array will house the left-up cell also needed for the new cell generation. A schematic of the data structures used can be found in table 6.1

	ref	T	G	T	T	A	C	G	G		ppD	pD	cD
seq	0	0	0	0	0	0	0	0	0	=>	0	0	0
G	0	0	3	1	0	0	?				0	0	?
G	0	0	3	1	0	?					1	0	?
T	0	3	1	6	?						1	6	?
T	0	3	1	?							3	1	?
G	0	1	?								0	1	?

Table 6.1 The 3 newly created arrays to house the data needed for generating the next diagonal of cells. The '?' in the cD array (red) represents cells currently being generated. Notice that all the information needed to generate the new cell is present in the other 2 arrays

Also, keep note that the FillIn layer should keep track of the maximum location in the matrix. Having gained this new information, we can recode the FillIn layer as follows:

1. Create the arrays *ppD*, *pD* and *cD*. They should be the length of the sequence. All are initialized on zeros.
2. Start the current diagonal at the second column.
3. For every cell at the diagonal, do the following:
 - (a) Check for edge cases. The row has to be smaller than the length of the sequence, the column can't be 0 or smaller and the column can't be bigger than the length of the reference.
 - (b) Generate the new cell using the data in the *ppD* and *pD* arrays.
 - (c) Write this new cell to memory, and its value to the *cD* array.

- (d) Check if it is bigger than the current maximum. If so, the current maximum should be this new cell.
4. Set current diagonal to the next one.
5. Shift the values ($cD \rightarrow pD$ and $pD \rightarrow ppD$) and repeat from step 3, until the total matrix is filled.
6. Return the position of the maximum to the alignment level.

Note that by programming the Fill in Layer this way, there is no need to read cells from memory, since they can be generated using the arrays. Only the reference and read sequences should be read from memory.

6.3 Hardware acceleration

When accelerating the application, it was decided to implement the FillIn and cell generation layer in hardware, since they are the functions where most computational time is spent (see Section 6.1). Therefore, they were merged into 1 convenient function, which will be transferred to the programmable hardware.

6.3.1 DMA

DMA or Direct Memory Access allows certain hardware systems in the computer to access the main system memory independent of the CPU. This can be of importance to the performance since the hardware system requires a lot of data movement between to and from the memory. The arrays (ref, seq, and matrix) are available in the memory, which means the programmable hardware should be able to access these arrays using DMA.

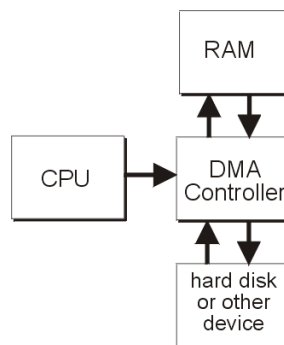


Figure 6.4: Schematic of a system using DMA

The following pragmas were used for the DMA:

```
#pragma SDS data access_pattern(ref:SEQUENTIAL, seq:SEQUENTIAL,
    matrix:SEQUENTIAL)
#pragma SDS data sys_port(ref:AFI, seq:AFI, matrix:AFI)
#pragma SDS data mem_attribute(ref:PHYSICAL_CONTIGUOUS,
    seq:PHYSICAL_CONTIGUOUS, matrix:PHYSICAL_CONTIGUOUS)
#pragma SDS data zero_copy(ref[0:refMax], seq[0:seqMax],
    matrix[0:refMax*seqMax])
```

- **access_pattern** can be either SEQUENTIAL or RANDOM. It specifies the data access pattern by the hardware to the memory. If SEQUENTIAL is used, the interface will be a stream (e.g. ap_fifo). On the other hand, when RANDOM is used, a RAM interface will be generated.
- **sys_port** can be ACP, AFI, GP or MIG. It specifies on which level the memory is interfaces.
 - ACP: Allows hardware functions to have cache-coherent access to DDR using the PS L2 cache.
 - AFI: Hardware functions have fast non-cache coherent access to DDR via the PS memory controller.
 - GP: The processor directly writes/reads data to/from hardware function. This would be inefficient for large data transfers.
 - MIG: Hardware functions access DDR from PL via a MIG IP memory controller.

We have quite a large amount of data, so GP doesn't seem like a good option. Likewise, the memory will be accessed uniform, we have no use for a cache as this would just slow it down. Therefore the AFI was selected.

- **mem_attribute** can be either PHYSICAL_CONTIGUOUS or NON_PHYSICAL_CONTIGUOUS. The default value is NON_PHYSICAL_CONTIGUOUS. PHYSICAL_CONTIGUOUS is used for memory allocated with sds_alloc. Likewise, NON_PHYSICAL_CONTIGUOUS is used with memory allocated using malloc.
- **zero_copy** means that the hardware function accesses the data directly from shared memory through an AXI master bus interface.

The data_pack pragma The data_pack pragma is used on the matrix since it is an array of structs (CELL type). This packs the data fields of a struct into a single scalar. The bitwidth of the packed struct is the sum of its attributes.

The bitwidth of (packed) data on the AXI bus must be a power of 2, to interface with the memory. Therefore, it was chosen to change the CELL_VALUE type to an int16_t and the direction to uint16_t. This is however quite wasteful towards memory usage. It can be optimized by remodeling some of the code, but due to time constraints, this was not implemented.

6.3.2 Unrolling pragmas

Which loops are unrolled and how, is best described in pseudocode, which is found below:

```

1 Create the current maximum (=0) and the position of the maximum;
2 Create the 3 arrays (pD, ppD, and cD) of size seqMax;
3
4 for i from 0 to seqMax: //initializing the arrays
5     #pragma HLS UNROLL
6     pD[i] = ppD[i] = cD[i] = 0;
7
8 for currentColumn from 2 to (refLength + seqLength): //The fillIn loop
9     #pragma HLS loop_tripcount min=5000 avg=5150 max=5700
10    #pragma HLS PIPELINE
11
12    for i from 0 to seqMax: //the diagonal loop
13        #pragma HLS UNROLL
14        row = i;
15        col = currentColumn - i;
16
17        //edge cases
18        if (i < seqLength && col != 0 && col <= refLength):
19            generateCell( //cell generation layer
20                diagonal value = ppD[i-1],
21                left value = pD[i],
22                up value = pD[i-1],
23                reference value = ref[col],
24                sequence value = seq[row]
25            );
26            store the current cell at cD[i];
27            store the current cell in memory;
28            if (value of cell bigger than maximum):
29                replace current max position with this cell;
30
31
32    for i from 0 to seqMax: //the shifting loop
33        #pragma HLS UNROLL
34        ppD[i] = pD[i];
35        pD[i] = cD[i];
36
37 return the position of the maximum;

```

The unroll pragma transforms the loop to multiples copies of the loop body in the FPGA hardware, which allows the loop iterations to be executed in parallel. Since no parameters are given to this pragma, the loops will be unrolled fully, so the whole loop will be executed in parallel. Note that the number of iterations of the loop should be known at compile time.



In the code, all loops are unrolled fully, except for the loop that runs over every diagonal (the fill-in loop). Unrolling this loop would not affect execution speed since all elements in the loop are dependent on the previous iteration of this loop.



The `loop_tripcount` pragma is applied to a loop to manually specify the total number of iterations performed by a loop. In our code, this is applied to the fill-in loop since the number of iterations is not known at compile time (`refLength` and `seqLength` are variables).

The `pipeline` pragma will try to transform the body of the loop to a pipeline, where every iteration of the loop has a given `II` or *Initialization Interval*. If no `II` is given, the default is 1. This means it will try to run one iteration of the loop body for every clock cycle.

6.3.3 Comparison with the software

If we examine execution time (using the built-in latency analyzers in SDSoC), we can see we achieved a speedup of 4.41. This means the hardware variant of the implementation runs 4.41 times faster than the software variant.

Performance estimates for 'FillInHW in align.c:250' funct ...			
SW-only (Measured cycles)		94414155	
Hardware accelerated (Estimated cycles)		21410374	
Estimated speedup		4,41	

Performance estimates for 'FillInHW in align.c:272' funct ...			
SW-only (Measured cycles)		94414155	
Hardware accelerated (Estimated cycles)		21410374	
Estimated speedup		4,41	



Resource utilization estimates for Hardware functions			
Resource	Used	Total	% Utilization
DSP	0	1728	0
BRAM	4	312	1,28
LUT	125724	230400	 54,57
FF	53266	460800	 11,56

Figure 6.5: Latency analyzer in SDSoC. We can see the number of clock cycles spent in the software and the hardware variant of the implementation, as well as the speedup. The speedup is calculated 2 times, since we run the `fillIn` function twice, once for the forward and once for the reverse sequence. The resource utilization of the FPGA hardware is also visible in this picture

Chapter 7

Conclusion and future research

Problem definition The best way to analyse DNA in detail is to sequence it. This will determine which bases and in which order they are occurring in the DNA. However, the most commonly used technology of DNA sequencing machines limit us to reads of 75-300 bases long. If we want to make assumptions about the whole genome, we will need to input a lot of these reads.

Typically, each read is compared with the whole genome in a local alignment, for example with the Smith-Waterman algorithm. As an output, we would get the position in the human genome and an alignment with its score (how well the sequence fits in that spot). This practice is commonly referred to as *Mapping to a reference genome*.

Sequencing machines produce millions of the reads in a sequencing experiment. Since every read from DNA sequencing machines are 75 to 300 bases long, and the whole human genome is approximately 3 billion bases, this mapping is computationally a very intensive task. If we analyze the S-W algorithm (as we have done in Subsection 3.3.2), we can see that the value of each cell in the matrix is only dependent on the left-upmost 3 cells. Therefore, it leads us to believe that this algorithm can be accelerated on hardware solutions such as an FPGA (as discussed in Chapter 4) since the S-W algorithm is heavily parallelizable.

In most clinical applications where mapping to a human reference genome is used, the number of reads to be compared with the genome is in the millions, which further increases the demand to speedup the process of mapping the reads in the genome, to decrease the time-to-result.

The idea of this thesis was to implement the Smith-Waterman alignment algorithm on an MPSoC, which contains both programmable FPGA hardware and an ARM processor in 1 chip. As a target board, the ZCU 104 evaluation kit was chosen.

Proposed solution and implementation As a starting point, a software implementation was implemented on the ARM processor. After running the software implementation overnight with the unmapped sequences from the coronavirus as a sample set, we obtained a dataset of mapped reads. The reads were also mapped by using the Galaxy online tool [9] and both were imported into the IGV analyzing software for visualisation and comparison, which can be seen in figure 7.1

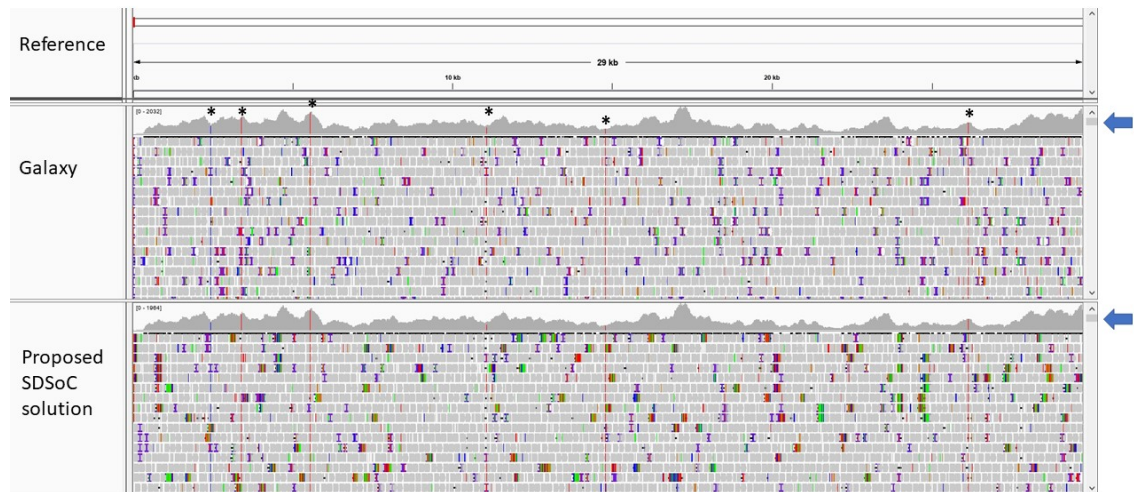


Figure 7.1: A comparison of the data obtained by Galaxy (top) and the data obtained by the implementation discussed in this chapter (bottom)

It can be observed that the implementation is working correctly since the reading depth graphs are approximately the same (indicated by an arrow). If we look closely to figure 7.1, some small differences between the exact reads are visible. This will probably be because both the algorithm and the parameters were a bit different. However, the important part is that the reading depths are the same, as well as the consistently mutated bases marked in the genome (indicated by an asterisk).

Hardware speedup After implementing the matrix fillIn in the FPGA hardware, we can examine execution time (using the built-in latency analyzers in SDSoC). The estimated achieved speedup was 4.41x. This means the hardware variant of the implementation runs 4.41 times faster than the software variant.

Performance estimates for 'FillInHW in align.c:250' funct ...			
SW-only (Measured cycles)			94414155
Hardware accelerated (Estimated cycles)			21410374
Estimated speedup			4,41

Performance estimates for 'FillInHW in align.c:272' funct ...			
SW-only (Measured cycles)			94414155
Hardware accelerated (Estimated cycles)			21410374
Estimated speedup			4,41

Resource utilization estimates for Hardware functions			
Resource	Used	Total	% Utilization
DSP	0	1728	0
BRAM	4	312	1,28
LUT	125724	230400	54,57
FF	53266	460800	11,56

Figure 7.2: The latency analyzer in SDSoC. We can see the number of clock cycles spent in the software and the hardware variant of the implementation, as well as the speedup. The speedup is calculated 2 times, since we run the fillIn function twice, once for the forward and once for the reverse sequence. The resource utilization of the FPGA hardware is also visible in this picture

Future work - recommendations for improving the current implementation the following optimizations could improve the proposed implementation:

- Parallelize the algorithm even more. This can be achieved in the following ways:
 - Map the forward and the reverse sequence in parallel. However, this will require double the memory space (for the fill-in matrix) and double the resources of the programmable hardware. In the current implementation this was not implemented since the programmable hardware is already utilized for more than 50%.
 - Parallelize the mapping of multiple sequences at the same time. Each sequence is completely independent of the other sequences, so their mapping could also be executed concurrently. This was not implemented in the current implementation, for the same reasons.
- Treat the alignment gap opening and extension separately. In the current implementation, the gap penalty is treated linearly. However, the Smith-Waterman algorithm could be refined to include an affine gap penalty. More info on the gap penalty can be found at Subsection 3.3.2.
- In a sequence, a base position may be suddenly marked with an 'N' character, which represents a base that was unidentified in the primary processing. In this thesis, it was decided to cut the sequence short at the moment an 'N' character is registered. However, a possible improvement might be to keep these unidentified bases in the alignment as a "generic" base and in this way improve the accuracy of the alignment. More info on the current implementation can be found at Subsection 5.3.1.2.

- Make some types more memory efficient, such as the BASE and the SEQ_INDEX type. Firstly, the BASE type only has 4 possible values (the 4 nucleotide bases). However, it is defined as 1 byte in the current implementation, since it is the smallest type that C supports. However, storing a base in 1 byte is memory inefficient. Improvements could be made by defining an own type, by stacking 4 bases in 1 byte. Secondly, the SEQ_INDEX type is stored as 2 bytes in the current implementation, even though it only contains values up to 300, which makes it memory inefficient. This could also be improved.

Some of these types could be fixed using the `std_logic_vector` from VHDL, for example for the BASE and DIR type. However, these types are also used in software-implemented parts of the application, so a separate type would be needed for the hardware and software part.

- If this implementation would be used in practice, there would be a need to find an easier way to on and offload data to board. Currently, this is accomplished by changing out the SD card, which could easily be improved using FTP since an Ethernet stack is, which are already implemented by the operating system. The speed at which the data is transferred, is not important since we can assume that the time it takes to map the sequence takes much longer.

Future work - recommendations for re-evaluating the alignment method In the field of alignment algorithms, a few alternatives exist to S-W since it takes a lot of computing power. In most cases, these algorithms will use a transformation or a lookup table to determine possible candidate locations, where a small regional S-W will be performed on a part of the genome. For example, The BFAST algorithm uses a hash table where candidate locations are stored for every sequence [27]. As a second example, in the Bowtie application, the candidate locations are determined using the Burrows-Wheeler Transform.

As future work, one of these more sophisticated algorithms can be selected. Some candidate locations can then be determined in a way defined by the algorithm. These candidate locations can then be mapped using the implementation of this thesis.

Bibliography

- [1] (April 2020). Bioinformatics company. <http://www.clccell.com/download.html>.
- [2] (April 2020). Bioinformatics company. <https://www.cray.com/>.
- [3] (April 2020). Bioinformatics company. <http://www.timelogic.com/>.
- [4] (April 2020b). FPGA diagram. https://evergreen.loyola.edu/dhhoe/www/FPGA_diagram.png.
- [5] (April 2020). Genome of coronavirus, NCBI. <https://www.ncbi.nlm.nih.gov/sra/SRX7852918>.
- [6] (April 2020). Human karyotype. https://en.wikipedia.org/wiki/Human_genome.
- [7] (April 2020). IGV download page. <http://software.broadinstitute.org/software/igv/>.
- [8] (April 2020). Nucleotide chemical structure (modified). <https://en.wikipedia.org/wiki/Nucleotide>.
- [9] (April 2020). Online bioinformatics tools. <https://usegalaxy.org/>.
- [10] (April 2020). Sequential working of microprocessor. <http://www.lighterra.com/>.
- [11] (March 2020a). Azure machine learning FPGA comparison. <https://docs.microsoft.com/en-us/azure>.
- [12] (March 2020a). DNA chemical structure.
- [13] (March 2020). *Sequence Alignment/Map Format Specification*. The SAM/BAM Format Specification Working Group.
- [14] (May 2020b). Double stranded DNA with coloured bases. <https://en.wikipedia.org/wiki/DNA>.
- [15] (May 2020). Down syndrome. https://en.wikipedia.org/wiki/Down_syndrome.
- [16] Alexandrov, V. (2006). General purpose computations on graphics hardware: methods, algorithms and applications. In *Computational science - ICCS 2006 : 6th international conference, Reading, UK, May 28-31, 2006 ; proceedings*. Berlin: Springer.
- [17] Benkrid, K. (2012). High performance biological pairwise sequence alignment: FPGA versus GPU versus Cell BE versus GPP. *International Journal of Reconfigurable Computing*, 2012:Article ID 752910.

- [18] Dahm, R. (2008). Discovering DNA: Friedrich Miescher and the early years of nucleic acid research. *Human Genetics*, 122:565–81.
- [19] Drury, S., Hill, M., and Chitty, L. (2016). *Cell-Free Fetal DNA Testing for Prenatal Diagnosis*. Elsevier, Inc.
- [20] James Arram, e. a. (2015). Leveraging FPGAs for accelerating short read alignment.
- [21] Keirsbilck, J. V. and Luccofier, A. (2015). Evolutie van de prenatale screening naar aneuploidie in het eerste trimester: van leeftijd tot nipt (dutch). *azlink*, 3:8–9.
- [22] Mardis, E. R. (2006). Anticipating the 1,000 dollar genome. *Genome Biology*, 7:112.
- [23] Martínek, T. (2020). Xilinx vivado HLS.
- [24] Nathaniel McVicar, e. a. (2012). FPGA acceleration of short read alignment.
- [25] Needleman, S. B. and Wunsch, C. D. (1970). A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48:443–53.
- [26] Nollet, F. (2019). powerpoint: Howest cursus 'nieuwste ontwikkelingen in de biotechnologie' 2019-2020 hoofdstuk sanger and NGS (dutch).
- [27] Olson, C. B. (2011). An FPGA acceleration of short read human genome mapping. Master's thesis, University of Washington.
- [28] Pauling, L. and Corey, R. (1953). A proposed structure for the nucleic acids. *Proceedings of the National Academy of Sciences of the United States of America*, 39:84–97.
- [29] Reid, J. and Ross, J. (2011). Mendel's genes: towards a full molecular characterization. *Genetics*, 189:3–10.
- [30] Scheinin, I. (2014). DNA copy number analysis of fresh and formalin-fixed specimens by shallow whole-genome sequencing with identification and exclusion of problematic regions in the genome assembly. *Genome Res.*, 24.
- [31] Smith, T. F. and Waterman, M. S. (1981). Identification of common molecular subsequences. *Journal of Molecular Biology*, 147:195–197.
- [32] Storaasli, O. and Strenski, D. (2011). Accelerating genome sequencing 100x with FPGAs. <http://ft.ornl.gov/olaf/pubs/RSSIOlafDave.pdf>.
- [33] Vergel, R. (2018). *Getting started with SDSoC*. Xilinx Inc. <https://github.com/Xilinx/SDSoC-Tutorials/tree/master/getting-started-tutorial>.
- [34] Vermij, E. (2011). Genetic sequence alignment on a supercomputing platform. Master's thesis, Delft university of technology.

Appendix A

Implementation code

The implementation can also be found on Github: <https://github.com/robinno/refGenMapping>

A.1 Filesystem organization

To keep the code manageable, it was split up in multiple files. These files were organized in the directory structure as seen in figure A.1

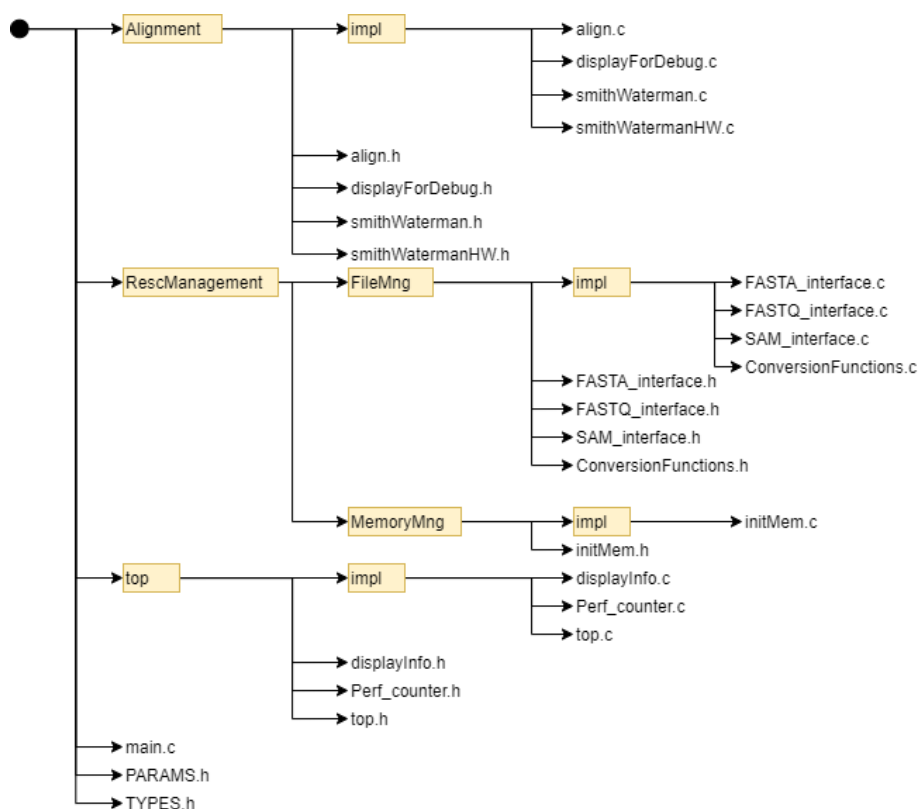


Figure A.1: The used directory structure in the implementation. Directories are colored in yellow.

A.2 Code

PARAMS.h

```

1 #ifndef PARAMS_H
2 #define PARAMS_H
3
4 //to make things work, defining NULL here
5 #define NULL ((void *)0)
6
7 //FILE PATHS
8 #define fastQPath "/mnt/reads.fastq"
9 #define samPathSW "/mnt/mappedSW.sam"
10 #define samPathHW "/mnt/mappedHW.sam"
11 #define fastaPath "/mnt/genome.fasta"
12
13 //PARAMETERS FOR THE ALIGNMENT
14 #define gp 2 //gap penalty
15 #define s 3 //similarity score
16 #define maxCigarParts 15 //the maximum amount of parts in the CIGAR (otherwise sequence mark
17
18 //BUFFER SIZES
19 #define refMax 5700 //maximum length of the reference genome (BUFFER SIZE)
20 #define seqMax 160 //maximum length of the sequence to match (BUFFER SIZE)
21 #define buffSize 64 //for character buffers (strings), such as qname, rname, cigar, ...
22
23 //define the bases:
24 #define A 0b00
25 #define C 0b01
26 #define G 0b10
27 #define T 0b11
28
29 #endif

```

TYPES.h

```

1 #ifndef TYPES_H
2 #define TYPES_H
3
4 #include <stdint.h>
5
6 #include "PARAMS.h"
7
8 typedef uint8_t BASE; //data type for base (A, C, G or T)
9 typedef uint16_t REF_INDEX; //data type for index of the reference
10 typedef uint16_t SEQ_INDEX; //data type for index of the reference
11 typedef uint32_t MATRIX_INDEX; //indexing the alignment matrix => ref_index size * seq_index
12
13 //SEQ and REF array types
14 typedef struct SEQ SEQ;
15 struct SEQ {
16     BASE* el;
17     SEQ_INDEX length;
18 };
19
20 typedef struct REF REF;
21 struct REF {
22     BASE* el;
23     REF_INDEX length;

```

```

24 };
25
26 //FILE_IO types:
27 typedef struct GENOME GENOME;
28 struct GENOME {
29     char Rname[buffSize];
30     REF ref;
31 };
32
33 typedef struct READ READ;
34 struct READ {
35     char Qname[buffSize];
36     SEQ seq;
37     char qualities[seqMax];
38 };
39
40 typedef struct MAPPED_READ MAPPED_READ;
41 struct MAPPED_READ {
42     //info from FASTQ line:
43     READ read;
44
45     //info from FASTA line
46     char Rname[buffSize];
47
48     //mapping results
49     uint16_t Flag;
50     REF_INDEX Pos;
51     uint8_t MapQ;
52     char CIGAR[buffSize];
53
54     //defaults:
55     char Rnext[1];      // = {'*'};
56     REF_INDEX Pnext;    // = 0;
57     REF_INDEX Tlen;     // = 0;
58 };
59
60 //TYPES USED WITHIN ALIGNMENT
61 typedef struct POS POS; //data type to keep track of position in alignment matrix
62 struct POS {
63     SEQ_INDEX row;
64     REF_INDEX col;
65 };
66
67
68 typedef uint16_t DIRECTION;
69 typedef int16_t CELL_VALUE; //data type for values during the alignment
70
71 typedef struct CELL CELL; //data type of a specific cell in the alignment matrix
72 struct CELL {
73     CELL_VALUE value;
74     DIRECTION d;
75 };
76
77 #endif

```

main.c

```

1 #include "top/top.h"
2

```

```

3 int main(int argc, char* argv[]) {
4     return top();
5 }

```

top.h

```

1 #ifndef TOP_H
2 #define TOP_H
3
4 #include "sds_lib.h" //for memory free functions
5 #include <stdio.h>    //for the FILE type
6 #include <inttypes.h> //to print weird types, like 64 bit integers
7
8 #include "../PARAMS.h" //including the parameters
9 #include "../TYPES.h"  //including the types
10
11 #include "../RescManagement/FileMng/FASTA_interface.h"
12 #include "../RescManagement/FileMng/FASTQ_interface.h"
13 #include "../RescManagement/FileMng/SAM_interface.h"
14 #include "../RescManagement/MemoryMng/initMem.h" //to initialize memory
15
16 #include "../Alignment/smithWaterman.h" //for the alignment
17 #include "../Alignment/align.h"
18
19
20 #include "displayInfo.h" //for feedback to the user via command line
21 #include "Perf_counter.h"
22
23 int top();
24
25 #endif

```

top.c

```

1 #include "../top.h"
2
3
4 int top() {
5     READ read;
6     GENOME genome;
7     MAPPED_READ mapped_read;
8
9     //RESERVE MEMORY
10    BASE *addrSpaceReverseSeq = 0;
11    initSeq(&(read.seq.el), &addrSpaceReverseSeq);
12    initRef(&(genome.ref.el));
13    CELL *addrSpaceMatrix = 0;
14    // #pragma HLS data_pack variable=addrSpaceMatrix struct_level //needed for hardware accele
15    initAlignMatrixAddrSpace(&addrSpaceMatrix);
16
17    //INIT THE SW MATRIX:
18    initMatrix(refMax, seqMax, addrSpaceMatrix);
19
20    //LOAD THE REF
21    loadGenome(fastaPath, &genome);
22    displayGenomeInfo(genome);
23
24    //OPEN THE FILES:
25    FILE* fastQfile = fopen(fastQPath, "r");
26    FILE* samFileSW = fopen(samPathSW, "w+");

```

```

27 FILE* samFileHW = fopen(samPathHW, "w+");
28
29 if (fastQfile == NULL || samFileSW == NULL || samFileHW == NULL) {
30     printf("ERROR: Files didn't open correctly");
31     fclose(fastQfile);
32     fclose(samFileSW);
33     fclose(samFileHW);
34     sds_free(read.seq.el);
35     sds_free(genome.ref.el);
36     sds_free(addrSpaceMatrix);
37
38     return -1;
39 }
40
41 ///////////////////////////////////////////////////
42
43 int counter = 1;
44 READ revRead;
45
46 //performance counters:
47 perf_counter sw_ctr, hw_ctr;
48 PCreset(&sw_ctr);
49 PCreset(&hw_ctr);
50
51 int allReadsDone = 0;
52 do {
53     allReadsDone = loadNextRead(fastQfile, &read);
54     //displayCurrReadInfo(read); //debug
55
56     //SOFTWARE
57     PCstart(&sw_ctr);
58     align(genome, &read, &mapped_read, addrSpaceMatrix, addrSpaceReverseSeq,
59         &revRead);
60     PCstop(&sw_ctr);
61
62     //displayCurrMappedReadInfo(mapped_read); //debug
63     // printf("%i: in software => sequence: %s\t => flag: %i\n", counter, read.Qname,
64     //     mapped_read.Flag);
65     writeSamLine(samFileSW, mapped_read);
66
67     //HARDWARE
68     PCstart(&hw_ctr);
69     alignHW(genome, &read, &mapped_read, addrSpaceMatrix,
70         addrSpaceReverseSeq, &revRead);
71     PCstop(&hw_ctr);
72
73     //displayCurrMappedReadInfo(mapped_read); //debug
74     // printf("%i: in hardware => sequence: %s\t => flag: %i\n", counter, read.Qname,
75     //     mapped_read.Flag);
76     writeSamLine(samFileHW, mapped_read);
77
78     printf("%i: sequence: %s\t => flag: %i\n", counter, read.Qname, mapped_read.Flag);
79     counter++;
80     printf("current clock: %PRIu64\n", (uint64_t) sds_clock_counter());
81 } while (allReadsDone != 255);
82
83 unsigned long long sw_cycles = avg_cpu_cycles(sw_ctr);
84 unsigned long long hw_cycles = avg_cpu_cycles(hw_ctr);

```

```

85     double speedup = (double) sw_cycles / (double) hw_cycles;
86
87     printf("Average number of CPU cycles running in software: %llu\n", sw_cycles);
88     printf("Average number of CPU cycles running in hardware: %llu\n", hw_cycles);
89     printf("speedup: %f\n", speedup);
90
91     //////////////////////////////////////
92
93     //CLOSE THE FILES
94     fclose(fastQfile);
95     fclose(samFileSW);
96     fclose(samFileHW);
97     printf("files saved\n");
98
99     //FREE THE MEMORY
100    sds_free(read.seq.el);
101    sds_free(genome.ref.el);
102    sds_free(addrSpaceMatrix);
103
104    return 0;
105 }

```

displayInfo.h

```

1  #ifndef DISPLAYINFO_H
2  #define DISPLAYINFO_H
3
4  #include <stdio.h>
5  #include "../TYPES.h"
6
7  void displayGenomeInfo(GENOME genome);
8  void displayCurrReadInfo(READ read);
9  void displayCurrMappedReadInfo(MAPPED_READ mapped_read);
10
11 #endif

```

displayInfo.c

```

1  #include "../displayInfo.h"
2
3  void displayGenomeInfo(GENOME genome) {
4      printf("the reference genome:\n");
5      printf("Rname: %s\n", genome.Rname);
6      printf("length of reference: %i\n", genome.ref.length);
7      printf("\n");
8  }
9
10 void displayCurrReadInfo(READ read) {
11     printf("The current sequence:\n");
12     printf("Qname: %s\n", read.Qname);
13     printf("Sequence: \t");
14     for (SEQ_INDEX i = 0; i < read.seq.length; i++) {
15         printf("%i", read.seq.el[i]);
16     }
17     printf("\n");
18     printf("Length of sequence: %i\n", (int) read.seq.length);
19     printf("Qualities: \t");
20     for (SEQ_INDEX i = 0; i < read.seq.length; i++) {
21         printf("%c", read.qualities[i]);
22     }

```



```

23     printf("\n\n");
24 }
25
26 void displayCurrMappedReadInfo(MAPPED_READ mapped_read) {
27     printf("The current sequence (mapped):\n");
28     printf("Qname: %s\n", mapped_read.read.Qname);
29     printf("Sequence: \t");
30     for (SEQ_INDEX i = 0; i < mapped_read.read.seq.length; i++) {
31         printf("%i", mapped_read.read.seq.el[i]);
32     }
33     printf("\n");
34     printf("Length of sequence: %i\n", (int) mapped_read.read.seq.length);
35     printf("Qualities: \t");
36     for (SEQ_INDEX i = 0; i < mapped_read.read.seq.length; i++) {
37         printf("%c", mapped_read.read.qualities[i]);
38     }
39     printf("\n");
40     printf("Flag: %i\n", mapped_read.Flag);
41     printf("Position found: %i\n", mapped_read.Pos);
42     printf("Mapping quality: %i\n", mapped_read.MapQ);
43     printf("CIGAR string: %s\n", mapped_read.CIGAR);
44
45     printf("\n\n");
46 }

```

Perf.counter.h

```

1 #include "stdlib.h"
2
3 //the performance counter
4 typedef struct perf_counter perf_counter;
5 struct perf_counter{
6     unsigned long long tot, cnt, calls;
7 };
8
9 //resets the performance counter
10 void PCreset(perf_counter* c);
11
12 //starts the counter
13 void PCstart(perf_counter* c);
14
15 //stops the counter
16 void PCstop(perf_counter* c);
17
18 //returns the average time between start and stop of the performance counter
19 unsigned long long avg_cpu_cycles(perf_counter c);

```

Perf.counter.c

```

1 #include "..\Perf_counter.h"
2
3 void PCreset(perf_counter* c) {
4     c->tot = 0;
5     c->cnt = 0;
6     c->calls = 0;
7 }
8
9 void PCstart(perf_counter* c) {
10     c->cnt = sds_clock_counter();
11     c->calls++;

```

```

12 }
13
14 void PCstop(perf_counter* c) {
15     c->tot += (sds_clock_counter() - c->cnt);
16 }
17
18 unsigned long long avg_cpu_cycles(perf_counter c) {
19     return (c.tot / c.calls);
20 }

```

ConversionFunctions.h

```

1 #ifndef FUNCTIONS_H
2 #define FUNCTIONS_H
3
4 #include "../TYPES.h"
5
6 /**
7  * Transforms the given character to fit in the "BASE" type.
8  */
9 BASE charToBase(char c);
10
11 /**
12  * Transforms the given BASE type to a character.
13  */
14 char baseToChar(BASE base);
15
16 #endif

```

ConversionFunctions.c

```

1 #include "../ConversionFunctions.h"
2
3 BASE charToBase(char c) {
4     switch (c) {
5         case 'a':
6         case 'A':
7             return A;
8             break;
9         case 'c':
10        case 'C':
11            return C;
12            break;
13        case 'g':
14        case 'G':
15            return G;
16            break;
17        case 't':
18        case 'T':
19            return T;
20            break;
21    }
22    return 0;
23 }
24
25 char baseToChar(BASE base) {
26     switch (base) {
27         case A:
28             return 'A';
29             break;

```

```

30     case C:
31         return 'C';
32         break;
33     case G:
34         return 'G';
35         break;
36     case T:
37         return 'T';
38         break;
39     }
40
41     return 'N';
42 }

```

FASTA_interface.h

```

1  #ifndef LOADREF_H
2  #define LOADREF_H
3
4  #include <stdio.h> //for reading files
5
6  #include "../TYPES.h"
7  #include "ConversionFunctions.h" //for the charToBase function
8
9  /**
10   * loads the genome from a FASTA file.
11   */
12 void loadGenome(char* filePath, GENOME* genome);
13
14 /**
15   * PRIVATE FUNCTION
16   * loads the info of the genome given at the beginning of the FASTA file
17   */
18 void loadGenomeInfo(FILE* fp, char* qname);
19
20 /**
21   * PRIVATE FUNCTION
22   * loads the genome itself.
23   */
24 void loadRefData(FILE* fp, REF* ref);
25
26 #endif

```

FASTA_interface.c

```

1  #include "../FASTA_interface.h"
2
3  void loadGenome(char* filePath, GENOME* genome) {
4      FILE* refFile = fopen(filePath, "r");
5
6      loadGenomeInfo(refFile, genome->Rname);
7      loadRefData(refFile, &(genome->ref));
8
9      fclose(refFile);
10 }
11
12 void loadGenomeInfo(FILE* fp, char* qname) { //first line of file
13     fgetc(fp); //skip the first @ in stream
14     int i = 0;
15     char read;

```

```

16  int stoploop = 0;
17  while (stoploop == 0) {
18      read = fgetc(fp);
19
20      if (read == ' ' || read == '\n') {
21          while (read != '\n') { //Get to next line of stream if it was a space
22              read = fgetc(fp);
23          }
24          qname[i] = '\0';
25          return;
26      } else {
27          qname[i] = read;
28          i++;
29      }
30  }
31 }
32
33 void loadRefData(FILE* fp, REF* ref) {
34     //read the file
35     ref->length = 0;
36     char read = fgetc(fp);
37     do {
38         if (charToBase(read) > 0) { //Valid base, no weird char like newline or space
39             ref->el[ref->length] = charToBase(read);
40             ref->length++;
41         }
42         read = fgetc(fp); //move on to next char
43     } while (((int) read) != 255); //end of the file
44 }

```

FASTQ_interface.h

```

1  #ifndef FASTQ_INF_H
2  #define FASTQ_INF_H
3
4  #include <stdio.h>
5  #include <stdlib.h>
6
7  #include "../TYPES.h"
8  #include "ConversionFunctions.h" //for the charToBase function
9
10 /**
11  * Loads the next read from the FASTQ file stream
12  * and stores it in the read type given.
13  * (fp = file pointer)
14  */
15 int loadNextRead(FILE* fp, READ* read);
16
17 /**
18  * PRIVATE FUNCTION:
19  * reads the Qname from the FASTQ file,
20  * given we start at the right position in the stream.
21  */
22 int readQName(FILE* fp, char* qname);
23
24 /**
25  * PRIVATE FUNCTION:
26  * reads the sequence from the FASTQ file,
27  * given we start at the right position in the stream.

```

```

28 */
29 int readSeq(FILE* fp, SEQ* seq);
30
31 /**
32  * PRIVATE FUNCTION:
33  * reads the qualities from the FASTQ file,
34  * given we start at the right position in the stream.
35  */
36 int readQualities(FILE* fp, READ* read);
37
38 /**
39  * PRIVATE FUNCTION:
40  * skips the stream until the next line break (newline) in the file
41  */
42 int nextLineOfFile(FILE* fp);
43
44 #endif

```

FASTQ_interface.c

```

1 #include "../FASTQ_interface.h"
2
3 int loadNextRead(FILE* fp, READ* read) {
4
5     if (readQName(fp, read->Qname) == 255)
6         return 255;
7     if (readSeq(fp, &(read->seq)) == 255)
8         return 255;
9     if (nextLineOfFile(fp) == 255)
10        return 255;
11    if (readQualities(fp, read) == 255)
12        return 255;
13
14    //display info on screen
15    //printf("length of the loaded sequence = %i\n", fastQLine->seq.length);
16
17    return 0;
18 }
19
20 int readQName(FILE* fp, char* qname) {
21     fgetc(fp); //skip the first @ in stream
22     int i = 0;
23     char read;
24     int stoploop = 0;
25     while (stoploop == 0) {
26         read = fgetc(fp);
27         if ((int) read == 255) { //end of file
28             //printf("\nfile finished\n");
29             return 255;
30         }
31
32         qname[i] = read;
33
34         switch (read) {
35             case '\n': //end of line
36                 //printf("\nend of line\n");
37                 stoploop = 1;
38                 break;
39             case ' ': //stop at the space

```

```
40     if (nextLineOfFile(fp) == 255)
41         return 255;
42     stoploop = 1;
43     break;
44 default:
45     i++;
46     break;
47 }
48
49 if (stoploop == 1)
50     qname[i] = '\0';
51 }
52
53 return 0;
54 }
55
56 int readSeq(FILE* fp, SEQ* seq) {
57     //Read the sequence
58     int i = 0;
59     char read;
60     int stoploop = 0;
61     while (stoploop == 0) {
62         read = fgetc(fp);
63
64         if ((int) read == 255) { //end of file
65             //printf("\nfile finished\n");
66             return 255;
67         }
68
69         switch (read) {
70             case '\n': //end of line
71                 //printf("\nend of line\n");
72                 stoploop = 1;
73                 break;
74             case 'n': //uncertain bases, trunc the read till here:
75             case 'N':
76                 if (nextLineOfFile(fp) == 255)
77                     return 255;
78                 stoploop = 1;
79                 break;
80             default:
81                 if (charToBase(read) > 0) { //valid base
82                     seq->el[i] = charToBase(read);
83                     //printf("%c", read);
84                     i++;
85                 }
86                 break;
87             }
88         }
89
90     seq->length = i;
91
92     return 0;
93 }
94
95 int readQualities(FILE* fp, READ* read) {
96     int i = 0;
97     char currChar;
```

```

98  int stoploop = 0;
99  while (stoploop == 0 && i < read->seq.length) {
100     currChar = fgetc(fp);
101
102     if ((int) currChar == 255) { //end of file
103         //printf("\nfile finished\n");
104         return 255;
105     }
106
107     if (currChar == ' ')
108         currChar = '\0';
109
110     read->qualities[i] = currChar;
111
112     switch (currChar) {
113     case '\n': //end of line
114         //printf("\nend of line\n");
115         stoploop = 1;
116         break;
117     case ' ': //stop at the space
118         if (nextLineOfFile(fp) == 255)
119             return 255;
120         stoploop = 1;
121         break;
122     default:
123         i++;
124         break;
125     }
126 }
127
128 if (!(i < read->seq.length)) {
129     if (nextLineOfFile(fp) == 255)
130         return 255;
131 }
132
133 return 0;
134 }
135
136 int nextLineOfFile(FILE* fp) {
137     char currChar = fgetc(fp);
138
139     while (currChar != '\n') {
140         if ((int) currChar == 255) {
141             //printf("file finished\n");
142             return 255;
143         }
144
145         currChar = fgetc(fp);
146     }
147
148     return 0;
149 }

```

SAM_interface.h

```

1  #ifndef SAM_INF_H
2  #define SAM_INF_H
3
4  #include <stdio.h>

```

```

5 #include <string.h>
6
7 #include "../PARAMS.h"
8 #include "ConversionFunctions.h"
9
10 /**
11  * Initializes the mapped_read type with the defaults for this implementation:
12  * Rnext, Pnext and Tlen all set to 0.
13  */
14 void initMappedRead(MAPPED_READ* mapped_read);
15
16 /**
17  * Writes the mapped_read type to the given file (fp = file pointer)
18  */
19 void writeSamLine(FILE* fp, MAPPED_READ mapped_read);
20
21 #endif

```

SAM_interface.c

```

1 #include "../SAM_interface.h"
2
3 void initMappedRead(MAPPED_READ* mapped_read) {
4     //initialize defaults
5     mapped_read->Rnext[0] = '*';
6     mapped_read->Pnext = 0;
7     mapped_read->Tlen = 0;
8 }
9
10 void writeSamLine(FILE* fp, MAPPED_READ mapped_read) {
11
12     fprintf(fp, "%s\t", mapped_read.read.Qname);
13     fprintf(fp, "%i\t", mapped_read.Flag);
14     fprintf(fp, "%s\t", mapped_read.Rname);
15     fprintf(fp, "%i\t", mapped_read.Pos);
16     fprintf(fp, "%i\t", mapped_read.MapQ);
17     fprintf(fp, "%s\t", mapped_read.CIGAR);
18     fprintf(fp, "%s\t", mapped_read.Rnext);
19     fprintf(fp, "%i\t", mapped_read.Pnext);
20     fprintf(fp, "%i\t", mapped_read.Tlen);
21
22     //print sequence
23     for (SEQ_INDEX i = 0; i < mapped_read.read.seq.length; i++) {
24         fputc(baseToChar(mapped_read.read.seq.el[i]), fp);
25     }
26     fputc('\t', fp);
27
28     fprintf(fp, "%s\t", mapped_read.read.qualities);
29
30     fprintf(fp, "\n");
31 }

```

initMem.h

```

1 #ifndef INITMEM_H
2 #define INITMEM_H
3
4 #include "../PARAMS.h"
5 #include "../TYPES.h"
6

```



```

7 #include "sds_lib.h" //for sds_alloc function
8
9 /**
10  * reserves space for the sequence (forward and reserve).
11  * Size is determined in the PARAMS.h file.
12  * will store pointers to the reserved memory location at the given pointers
13  */
14 int initSeq(BASE** LRseq, BASE** RLseq);
15
16 /**
17  * reserves space for the reference.
18  * Size is determined in the PARAMS.h file.
19  * will store pointer to the reserved memory location at the given pointer
20  */
21 int initRef(BASE** ref);
22
23 /**
24  * reserves space for the matrix.
25  * Size is determined in the PARAMS.h file.
26  * will store pointers to the reserved memory location at the given pointers
27  */
28 int initAlignMatrixAddrSpace(CELL** matrix);
29
30 #endif

```

initMem.c

```

1 #include "../initMem.h"
2
3 int initSeq(BASE** LRseq, BASE** RLseq) {
4     //in physical memory
5     *LRseq = (BASE *) sds_alloc(sizeof(BASE) * seqMax);
6     *RLseq = (BASE *) sds_alloc(sizeof(BASE) * seqMax);
7
8     if (LRseq) {
9         sds_free(LRseq);
10        return -1;
11    } else if (RLseq) {
12        sds_free(RLseq);
13        return -1;
14    } else {
15        return 0;
16    }
17 }
18
19 int initRef(BASE** ref) {
20     //in physical memory
21     *ref = (BASE *) sds_alloc(sizeof(BASE) * refMax);
22
23     if (ref) {
24         sds_free(ref);
25         return -1;
26     } else {
27         return 0;
28     }
29 }
30
31 int initAlignMatrixAddrSpace(CELL** matrix) {
32     //INIT MATRIX (in physical memory)

```

```

33 *matrix = (CELL *) sds_alloc(sizeof(CELL) * seqMax * refMax);
34
35 if (matrix) {
36     sds_free(matrix);
37     return -1;
38 } else {
39     return 0;
40 }
41 }

```

align.h

```

1  #ifndef ALIGN_H
2  #define ALIGN_H
3
4  #include "../PARAMS.h"
5  #include "../TYPES.h"
6  #include "smithWaterman.h"
7  #include "smithWatermanHW.h"
8
9  /**
10 * Aligns the sequence in both forward and reverse way and pick's the best option.
11 * The full MAPPED_READ-type will be filled in.
12 *
13 * @param genome: the reference sequence
14 * @param read: the FASTQ information of the sequence to match
15 * @param mapped_read: a pointer to the mapped_read type to be filled in
16 * @param addrSpaceMatrix: array (1D) of the required length to fill in, in physical memory
17 * @param addrSpaceReverseSeq: pointer to reserved memory for the reversed sequence
18 * @param revRead: the reverse sequence FASTQ information
19 */
20 void align(GENOME genome, READ* read, MAPPED_READ* mapped_read,
21           CELL* addrSpaceMatrix, BASE* addrSpaceReverseSeq, READ* revRead);
22
23 /**
24 * displays the first 15 rows and columns of the matrix on the screen,
25 * as well as the corresponding reference and sequence
26 *
27 * @param ref: the reference sequence
28 * @param seq: the sequence to match
29 * @param matrix: array (1D) of the required length to fill in, in physical memory
30 */
31 void alignHW(GENOME genome, READ* read, MAPPED_READ* mapped_read,
32             CELL* addrSpaceMatrix, BASE* addrSpaceReverseSeq, READ* revRead);
33
34
35 /**
36 * PRIVATE FUNCTION:
37 * fills in the reverse sequence address space with the complementary
38 *
39 * @param LeftToRight: the FASTQ information of the sequence
40 * @param RightToLeft: the reverse FASTQ information
41 * @param addrSpaceReverseSeq: address space reserved for the reverse sequence
42 */
43 void reverseSeq(READ LeftToRight, READ* RightToLeft, BASE* addrSpaceReverseSeq);
44
45 /**
46 * PRIVATE FUNCTION:
47 * Generates the CIGAR string from the filled in matrix and the maximum position

```

```

48  *
49  * @param CIGAR: an array of chars where the finished CIGAR string should be stored
50  * @param matrix: the filled in matrix by Smith-Waterman
51  * @param maxPos: Position of the maximum in the matrix
52  *
53  * @return 0 if a full CIGAR was generated;
54  *         -1 if the CIGAR would be too long according to maxCigarParts parameter
55  */
56 int generateCIGAR(char* CIGAR, CELL* matrix, POS maxPos);
57
58 /**
59  * PRIVATE FUNCTION:
60  * Generates the mapping quality from the length of the sequence and the maximum value
61  *
62  * @param mapQ: returned mapping quality
63  * @param max_value: max value in the matrix filled in by Smith-Waterman
64  * @param seqLength: length of the sequence used to fill in the matrix
65  */
66 void generateMapQ(uint8_t* mapQ, CELL_VALUE max_value, SEQ_INDEX seqLength);
67
68 #endif

```

align.c

```

1  #include "../align.h"
2
3  //////////////////////////////////
4  //PRIVATE FUNCTIONS://
5
6
7  //translating coordinates to address
8  static inline MATRIX_INDEX coordToAddr(SEQ_INDEX row, REF_INDEX column) {
9      return ((MATRIX_INDEX) row) + (MATRIX_INDEX) column * seqMax;
10 }
11
12
13 static inline POS goToDirection(CELL* matrix, POS pos) {
14     switch (matrix[coordToAddr(pos.row, pos.col)].d) {
15     case 1:
16         return (POS ) { pos.row - 1, pos.col - 1 } ;
17         break;
18     case 2:
19         return (POS ) { pos.row, pos.col - 1 } ;
20         break;
21     case 3:
22         return (POS ) { pos.row - 1, pos.col } ;
23         break;
24     default:
25         printf("something went wrong when backtracking");
26         return (POS ) { 0, 0 } ;
27         break;
28     }
29 }
30
31
32
33 int generateCIGAR(char* CIGAR, CELL* matrix, POS maxPos) {
34     uint8_t cigarPartsCounter = 1;
35     char ipCIGAR[buffSize];

```

```

36  char ipCIGAR_temp[buffSize];
37
38  //reset all strings
39  *CIGAR = '\0';
40  ipCIGAR[0] = '\0';
41  ipCIGAR_temp[0] = '\0';
42
43  SEQ_INDEX counter = 0;
44  char currentDir = '\0';
45
46  POS pos = maxPos;
47  DIRECTION d = matrix[coordToAddr(pos.row, pos.col)].d;
48
49  switch (d) {
50  case 1: //diagonal: match
51      currentDir = 'M';
52      break;
53  case 2: //up : insertion
54      currentDir = 'I';
55      break;
56  case 3: //left: deletion
57      currentDir = 'D';
58      break;
59  }
60
61  while (d != 0) {
62      switch (d) {
63      case 1:
64          if (currentDir == 'M') {
65              counter++;
66          } else {
67              cigarPartsCounter++;
68              if (cigarPartsCounter >= maxCigarParts)
69                  return -1;
70
71              sprintf(ipCIGAR_temp, "%i%c", counter, currentDir);
72              strcat(ipCIGAR_temp, ipCIGAR);
73              strcpy(ipCIGAR, ipCIGAR_temp);
74              ipCIGAR_temp[0] = '\0';
75
76              counter = 1;
77              currentDir = 'M';
78          }
79          pos = (POS ) { pos.row - 1, pos.col - 1 };
80          break;
81      case 2:
82          if (currentDir == 'I') {
83              counter++;
84          } else {
85              cigarPartsCounter++;
86              if (cigarPartsCounter >= maxCigarParts)
87                  return -1;
88
89              sprintf(ipCIGAR_temp, "%i%c", counter, currentDir);
90              strcat(ipCIGAR_temp, ipCIGAR);
91              strcpy(ipCIGAR, ipCIGAR_temp);
92              ipCIGAR_temp[0] = '\0';
93

```

```

94     counter = 1;
95     currentDir = 'I';
96 }
97 pos = (POS ) { pos.row, pos.col - 1 };
98 break;
99 case 3:
100     if (currentDir == 'D') {
101         counter++;
102     } else {
103         cigarPartsCounter++;
104         if (cigarPartsCounter >= maxCigarParts)
105             return -1;
106
107         sprintf(ipCIGAR_temp, "%i%c", counter, currentDir);
108         strcat(ipCIGAR_temp, ipCIGAR);
109         strcpy(ipCIGAR, ipCIGAR_temp);
110         ipCIGAR_temp[0] = '\0';
111
112         counter = 1;
113         currentDir = 'D';
114     }
115     pos = (POS ) { pos.row - 1, pos.col };
116     break;
117 }
118 // printf("row: %i\tcol: %i\tdirection: %c\tcounter: %i\tCIGAR: %s\n",
119 //         currPos.row, currPos.col, currentDirection, counter, ipCIGAR);
120
121     d = matrix[coordToAddr(pos.row, pos.col)].d;
122 }
123
124 //write last counted part also in string
125 sprintf(ipCIGAR_temp, "%i%c", counter, currentDir);
126 strcat(ipCIGAR_temp, ipCIGAR);
127 strcpy(ipCIGAR, ipCIGAR_temp);
128 ipCIGAR_temp[0] = '\0';
129
130 strcpy(CIGAR, ipCIGAR);
131
132 return 0;
133 }
134
135 void generateMapQ(uint8_t* mapQ, CELL_VALUE max_value, SEQ_INDEX seqLength) {
136     //I calculate quality based on amount of similarities and gap penalties:
137
138     //max_value is between 0 and seqLength * s => rescale between 0 and 254
139
140     *mapQ = (uint8_t) (((int) max_value) * 254) / (((int) seqLength) * s);
141 }
142
143 void reverseSeq(READ LeftToRight, READ* RightToLeft, BASE* addrSpaceReverseSeq) {
144     strcpy(RightToLeft->Qname, LeftToRight.Qname);
145
146     RightToLeft->seq.length = LeftToRight.seq.length;
147     RightToLeft->seq.el = addrSpaceReverseSeq;
148
149     //reverse AND complementary base
150
151     for (SEQ_INDEX i = 0; i < LeftToRight.seq.length; i++) {

```

```

152     RightToLeft->qualities[i] = LeftToRight.qualities[LeftToRight.seq.length
153         - i - 1];
154     BASE b = LeftToRight.seq.el[LeftToRight.seq.length - i - 1];
155     switch (b) {
156     case A:
157         RightToLeft->seq.el[i] = T;
158         break;
159     case C:
160         RightToLeft->seq.el[i] = G;
161         break;
162     case G:
163         RightToLeft->seq.el[i] = C;
164         break;
165     case T:
166         RightToLeft->seq.el[i] = A;
167         break;
168     }
169 }
170 }
171
172 //////////////////////////////////////////////////
173
174 //////////////////////////////////////
175 //PUBLIC FUNCTIONS://
176
177 void align(GENOME genome, READ* read, MAPPED_READ* mapped_read,
178     CELL* addrSpaceMatrix, BASE* addrSpaceReverseSeq, READ* revRead) {
179
180     mapped_read->read = *read;
181     CELL* matrix = addrSpaceMatrix;
182
183     //left to right
184     // printf("left to right alignment: \n\n");
185     POS maxPosLR = FillInMatrix(genome.ref, read->seq, matrix);
186     CELL_VALUE maxVal = matrix[coordToAddr(maxPosLR.row, maxPosLR.col)].value;
187
188     int retValueLR = generateCIGAR(mapped_read->CIGAR, matrix, maxPosLR);
189     if (retValueLR == 0) {
190         generateMapQ(&(mapped_read->MapQ), maxVal, read->seq.length);
191
192         POS pos = goToDirection(matrix, maxPosLR);
193         while (matrix[coordToAddr(pos.row, pos.col)].value != 0) {
194             pos = goToDirection(matrix, pos);
195         }
196
197         mapped_read->Pos = pos.col + 1;
198         mapped_read->Flag = 0;
199
200         strcpy(mapped_read->Rname, genome.Rname);
201     }
202
203     reverseSeq(*read, revRead, addrSpaceReverseSeq);
204
205     POS maxPosRL = FillInMatrix(genome.ref, revRead->seq, matrix);
206
207     int retValueRL = -1;
208
209     if (matrix[coordToAddr(maxPosRL.row, maxPosRL.col)].value > maxVal) {

```

```

210 //      printf("choosing the reverse alignment as the better one\n");
211 maxVal = matrix[coordToAddr(maxPosRL.row, maxPosRL.col)].value;
212 retValueRL = generateCIGAR(mapped_read->CIGAR, matrix, maxPosRL);
213
214 if (retValueRL == 0) {
215     generateMapQ(&(mapped_read->MapQ), maxVal, read->seq.length);
216
217     POS pos = goToDirection(matrix, maxPosRL);
218     while (matrix[coordToAddr(pos.row, pos.col)].value != 0) {
219         pos = goToDirection(matrix, pos);
220     }
221
222     mapped_read->Pos = pos.col + 1;
223     mapped_read->Flag = 16;
224
225     strcpy(mapped_read->Rname, genome.Rname);
226
227     //use reverse read
228     mapped_read->read = *revRead;
229 }
230 }
231
232 if (retValueLR != 0 && retValueRL != 0) {
233     //unmatched
234     mapped_read->Flag = 4;
235     strcpy(mapped_read->Rname, "*\0");
236     mapped_read->Pos = 0;
237     mapped_read->MapQ = 0;
238     strcpy(mapped_read->CIGAR, "*\0");
239 }
240 }
241
242 void alignHW(GENOME genome, READ* read, MAPPED_READ* mapped_read,
243             CELL* addrSpaceMatrix, BASE* addrSpaceReverseSeq, READ* revRead) {
244
245     mapped_read->read = *read;
246     CELL* matrix = addrSpaceMatrix;
247
248     //left to right
249     // printf("left to right alignment: \n\n");
250     MATRIX_INDEX addr = FillInHW(genome.ref.el, genome.ref.length, read->seq.el, read->seq.length);
251     POS maxPosLR = {addr % seqMax, addr / seqMax};
252
253     CELL_VALUE maxVal = matrix[coordToAddr(maxPosLR.row, maxPosLR.col)].value;
254
255     int retValueLR = generateCIGAR(mapped_read->CIGAR, matrix, maxPosLR);
256     if (retValueLR == 0) {
257         generateMapQ(&(mapped_read->MapQ), maxVal, read->seq.length);
258
259         POS pos = goToDirection(matrix, maxPosLR);
260         while (matrix[coordToAddr(pos.row, pos.col)].value != 0) {
261             pos = goToDirection(matrix, pos);
262         }
263
264         mapped_read->Pos = pos.col + 1;
265         mapped_read->Flag = 0;
266
267         strcpy(mapped_read->Rname, genome.Rname);

```

```

268     }
269
270     reverseSeq(*read, revRead, addrSpaceReverseSeq);
271
272     addr = FillInHW(genome.ref.el, genome.ref.length, revRead->seq.el, revRead->seq.length, mat
273     POS maxPosRL = {addr % seqMax, addr / seqMax};
274
275     int retValueRL = -1;
276
277     if (matrix[coordToAddr(maxPosRL.row, maxPosRL.col)].value > maxVal) {
278 //      printf("choosing the reverse alignment as the better one\n");
279     maxVal = matrix[coordToAddr(maxPosRL.row, maxPosRL.col)].value;
280     retValueRL = generateCIGAR(mapped_read->CIGAR, matrix, maxPosRL);
281
282     if (retValueRL == 0) {
283         generateMapQ(&(mapped_read->MapQ), maxVal, read->seq.length);
284
285         POS pos = goToDirection(matrix, maxPosRL);
286         while (matrix[coordToAddr(pos.row, pos.col)].value != 0) {
287             pos = goToDirection(matrix, pos);
288         }
289
290         mapped_read->Pos = pos.col + 1;
291         mapped_read->Flag = 16;
292
293         strcpy(mapped_read->Rname, genome.Rname);
294
295         //use reverse read
296         mapped_read->read = *revRead;
297     }
298 }
299
300 if (retValueLR != 0 && retValueRL != 0) {
301     //unmatched
302     mapped_read->Flag = 4;
303     strcpy(mapped_read->Rname, "\0");
304     mapped_read->Pos = 0;
305     mapped_read->MapQ = 0;
306     strcpy(mapped_read->CIGAR, "\0");
307 }
308 }

```

displayForDebug.h

```

1 #ifndef DISPLAYFORDEBUG_H
2 #define DISPLAYFORDEBUG_H
3
4 #include <stdio.h>
5
6 #include "../TYPES.h"
7
8 /**
9  * displays the first 15 rows and columns of the matrix on the screen,
10  * as well as the corresponding reference and sequence
11  *
12  * @param ref: the reference sequence
13  * @param seq: the sequence to match
14  * @param matrix: array (1D) of the required length to fill in, in physical memory
15  */

```



```

16 void displayMatrix(REF ref, SEQ seq, CELL matrix[seqMax * refMax]);
17
18 /**
19  * displays the maximum value and position
20  *
21  * @param maxPos: position of the maximum
22  * @param max: value of the maximum
23  */
24 void displayMax(POS maxPos, CELL_VALUE max);
25
26 #endif

```

displayForDebug.c

```

1  #include "../displayForDebug.h"
2
3  ///////////////////////////////////
4  //PRIVATE FUNCTIONS://
5
6
7  //translating coordinates to address
8  static inline MATRIX_INDEX coordToAddr(SEQ_INDEX row, REF_INDEX column) {
9      return ((MATRIX_INDEX) row) * refMax + (MATRIX_INDEX) column;
10 }
11
12 ///////////////////////////////////
13 //PUBLIC FUNCTIONS://
14
15 void displayMatrix(REF ref, SEQ seq, CELL matrix[seqMax * refMax]) {
16     printf("\n\r\n\r");
17
18     printf("Length of ref: %i", ref.length);
19
20     printf("\n\r\n\r");
21
22     printf("\t\t");
23     for (REF_INDEX i = 0; i <= ref.length && i < 15; i++) {
24         if (i == 0)
25             printf("\t");
26         else
27             printf("%i\t", ref.el[i - 1]);
28     }
29     printf("\n\r");
30
31     printf("\t");
32     for (REF_INDEX i = 0; i <= ref.length && i < 15; i++) {
33         printf("-----");
34     }
35     printf("\n\r");
36
37     for (SEQ_INDEX i = 0; i <= seq.length && i < 15; i++) {
38         if (i == 0)
39             printf("\t|\t");
40         else
41             printf("%i\t|\t", seq.el[i - 1]);
42     }
43
44     for (REF_INDEX j = 0; j <= ref.length && j < 15; j++) {
45         printf("%i\t", matrix[coordToAddr(i, j)].value);
46     }

```

```

46     printf("\n\r");
47 }
48 printf("\n\r\n\r");
49 }
50
51 void displayMax(POS maxPos, CELL_VALUE max) {
52     printf("Position of maximum = row %i, column %i => with maximum %i\n\r\n\r",
53         maxPos.row, maxPos.col, max);
54 }

```

smithWaterman.h

```

1  #ifndef SMITHWATERMAN_H
2  #define SMITHWATERMAN_H
3
4  #include "displayForDebug.h"
5  #include "../PARAMS.h"
6  #include "../TYPES.h"
7
8  #include <stdio.h> //for sprintf function
9  #include <string.h> //for strcat function
10
11 /**
12  * Fills in the matrix' first row and column with zeros
13  *
14  * @param refLength: the length of the reference sequence
15  * @param seqLength: the length of the sequence to match
16  * @param matrix: array (1D) of the required length to fill in, in physical memory
17  */
18 void initMatrix(REF_INDEX refLength, SEQ_INDEX seqLength, CELL* matrix);
19
20 /**
21  * Fills in the matrix according to the Smith-Waterman algorithm
22  *
23  * @param ref: the reference sequence
24  * @param seq: the sequence to match
25  * @param matrix: array (1D) of the required length to fill in, in physical memory
26  *
27  * @return the position of the maximum cell in the matrix
28  */
29 POS FillInMatrix(REF ref, SEQ seq, CELL matrix[refMax * seqMax]);
30
31 #endif

```

smithWaterman.c

```

1  #include "../smithWaterman.h"
2
3  //////////////////////////////////////////////////
4  //PRIVATE FUNCTIONS://
5
6  //similarity function
7  static inline CELL_VALUE sim(BASE a, BASE b) {
8      return (a == b) ? s : -s;
9  }
10
11 //translating coordinates to address
12 static inline MATRIX_INDEX coordToAddr(SEQ_INDEX row, REF_INDEX column) {
13     return ((MATRIX_INDEX) row) + (MATRIX_INDEX) column * seqMax;
14 }

```

```

15
16 //FILL THE CURRENT CELL => core of the algorithm
17 static inline CELL generateCell(CELL diagonal, CELL left, CELL up, BASE refVal,
18     BASE seqVal) {
19     //calculate the possible values
20     CELL diagonalCELL = { diagonal.value + sim(refVal, seqVal), 1 };
21     CELL leftCELL = { left.value - gp, 2 };
22     CELL upCELL = { up.value - gp, 3 };
23     CELL zeroCELL = { 0, 0 };
24
25     CELL upstreamA = (leftCELL.value > upCELL.value) ? leftCELL : upCELL;
26     CELL upstreamB = (diagonalCELL.value > zeroCELL.value) ?
27     diagonalCELL : zeroCELL;
28
29     CELL newCell = (upstreamA.value > upstreamB.value) ? upstreamA : upstreamB;
30
31     //Return the cell:
32     return newCell;
33 }
34
35 ///////////////////////////////////////////////////
36
37 ///////////////////////////////////////////////////
38 //PUBLIC FUNCTIONS://
39
40 void initMatrix(REF_INDEX refLength, SEQ_INDEX seqLength, CELL* matrix) {
41     //first row and first column of matrix = 0;
42     for (SEQ_INDEX row = 0; row < seqLength; row++) {
43         matrix[coordToAddr(row, 0)].value = 0;
44         matrix[coordToAddr(row, 0)].d = 0;
45     }
46     for (REF_INDEX col = 0; col < refLength; col++) {
47         matrix[coordToAddr(0, col)].value = 0;
48         matrix[coordToAddr(0, col)].d = 0;
49     }
50 }
51
52 POS FillInMatrix(REF ref, SEQ seq, CELL matrix[refMax * seqMax]) {
53     CELL_VALUE max = 0;
54     POS maxPos = { 0, 0 }; //position of the maximum value in the matrix
55
56     for (SEQ_INDEX row = 1; row <= seq.length; row++) {
57         for (REF_INDEX col = 1; col <= ref.length; col++) {
58
59             CELL Cell = generateCell(matrix[coordToAddr(row - 1, col - 1)],
60                 matrix[coordToAddr(row, col - 1)],
61                 matrix[coordToAddr(row - 1, col)], ref.el[col - 1],
62                 seq.el[row - 1]);
63
64             if (Cell.value > max) {
65                 maxPos = (POS) { row, col };
66                 max = Cell.value;
67             }
68
69             matrix[coordToAddr(row, col)] = Cell;
70         }
71     }
72

```

```

73 //////////////////////////////////////////////////
74 //DEBUG PURPOSES//
75 //  displayMatrix(ref, seq, matrix);
76 //  displayMax(maxPos, max);
77 //displayLL(&(matrix[coordToAddr(maxPos.row, maxPos.col)]));
78 //////////////////////////////////////////////////
79
80 return maxPos;
81 }

```

smithWatermanHW.h

```

1 #ifndef SMITHWATERMANHW_H
2 #define SMITHWATERMANHW_H
3
4 #include "displayForDebug.h"
5 #include "../PARAMS.h"
6 #include "../TYPES.h"
7
8
9 /**
10  * Fills in the matrix according to the Smith-Waterman algorithm
11  *
12  * @param ref: array of the reference sequence, in physical memory
13  * @param refLength: length of ref array
14  * @param seq: array of the sequence to match, in physical memory
15  * @param seqLength: length of seq array
16  * @param matrix: array (1D) of the required length to fill in, in physical memory
17  *
18  * @return the position of the maximum cell in the matrix
19  */
20 #pragma SDS data access_pattern(ref:SEQUENTIAL, seq:SEQUENTIAL, matrix:SEQUENTIAL)
21 #pragma SDS data sys_port(ref:AFI, seq:AFI, matrix:AFI)
22 #pragma SDS data mem_attribute(ref:PHYSICAL_CONTIGUOUS, seq:PHYSICAL_CONTIGUOUS, matrix:PHYSICAL_CONTIGUOUS)
23 #pragma SDS data zero_copy(ref[0:refMax], seq[0:seqMax], matrix[0:refMax*seqMax])
24 MATRIX_INDEX FillInHW(BASE ref[refMax], REF_INDEX refLength, BASE seq[seqMax], SEQ_INDEX seqLength, CELL matrix[refMax * seqMax]) {
25
26 #endif

```

smithWatermanHW.c

```

1 #include "../smithWatermanHW.h"
2 #include "../../PARAMS.h"
3
4 //////////////////////////////////////////////////
5 //PUBLIC FUNCTIONS://
6 MATRIX_INDEX FillInHW(BASE ref[refMax], REF_INDEX refLength, BASE seq[seqMax],
7     SEQ_INDEX seqLength, CELL matrix[refMax * seqMax]) {
8 #pragma HLS data_pack variable=matrix
9     // #pragma HLS PIPELINE
10
11     //maximum value searching:
12     CELL_VALUE currentMaxVal = 0;
13     POS maxPos = { 0, 0 };
14
15     //diagonal registers:
16     CELL_VALUE currDiagonal[seqMax];
17     CELL_VALUE prevDiagonal[seqMax];
18     CELL_VALUE preprevDiagonal[seqMax];
19

```

```

20 //init all on zero:
21 for (SEQ_INDEX index = 0; index < seqMax; index++) {
22     #pragma HLS UNROLL
23     prevprevDiagonal[index] = 0;
24     prevDiagonal[index] = 0;
25     currDiagonal[index] = 0;
26 }
27
28 //start at second column:
29 COL: for (MATRIX_INDEX currCol = 2; currCol < (refLength + seqLength);
30     currCol++) {
31     #pragma HLS loop_tripcount min=5000 avg=5150 max=5700
32     //#pragma HLS loop_merge
33     #pragma HLS PIPELINE
34
35     //new diagonal generation loop
36     DIAG: for (SEQ_INDEX i = 1; i < seqMax; i++) {
37         #pragma HLS UNROLL
38         SEQ_INDEX row = i;
39         MATRIX_INDEX col = currCol - i;
40
41         if (i < seqLength && col != 0 && col <= refLength) { //no edge cases
42
43             //Generation
44             CELL_VALUE sim = (ref[col - 1] == seq[row - 1]) ? s : -s;
45
46             //calculate the possible values
47             CELL diagonalCELL = { prevprevDiagonal[i - 1] + sim, 1 };
48             CELL leftCELL = { prevDiagonal[i] - gp, 2 };
49             CELL upCELL = { prevDiagonal[i - 1] - gp, 3 };
50             CELL zeroCELL = { 0, 0 };
51
52             CELL upstreamA =
53                 (leftCELL.value > upCELL.value) ? leftCELL : upCELL;
54             CELL upstreamB =
55                 (diagonalCELL.value > zeroCELL.value) ?
56                     diagonalCELL : zeroCELL;
57
58             CELL cell =
59                 (upstreamA.value > upstreamB.value) ?
60                     upstreamA : upstreamB;
61             //storing
62             currDiagonal[i] = cell.value;
63             matrix[row + col * seqMax] = cell;
64
65             //maximum value search:
66             if (cell.value > currentMaxVal) {
67                 maxPos = (POS ) { row, (REF_INDEX) col };
68                 currentMaxVal = cell.value;
69             }
70         }
71     }
72 }
73
74 //shift values:
75 SHIFT: for (SEQ_INDEX j = 0; j < seqMax; j++) {

```

```
78     #pragma HLS UNROLL
79     prevprevDiagonal[j] = prevDiagonal[j];
80     prevDiagonal[j] = currDiagonal[j];
81 }
82 }
83
84 return maxPos.row + ((MATRIX_INDEX) maxPos.col) * seqMax;
85 }
```

FACULTEIT INDUSTRIËLE INGENIEURSWETENSCHAPPEN
CAMPUS BRUGGE
Spoorwegstraat 12
8200 BRUGGE, België
tel. + 32 50 66 48 00
iiw.brugge@kuleuven.be
www.iw.kuleuven.be

