DELFT UNIVERSITY OF TECHNOLOGY

Low-Cost Smith-Waterman Acceleration

by

Matthijs Geers Fatih Han Çağlayan Roelof Willem Heij

A thesis submitted in partial fulfillment of the
degree of Bachelor of Science

in the
EWI faculty
Computer Engineering department

August 2013

DELFT UNIVERSITY OF TECHNOLOGY

Abstract

EWI faculty
Computer Engineering department

Bachelor of Science

by Matthijs Geers Fatih Han Çağlayan Roelof Willem Heij

Due to advancing technology, genetic sequencing has become cheaper over the years. This has caused the demand for computational power to grow even faster than Moore's law. To remedy this problem, we analyzed low-cost hardware solutions to parallelize the computational part of the genetic sequencing. We proposed a novel method for calculating the score matrix of the Smith-Waterman algorithm, which solves the bandwidth bottleneck in earlier solutions. This method calculates the score matrix differentially and a buffer keeps track of the maximum value. Due to the nature of the Smith-Waterman algorithm, the resulting implementation can do the calculations fully in parallel. Since it fits on an Artix 7 XC7A200T chip 908 times, this leads to more than a twelve-fold improvement in performance/price compared to modern supercomputing platforms.

Acknowledgements

First of all, we would like to show our gratitude and appreciation to Dr. Zaid Al-Ars for supervising our group and our project. Furthermore, we would like to thank Prof. Koen Bertels, head of the Computer Engineering Laboratory, for making this project possible. We also want to thank Dr.ing. I.E. Lager for his appreciated hard work of organizing and managing the Bachelor End Project. Last but not least, we would like to thank Matthijs Brobbel and Barry Strengholt for the cooperation on this project.

Matthijs Geers, Fatih Han Çağlayan, Roelof Willem Heij, Delft, The Netherlands, August 2013

Contents

Abstract **i**

Acknowledgements **ii**

1 Introduction **1**

1.1 Bioinformatics . . . 1

1.2 Basic biology for bioinformatics applications . . . 2

1.2.1 Cell . . . 2

1.2.2 Chromosomes . . . 2

1.2.3 DNA, RNA and Proteins . . . 3

1.2.4 Genes . . . 3

1.3 Sequencing . . . 4

1.4 Alignment . . . 4

1.5 Applications of DNA sequencing and alignment . . . 5

1.6 Problem definition . . . 5

2 Algorithms **7**

2.1 Dynamic programming . . . 7

2.1.1 Global alignment . . . 7

2.1.2 Local alignment . . . 8

2.2 Heuristic algorithms . . . 8

2.2.1 FASTA . . . 9

2.2.2 BLAST . . . 10

2.3 Comparison . . . 10

2.4 Smith-Waterman explained . . . 10

2.4.1 Score measure . . . 11

2.4.2 Gap penalty . . . 12

2.4.3 Performing the alignment . . . 12

2.4.4 Important note . . . 14

2.5 Optional extensions . . . 14

2.5.1 Greedy algorithms . . . 14

2.5.2 Recursive variable expansion . . . 14

2.5.3 Divide and conquer . . . 15

3 Design Considerations **16**

3.1 Platforms . . . 16

3.1.1 CPU . . . 16
3.1.2 GPU . . . 16
3.1.3 FPGA . . . 17
3.1.4 Relevant properties . . . 17
3.1.5 Platform selection . . . 18
3.2 Optimization . . . 18
3.2.1 Linear Systolic Arrays (LSA) . . . 19
3.2.2 Recursive Variable Expansion (RVE) . . . 20
3.3 Differential Smith-Waterman . . . 22

4 Implementation & Results **25**
4.1 Functional Unit . . . 25
4.2 Basic Components . . . 26
4.2.1 Differential counter . . . 26
4.2.2 Zerocheck . . . 26
4.2.3 Shift register . . . 27
4.3 Processing Element . . . 27
4.3.1 Structural Description . . . 28
4.3.2 Inner working . . . 29
4.3.2.1 Algorithm execution . . . 29
4.3.2.2 Coupling . . . 29
4.3.3 Linear Systolic Array . . . 30
4.4 Control . . . 31
4.4.1 Main control unit for query sequence input . . . 31
4.4.2 Design considerations and overview . . . 31
4.4.3 Control for jump moment . . . 33
4.4.4 Adjusted shift register . . . 33
4.4.5 PE buffer array . . . 34
4.5 FIFO buffer . . . 34
4.5.1 Specifications . . . 35
4.5.2 Block diagram representation . . . 35
4.5.3 Implementation . . . 36
4.6 Results . . . 38

5 Conclusions and recommendations **39**
5.1 Conclusions . . . 39
5.2 Recommendations for Future Research . . . 40
5.2.1 GPU implementations . . . 40
5.2.2 RVE . . . 40
5.2.3 Affine gap penalty . . . 40
5.2.4 Divide and conquer . . . 41
5.2.5 Differential calculation . . . 41

A Work distribution **42**

Bibliography **44**

Chapter 1

Introduction

Throughout history, mankind has always been interested in the working mechanisms of life. There has been a lot of research on this subject. In 1869, doctor Friedrich Miescher discovered the DNA [1]. It stands for DeoxyriboNucleic Acid. Miescher called his discovery nuclei. At that time, the function of the DNA was not yet known and it took almost one century to realize the real importance of this discovery.

In 1944, Avery, Macleod and McCarty published a paper in which they suggested that all genetic information was held in the DNA [2]. Before this publication, proteins were thought to be the carriers of this information. After 1944, a lot of research was done to investigate the workings of the DNA. Among these, one of the most important research projects was the Human Genome Project which started in 1990. This project had some important main goals to achieve:

- Identifying the approximately 20,000-25,000 genes that are present in human DNA
- Determining the sequences of the 3 billion chemical base pairs that make up human DNA
- Storing this information in databases

The project was finally completed in 2003, two years earlier than expected [3].

1.1 Bioinformatics

The term bioinformatics was officially introduced in 1978 by Paulien Hogeweg [4]. It has been widely used to address the analysis of genomic data. Hogeweg defines the term in a broader range and explains that “bioinformatics” refers to the study of informatic processes in biotic systems [5].

1.2 Basic biology for bioinformatics applications

1.2.1 Cell

In biology, the cell is considered to be the smallest building block of an organism. All plants, animals, humans, bacteria and fungi are organisms. All genetic information is contained within cells. Thus, every living being, from unicellular organisms up to human beings, are built from cells [6]. In the following sections, a top down approach will be used to show how a cell is built up. The part of a cell which makes it the genetic information carrier will also be discussed.

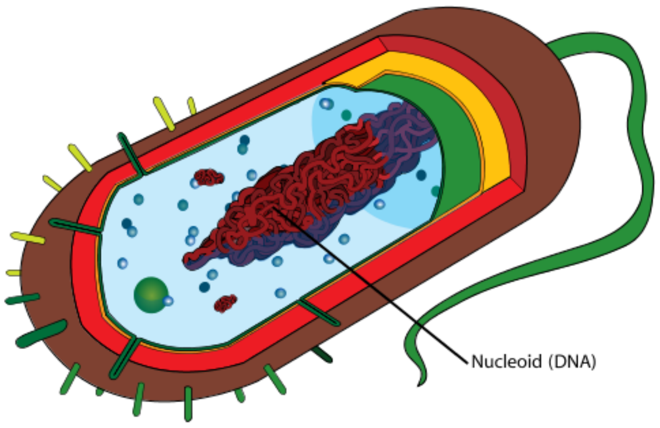

Figure 1.1: Diagram of a cell adapted from [7]

1.2.2 Chromosomes

In Figure 1.1 the diagram of a cell is given. Cells contain so-called DNA and, as can be seen, it is only a small part of the cell. The DNA is organized in structures called chromosomes, which are different for every type of organism. Each chromosome consists of one long DNA molecule, usually carrying several hundred or more genes. Human beings have 23 chromosome pairs, which is a total of 46 chromosomes. One of these pairs is sex-dependent and the remaining 22 chromosome pairs are similar for both men and women [6][8].

1.2.3 DNA, RNA and Proteins

Everything related to the development and functioning of organisms is encoded in DNA. The DNA is copied when a cell divides [6][8], so this information is inherited from the parents. That makes DNA the most important carrier of genetic information.

The roman alphabet is used to “encode” the English language consisting of 26 letters. A sequence of letters produces a word. Similarly, the genetic information in DNA is encoded by a sequence consisting of four different molecules called nucleotides. These are Adenine, Cytosine, Guanine and Thymine and they are abbreviated with the letters A, C, G and T [8].

The chemical structure of RNA, which stands for RiboNucleic Acid, is quite similar to that of DNA, because both are built up of nucleotides. The function of RNA is to copy the genetic information that is held in the DNA.

The chain of DNA, RNA and proteins is what organizes all chemical reactions in every living being, proteins being the information carriers in this chain [6].

1.2.4 Genes

Genes are a subset of DNA [6][8] and encode the proteins the cell will produce. They define how the cell will live and function and they also define what will and will not be inherited from the parents. An overview of the different hieararchy levels and the working of the DNA can be found in Figure 1.2.

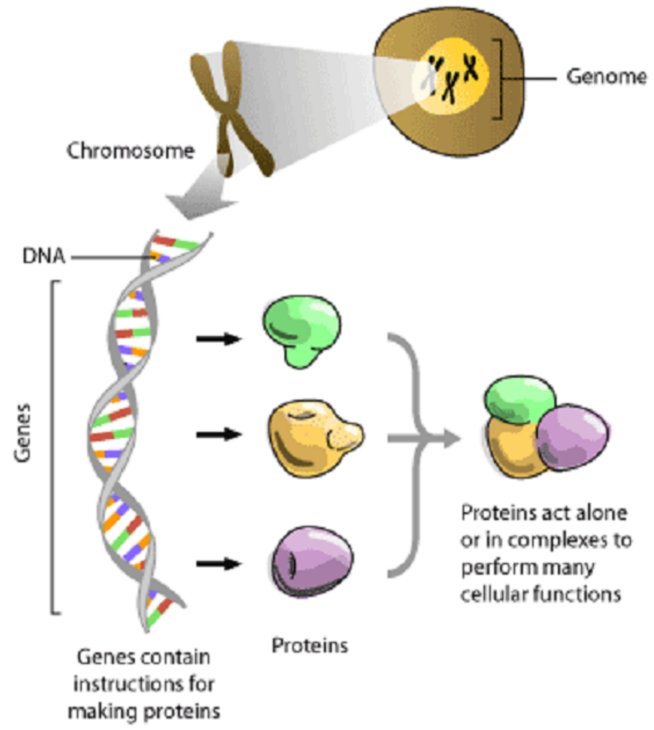
Genome
Chromosome
DNA
Genes
Genes contain instructions for making proteins
Proteins
Proteins act alone or in complexes to perform many cellular functions

FIGURE 1.2: Overview of the way DNA works adapted from [9]

1.3 Sequencing

Sequencing is the name of the process which is used to get information on the structure of DNA. This information about the structure of the molecule can be represented with symbols.

For DNA this means finding out the order of nucleotides. Since DNA consists of four different molecules, the four letters A, C, G and T are sufficient to provide information about the structure. Therefore, sequenced DNA is represented with different series of these four letters [6].

1.4 Alignment

Biological sequence alignment is like comparing and matching strings [10]. It is used to find regions of similarity in the database query and request query.

There are various algorithms to compare the similarity between two different sequences. These are explained in Chapter 2. The first algorithm was proposed in 1970 by Needleman [11]. This is a global method, which means that it tries to match as many characters as possible when comparing two complete sequences. In 1981, the Smith-Waterman (S-W) algorithm was proposed for optimal sub-sequence alignments [12]. Until now, there have been no better or newer proposals for sequence alignment anymore. It is a well-known algorithm in bioinformatics that finds the common regions of local similarity [13]. Local algorithms will be explained further in Chapter 2. Gotoh improved this algorithm one year later by taking linear gap penalties into account [14]. These optimal algorithms are, unfortunately, very slow and thus not practical to use. To compensate this, many heuristics have been developed, like FASTA [15] and BLAST [16]. These methods accelerate the database searching process at the cost of accuracy [17], so heuristics cannot guarantee an optimal alignment [18].

1.5 Applications of DNA sequencing and alignment

Better understanding of human DNA will allow for better personal treatment. Some people are at higher risk to get specific diseases and sometimes this is determined by heredity. Analyzing DNA can uncover this information, which means taking precautions for these types of illnesses will be much easier [8][19]. Analyzing genomic data also makes it possible to discover homologous genes and to get a deeper insight into the evolutionary history of molecules and species [20].

1.6 Problem definition

Sequenced material is saved in databases. The doubling time of the volume of these databases is becoming shorter and shorter with time, so there is hyperexponential growth in this case. In 1995, these databases were doubling in size roughly every 21 months [21]. By the year 1999, this rate increased to doubling every 15 months [22] [17]. Moore's law dictates that computational power of computers doubles roughly every 18 months, so at this point, the rate of growth was already faster than Moore's law could account for. However, the growth did not stabilize, as in 2003, the data volume in genetics started doubling every 12 months. This is an enormous growth compared to the data growth in other fields, where the amount of data doubles approximately every 20 months [23] [24].

Consequently, the problem is not the growing database itself. This data eventually needs to be processed. Comparing data with a database that is growing faster than Moore's law is the biggest challenge, so the challenge shifted from data gathering to data processing [9]. Scientists have been forced to implement heuristics, because database searching was too slow [17]. This results in a high loss of accuracy. Over the years, several hardware implementations have been proposed to accelerate the Smith-Waterman algorithm so that a loss of accuracy would not be necessary anymore, but these implementations vary in their financial feasibility.

Our goal for this project is to develop an implementation that is quick, cheap and highly accurate. Different algorithms for sequence alignment are weighted in Chapter 2. Subsequently, in Chapter 3, the different possible platforms are discussed and finally, a suitable platform is chosen to implement the algorithm. The implementation and results are explained in Chapter 4. Ultimately, in Chapter 5, the conclusion is presented.

Chapter 2

Algorithms

There are a lot of different algorithms to process large amounts of data, all with their own advantages and disadvantages. This document compares the most promising and well-known algorithms, focusing specifically on DNA sequence alignment. The best option is worked out in detail at the end of the chapter.

2.1 Dynamic programming

Dynamic programming seeks to solve complex problems by breaking them down into multiple smaller problems. The solutions of these smaller problems are then combined to reach the overall solution. This method appears to be both very precise and efficient. However, it does require a very large amount of computational power. Various methods to improve the speed of implementations of such algorithms have been proposed and proven effective [25]. Dynamic programming can be categorized in two classes: global alignment and local alignment.

2.1.1 Global alignment

This approach finds a very precise solution by relating two DNA sequences (database and query sequence) over their whole length. These two sequences are aligned with each other so that matches and mismatches are easy to identify. The downside of this process is that it requires a lot of computational power, as mentioned earlier.

The ruling global alignment algorithm is the Needleman-Wunsch algorithm [11]. This algorithm was the first to answer to the need of digital DNA analysis. Already proposed in 1970, this algorithm was revolutionary. Global alignment proved to work very well, but soon, proteins that only shared isolated regions of similarity were discovered. The global approach appeared to be insufficient for the new methods of DNA analysis, where smaller sequences were compared to one large sequence. Global approaches are hardly used nowadays for this reason.

2.1.2 Local alignment

Local alignment approaches are considered to provide the most precise and complete solution possible. Algorithms of this sort are thus highly valued. But due to computational restrictions, the implementation of local alignment algorithms is not profitable to everyone yet.

In 1980, the local alignment implementation of the Needleman-Wunsch algorithm was proposed by T.F. Smith and M.S. Waterman [26]. With current knowledge of DNA analysis, this algorithm provides the optimal alignment to every possible set of sequences by introducing or extending gaps in the sequences. It is for this reason that engineers all around the world have been looking for a fast and cost-effective implementation of this algorithm. Several proposals have been made in order to achieve this goal. Some seek to reduce the computational power required by the algorithms [14][16], while others seek to minimize the amount of data needed for the implementation with techniques such as alignment in linear space [27] or divide and conquer [28][29]. Although these solutions realize a major speed-up, computational power is still an issue.

2.2 Heuristic algorithms

When the implementation of dynamic programming algorithms appeared to be virtually impossible due to a lack of computational power, solutions that approximated an optimal alignment were introduced. An approximation is obviously faster than a dynamic programming solution since less and easier calculations need to be performed, but there is always a trade-off. The approximation will most times be less accurate than the exact solution determined by direct programming, because it might miss one or more solution that would be found in the exact solution. Only for small alignments there is still a chance that the heuristics can still find the optimum solution.

Heuristic algorithms with global approaches have not been made yet, so all heuristic algorithms have a local approach.

2.2.1 FASTA

FASTA comes from FAST-All, because it is intended to fasten all types of alingment. In 1987, Pearson and Lipman [15] introduced this software package with multiple smaller heuristic algorithms. Among these were FASTP (Fast Protein), for analysing proteins, and FASTN (Fast Nucleotides), for the analysis of nucleotides. Very soon after the release, all packages were combined into a single algorithm: FASTA. This algorithm provided a speedup in DNA analysis, but the sensitivity was insufficient for reliable alignment. The working of the algorithm is shown in Figure 2.1.

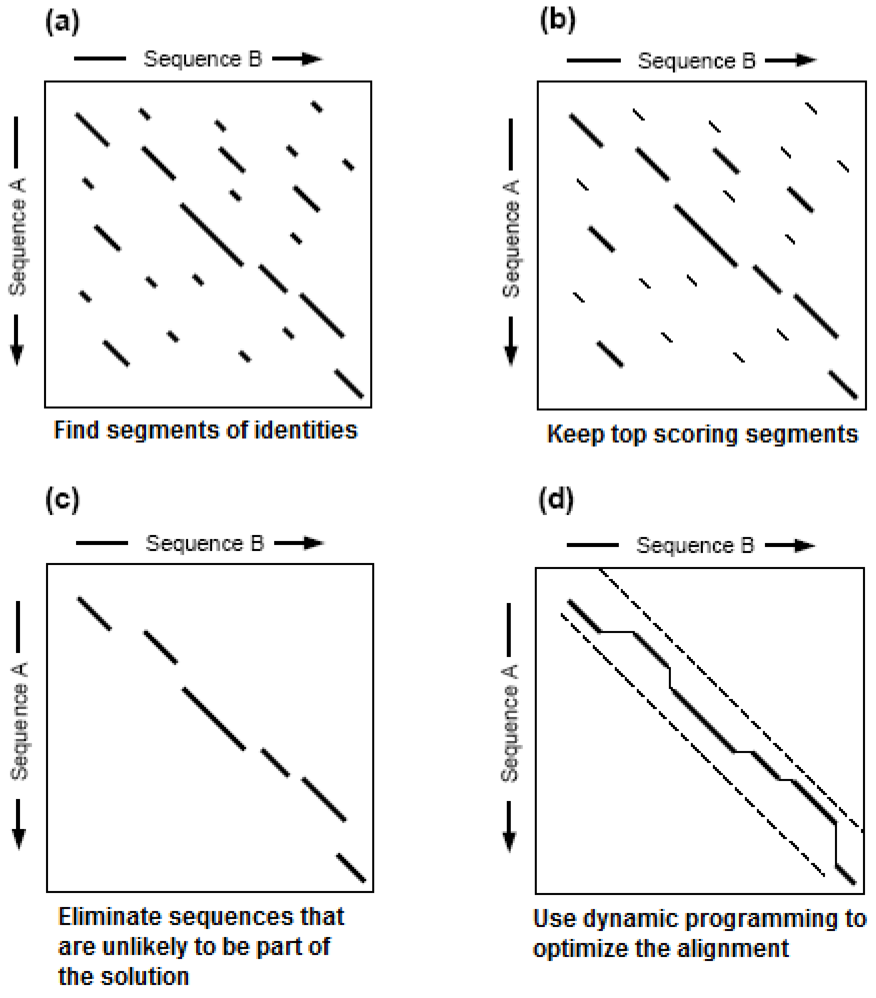
(a)
Sequence B
Sequence A
Find segments of identities
(b)
Sequence B
Sequence A
Keep top scoring segments
(c)
Sequence B
Sequence A
Eliminate sequences that are unlikely to be part of the solution
(d)
Sequence B
Sequence A
Use dynamic programming to optimize the alignment

FIGURE 2.1: FASTA algorithm adapted from [15]

In addition to the algorithm, Pearson and Lipman introduced a new file format, which they called FASTA as well [15].

2.2.2 BLAST

BLAST[30] (Basic Local Alignment Search Tool) is a heuristic algorithm that is used often in bioinformatics. It originated in 1990 as an improvement to FASTA, and was updated to its current form in 1997 [31]. BLAST consists of multiple variations, such as BLASTN, BLASTP [32][33], BLASTX and more. While BLAST requires less computational power than Smith-Waterman, it cannot guarantee optimal alignment of the database- and query-sequences like Smith-Waterman does.

Because of this, some questions can't be answered by a BLAST solution. Questions that BLAST can answer with a relative high degree of certainty are, for example:

- Where does a certain DNA sequence originate?
- Which DNA sequences share similarity with the sequence in question?

An extremely fast but even less optimal alternative to BLAST is called BLAT. The solutions provided by this package are therefore even less trustworthy.

2.3 Comparison

The heuristic algorithms are the faster approaches, but the dynamic programming algorithms have more potential with respect to accuracy, scalability and implementation options. Given the computational possibilities that can be reached with today's technology, the time required for getting a solution with the Smith-Waterman algorithm can be drastically decreased, so that it outperforms other approaches. For this reason, the Smith-Waterman algorithm is investigated further.

2.4 Smith-Waterman explained

The starting data consists of two DNA sequences, named a and b. Define matrix $H(i, j)$ with $0 \leq i \leq m+1$ where $m = length(a)$ and $0 \leq j \leq n+1$ where $n = length(b)$. Fill the first row and column with zeros. Introduce a score measure w, with $w(a_i, b_i) = w(match)$ if $a_i = b_i$ and $w(a_i, b_i) = w(mismatch)$ if $a_i \neq b_i$.

$$H_{i,j} = \max \begin{cases} 0 \\ H_{i-1,j-1} + w(match) \\ H_{i-1,j} + w(mismatch) \\ H_{i,j-1} + w(mismatch) \end{cases} \tag{2.1}$$

with $1 \le i \le m, 1 \le j \le n$

With this set of solutions the matrix H can be filled. However, there are some design options.

2.4.1 Score measure

The measure $w(a_i, b_j)$ determines the score for matches and mismatches between the sequences. The value of these scores may be chosen freely as long as the scores are used consistently the same for all sequences that need to be analyzed. Smith and Waterman themselves propose to use a $w(match)$ of 3, and $w(mismatch)$ of -2 [26]. Since the DNA data consists of four different symbols that are approximately equally distributed, this results in an average of 0 for the result of w.

Example 1

Consider the following sequences a and b respectively: $ATGCG$ and $ATCG$. The Needleman-Wunsch algorithm would align them as shown in Figure 2.2

$$\begin{array}{lllll} A & T & G & C & G \\ A & T & C & G & \\ or & & & & \\ A & T & G & C & G \\ & A & T & C & G \end{array}$$

FIGURE 2.2: Possible Needleman-Wunsch alignments for sequences a and b

Both options result in a score of $3 + 3 - 2 - 2 = 2$.

Smith-Waterman would introduce a gap in the middle of sequence b and align them as depicted in Figure 2.3

$$\begin{array}{ccccc} A & T & G & C & G \\ A & T & - & C & G \end{array}$$

FIGURE 2.3: Smith-Waterman alignment for sequences a and b

This results in a score of $3 + 3 + 3 + 3-$ some gap penalty $d = 12 - d$. But what is this gap penalty d?

2.4.2 Gap penalty

The big advantage of Smith-Waterman over Needleman-Wunsch is the usage of gaps. By introducing and extending gaps into one of both sequences, more optimal alignment can be obtained. It holds that $d = w(a_i, b_j) = w(-, b_j)$. The dash in $w(-, b_j)$ stands for an introduction of a gap. As with the w score measure, d has no predetermined value, and is to be chosen freely, depending on the importance of avoiding gaps. There are two approaches to the implementation of the gap penalty.

- The linear gap approach values each gap with an equal penalty d. This method is the easiest to implement, since the calculations are quite straightforward. However, this approach is not the most realistic according to today's knowledge of DNA [16].
- The affine gap approach differentiates between gap *introduction* and gap *extension*. Gap introduction is the step where the actual cut in the sequence happens. The extension occurs when the partial sequence is shifted further after the actual cut. This approach is more realistic than the linear one, because in reality it takes quite some effort to introduce a gap into a sequence, while moving two separate partial sequences further apart is rather easy. In this case d_{intr} and d_{ext} are still free to choose, as long as $d_{intr} > d_{ext}$.

2.4.3 Performing the alignment

At this point all information required to calculate H is available.

Example 2

Consider a linear gap implementation with $d = 1, w(match) = 3$ and $w(mismatch) = -2$. This leads to the following equations.

$$H_{i,j} = \max \begin{cases} 0 \\ H_{i-1,j-1} + w(match/mismatch) \\ H_{i-1,j} - d \\ H_{i,j-1} - d \end{cases} \tag{2.2}$$

Take the same sequences as in the previous example. The H matrix will be built based on these sequences. After the initialization of the matrix, the cells can be calculated. Every iteration, a new anti-diagonal can be calculated. These iterations are indicated

1	-	A	T	G	C	G
-	0	0	0	0	0	0
A	0	**3**				
T	0					
C	0					
G	0					

2	-	A	T	G	C	G
-	0	0	0	0	0	0
A	0	3	**2**			
T	0	**2**				
C	0					
G	0					

3	-	A	T	G	C	G
-	0	0	0	0	0	0
A	0	3	2	**1**		
T	0	2	**6**			
C	0	**1**				
G	0					

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

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

6	-	A	T	G	C	G
-	0	0	0	0	0	0
A	0	3	2	1	0	0
T	0	2	6	5	4	**3**
C	0	1	5	4	**8**	
G	0	0	4	**8**		

7	-	A	T	G	C	G
-	0	0	0	0	0	0
A	0	3	2	1	0	0
T	0	2	6	5	4	3
C	0	1	5	4	8	**7**
G	0	0	4	8	**7**	

8	-	A	T	G	C	G
-	0	0	0	0	0	0
A	0	3	2	1	0	0
T	0	2	6	5	4	3
C	0	1	5	4	8	7
G	0	0	4	8	7	**11**

TABLE 2.1: The iterations in the process of filling matrix H

with red in Table 2.1. The number of each iteration is indicated in bold, in the top-left of the matrix.

After this is done, the optimal sequence alignment can be determined from this matrix by working out another, simple, algorithm:

- Find the maximum value in H, named $H(i,j)$
- Trace back to the maximum value of $H(i-1,j), H(i-1,j-1)$ or $H(i,j-1)$, where $H(i-1,j)$ yields introducing a gap in sequence a, $H(i-1,j-1)$ implies that no gap is introduced or extended, and $H(i-1,j)$ introduces a gap in sequence b.
- Repeat this until a zero is encountered in the matrix. The optimal alignment is now complete. The solution for the example is given in Table 2.2. The algorithm returns the same solution as we found in **Example 1**, as expected.

	-	A	T	G	C	G
-	0	0	0	0	0	0
A	0	**3**	2	1	0	0
T	0	2	**6**	**5**	4	3
C	0	1	5	4	**8**	7
G	0	0	4	8	7	**11**

TABLE 2.2: The solution for optimal alignment of a and b, based on H

2.4.4 Important note

The implementation of the Smith-Waterman algorithm in computer code is slightly different from the explanation above. Several studies proposed alternatives that have better scaling [14] and are more accurate [16].

2.5 Optional extensions

There are several optional approaches and techniques that can provide a speed-up in the execution of the algorithm. These are discussed here. However, a decision regarding these and several other extensions is deferred to Section 3.2.

2.5.1 Greedy algorithms

Greedy alignment algorithms work directly with a measurement of the difference between two sequences rather than their similarity. In many problems, a greedy strategy does not in general produce an optimal solution, but nonetheless a greedy heuristic may yield locally optimal solutions that approximate a global optimal solution in a reasonable time.[34]

Since the greedy approach focuses on difference in stead of similarity, it is most effective when the compared sequences are near identical. When this is not the case, the algorithm will not be useful, and might possibly deliver the unique worst solution. In the analysis of DNA sequences, the greedy approach might be slightly beneficial since the sequences often resemble each other rather well.

2.5.2 Recursive variable expansion

This method achieves a speedup by exposing more parallelism in the algorithm. However, its implementation is a tough nut to crack. This is described in more detail in Section 3.2.2.

2.5.3 Divide and conquer

This approach reduces the size of H, the matrix that holds the solution to the algorithm, from $n \cdot m$ to $m + n$. However, it takes significantly more calculations to reach this solution.

Chapter 3

Design Considerations

In this chapter, several design considerations will be made as far as hardware platforms and algorithm optimizations go. Relevant options will be discussed briefly and among these, the best options will be chosen, supported by relevant arguments.

3.1 Platforms

3.1.1 CPU

Historically, the Central Processing Unit (CPU) has always been the working force of the personal computer. It is versatile and has gone through a development of optimizations and improvements for general purposes. This makes the CPU a very flexible worker, allowing any software instructions supplied by the user to run. The Smith-Waterman algorithm, however, is a process that takes a lot of time to compute, due to it consisting of many very small operations. Due to these time issues, the CPU is not a very suitable option for aligning large DNA sequences [13].

3.1.2 GPU

The Graphics Processing Unit (GPU) has a history of being used for a very specific task in a personal computer: graphics. Computing shader operations and other complex graphical problems have lead to the GPU being comprised of hundreds of small cores that are only suitable for very specific tasks. It has only recently become possible to manipulate the GPU manually instead of operating it through DirectX or OpenGL [13].

This recent development has been brought about by the introduction of the programming language Cuda/OpenCL by NVIDIA, making the GPU an option to consider for purposes like the Smith-Waterman algorithm [35].

3.1.3 FPGA

The field-programmable gate array (FPGA) is an external platform that is not usually integrated into a personal computer. It consists of a large programmable chip that allows the user to implement their application in hardware. It is well-suited for doing simple calculations, because it can be programmed to deal with a specific operation very quickly. Furthermore, these calculations can be paralellized by placing several of these job-specific cores on one chip. For the Smith-Waterman algorithm, this means that a cooperation between the CPU and the FPGA can be established, with the CPU supplying the simple operations that the FPGA can be made to handle [13].

3.1.4 Relevant properties

To draw an accurate conclusion, we listed our main design criteria and evaluated every one of these for each of the different platforms. Additionally, these evaluations have been summarized in Table 3.1.

- **Performance/price**: A typical desktop computer of about 600 euros, equipped with a quad-core CPU, delivers approximately 16 Giga cell updates per second (GCUPS) [9] (27 Mega cell updates per second (MCUPS)/euro). Another option is to use a GPU to implement the Smith-Waterman algorithm. On a desktop system similar to the one above, it is possible to achieve up to 21 GCUPS [36] or even 28 GCUPS [37] on a slightly faster system. This yields approximately 35 MCUPS/euro. A modern array of 16 FPGAs priced at about 20000 euros is capable of achieving up to 756 GCUPS [9], while a single, lower-end FPGA is capable of achieving 8 GCUPS at a price of around 500 euros [38][39]. Note that FPGA performance varies greatly and is difficult to estimate without extensive information. Therefore, let us use the best result when comparing, which is approximately 38 MCUPS/euro [9]. In initial costs, the CPU scores low as compared to the GPU and FPGA. This is because the CPU is not specialized for parallel operations while the other two are. It must be noted, though, that with multi-core CPUs on the rise, this may change in the near future.
- **Cumulative costs**: Distinction can be made between initial costs and cumulative costs (energy). When it comes to energy costs, the FPGA performs best. CPUs

Platform	Performance/price	Cumulative costs	Scalability
CPU	27 MCUPS/euro	-	+-
GPU	35 MCUPS/euro	-	+
FPGA	38 MCUPS/euro	+	++

Table 3.1: Comparison between different hardware platforms

and GPUs are more general purpose than FPGAs, so they have significantly more overhead, decreasing the efficiency of both the energy and initial costs.

- **Scalability**: As a low-cost solution, the CPU is very non-scalable [13]. There are supercomputers with a lot of processor capacity, but fow low cost solutions the scalability of the CPU is low, because consumer motherboards normally don't allow more than one CPU. While some desktop computers offer space for 2 CPUs and some supercomputers offer space for a couple hundred CPUs, they are extremely expensive and do not make for an efficient option. GPUs are a little more scalable, allowing up to 4 in a regular desktop computer. FPGAs, on the other hand, are very scalable as they do not require an architecture to run within. They are stand-alone systems. This allows the user to combine as many of them as they like, without any extra costs for up-scaling. This makes the FPGA the most scalable solution.

3.1.5 Platform selection

The most important goal of the project is to achieve as much performance as possible for the lowest possible price. In terms of this, the GPU and FPGA are in the same range, leaving the CPU slightly behind. As for cumulative costs, CPU and GPU systems use a very high amount of energy in order to perform their calculations, while FPGAs are much more energy-efficient [13]. Finally, CPUs and GPUs are decently scalable in supercomputers, but buying a complete supercomputer is an enormous step to take for everyone but large companies. The FPGA, on the other hand, can be made as scalable as the designer wishes. Based upon these observations, we have decided to realize an FPGA acceleration of the Smith-Waterman algorithm. The main choice factors are summarized in Table 3.1

3.2 Optimization

Executing the Smith-Waterman algorithm is a time-consuming task [9]. Luckily, some arrangements can be made that significantly speed up the calculations. These are discussed in detail in this section.

3.2.1 Linear Systolic Arrays (LSA)

As is discussed in Section 4.3, the computational core of the FPGA implementation is the individual processing unit called the processing element (PE). One PE calculates one element of the H-matrix every clock cycle. When executing on a CPU, this would mean that every core is essentially one of these processing elements and they can be operated at a very high clock frequency, although due to caching overhead it would likely take a CPU more than one clock cycle to finish a single calculation.

What the FPGA platform allows us to do is to chain a lot of processing elements into a so-called linear systolic array [40] and keep increasing the frequency up to the point where the processing elements no longer have enough time to finish their operation. This equates to a many-core CPU (up to a couple hundred) at a lower frequency, which would be a bad idea for a typical CPU operation but perfect for a parallel problem such as the Smith-Waterman algorithm. Studies show varying benefits from linear systolic arrays, but are generally somewhere between a factor 1.5 and 4 [40][41]. A trade-off can be made by increasing the speed of every processing element at the cost of more chip area. This is discussed further in Section 3.2.2.

Data dependencies are an important factor in linear systolic arrays. As was seen in Section 2.4, three adjacent cell values are needed to compute the next one. Consequently, computed matrix values need to be passed on to the neighboring processing element as soon as they are available. This means that not all processing elements in the systolic array can start at the same time, as they need data that is computed by their peers. This is illustrated in Figure 3.1. The numbers in the matrix indicate the amount of clock ticks up until the moment the corresponding cell is computed.

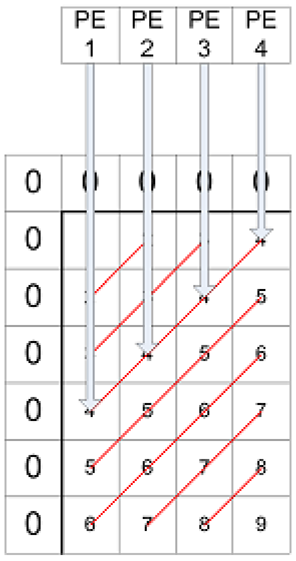

FIGURE 3.1: LSA structure adapted from [9]

In a practical application, a sequence is never equally long to the amount of processing elements. Therefore, it is necessary to have a processing element jump to a next column once it has finished one. In Figure 3.2 it can be seen how this jump is made. At clock tick 7, PE 2 up to 4 keep computing their own columns while PE 1 jumps to the new column. Eventually, when the end of the sequence has been reached, any processing element that gets invalid data is padded with zeroes. Implementing this in hardware is rather complicated and is explained in detail in Chapter 4.

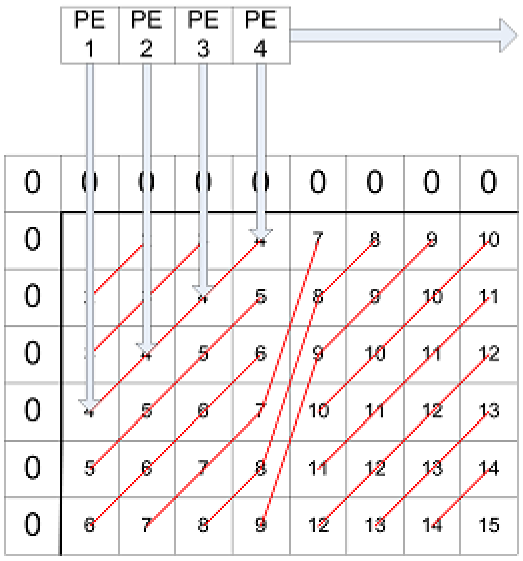

FIGURE 3.2: LSA jump mechanism adapted from [9]

3.2.2 Recursive Variable Expansion (RVE)

It has become clear from the regular linear systolic array implementation that, due to the fact that Smith-Waterman has three data dependencies from adjacent cells, it takes a while before all processing elements have something to compute. This is especially the case for relatively small sequences. If we could remove the data dependency altogether, the array would become 100% parallelizable, effectively removing the computational bottleneck.

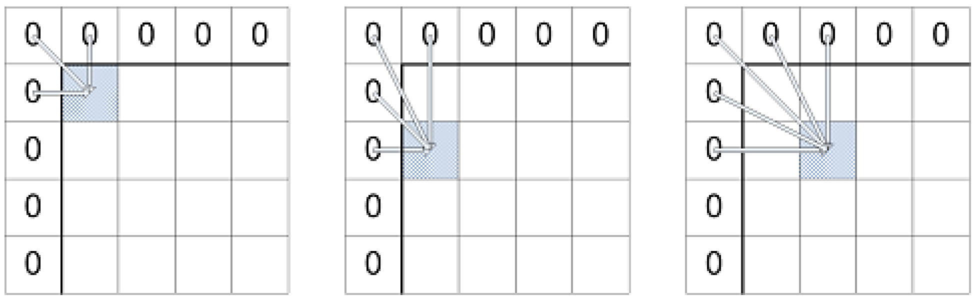

FIGURE 3.3: RVE data dependencies adapted from [9]

	Speed-up factor	Increased FPGA chip area cost factor	Ratio
[43]	1.4	1.3	1.08
[44]	2.5	2.3	1.09
[45]	2.3	2	1.15
[39]	2.3	2.8	0.82

Table 3.2: Approximate speed gain and area loss from RVE, rounded to 1 decimal

This is exactly what can be realized through recursive variable expansion, introduced in [42]. It removes data dependencies by computing H-matrix values directly from the first row and column of zeroes. Those, in addition to the sequence information for that specific matrix position, are then the only values necessary to compute a matrix value, as seen in Figure 3.3. It does, however, make the Smith-Waterman equation a lot more complicated. This is shown in Equation 3.1.

$$H_{i,j} = \max \begin{cases} H_{i,j-2} - 2d \\ H_{i-1,j-2} - d + S_{i,j-1} \\ H_{i-1,j-2} - d + S_{i,j} \\ H_{i-2,j-2} + S_{i,j} + S_{i-1,j-1} \\ H_{i-2,j-1} - d + S_{i,j} \\ H_{i-2,j-1} - d + S_{i-1,j} \\ H_{i-2,j} - 2d \\ 0 \end{cases} \tag{3.1}$$

Despite these disadvantages, it might still be beneficial to use recursive variable expansion. Table 3.2 illustrates both the benefit and handicap of doing so, judging by different studies.

Since less FPGA chip area usage per processing element means that we can put an increased number of processing elements on the chip, we can assume that area is just as relevant as relative speed-up. Consequently, the ratio between speed-up and increased area cost is a fairly accurate method of estimating the performance gain or loss we would obtain if we decided to use recursive variable expansion in our implementation. Judging from Table 3.2, the approximate overall performance gain of 10% is not worth the limited time investment that can be made for this thesis. Therefore, the decision was against implementing recursive variable expansion.

3.3 Differential Smith-Waterman

Since cell values in a score matrix can grow very large, calculating and storing this matrix is very area-inefficient. Furthermore, for implementations which return the maximum of the alignment only (max-only implementations), bandwidth is also a bottleneck. Computing and saving the Smith-Waterman differentially solves both of these problems [46]. An adder combined with a buffer are necessary to keep track of the maximum value. Thus, in the case of a max-only implementation, an assessment needs to be made whether to implement differential Smith-Waterman. Since solving the bandwidth limit and decreasing the size of internal signals of the PEs should weigh up against a little more required chip area, it is presumed feasible. In the following section, the method of calculating the differential values is explained and afterwards, this is demonstrated through an example.

First equation 2.2 needs to be rewritten as follows:

$$H^*_{i,j} = \max \begin{cases} H_{i-1,j-1} + w(match/mismatch) \\ H_{i-1,j} - 1 \\ H_{i,j-1} - 1 \end{cases} \tag{3.2}$$

and $H_{i,j}$ is defined as:

$$H_{i,j} = \max \begin{cases} 0 \\ H^*_{i,j} \end{cases} \tag{3.3}$$

The variables $\delta^V_{i,j}$ and $\delta^H_{i,j}$ are chosen to define the vertical and horizontal differences, respectively. They are defined in Equations 3.4 and 3.5. Thus, instead of one big absolute value, every cell in the score matrix contains two differences. This is depicted in Table 3.3.

$$\delta^V_{i,j} = H_{i,j} - H_{i-1,j} \tag{3.4}$$

$$\delta^H_{i,j} = H_{i,j} - H_{i,j-1} \tag{3.5}$$

Equation 2.2, which describes how to calculate the value of a cell in the score matrix, can be rewritten to be differential, as shown in Equations 3.6 and 3.7. A proof for both

$\ddots$		$\vdots$		$\vdots$
$\cdots$	$\delta^H_{i-1,j-1}$	$\delta^V_{i-1,j-1}$ $H_{i-1,j-1}$	$\delta^H_{i-1,j}$	$\delta^V_{i-1,j}$ $H_{i-1,j}$
$\cdots$	$\delta^H_{i,j-1}$	$\delta^V_{i,j-1}$ $H_{i,j-1}$	$\delta^H_{i,j}$	$\delta^V_{i,j}$ $H_{i,j}$

TABLE 3.3: The differential H-matrix

this and the fact that the difference matrix contains the same amount of information as a conventional score matrix is given in [46].

$$\delta^V_{i,j} = \begin{cases} -H_{i-1,j}, & \text{if } H^*_{i,j} < 0 \\ \delta^{V*}_{i,j}, & \text{else} \end{cases} \tag{3.6}$$

$$\delta^H_{i,j} = \begin{cases} -S_{i,j-1}, & \text{if } H^*_{i,j} < 0 \\ \delta^{H*}_{i,j}, & \text{else} \end{cases} \tag{3.7}$$

To explain the method of differential Smith-Waterman by example, we assume the following parameters, for the simplicity of the example. Other values can be chosen as long as consistency is guaranteed:

- Sequence 1 = AGCA
- Sequence 2 = ACAC
- Gap penalty $d = 1$
- $w(match) = 2$
- $w(mismatch) = -1$

These parameters cause Equation 2.2 to transform into the following equation:

$$H_{i,j} = \max \begin{cases} 0 \\ H_{i-1,j-1} + w(match/mismatch) \\ H_{i-1,j} - 1 \\ H_{i,j-1} - 1 \end{cases} \tag{3.8}$$

	-	A	G	C	A
-	0	0	0	0	0
A	0	2	1	0	2
C	0	1	1	3	2
A	0	2	1	2	5
C	0	1	1	3	4

Table 3.4: Conventional score matrix

The score matrix in Table 3.4 follows according to the conventional method of calculating Smith-Waterman. On the other hand, when vertical and horizontal differences are used, this results in a differential score matrix, which is depicted in Table 3.5.

	-	A	G	C	A
-	0, 0	0, 0	0, 0	0, 0	0, 0
A	0, 0	2, 2	-1, 1	-1, 0	2, 2
C	0, 0	1, -1	0, 0	2, 3	-1, 0
A	0, 0	2, 1	-1, 0	1, -1	3, 3
C	0, 0	1, -1	0, 0	2, 1	1, -1

Table 3.5: An example of a differential score matrix

Translating the differential score matrix back and forth from the conventional score matrix is possible. Trace back is possible from both the differential score matrix and the conventional score matrix.

Chapter 4

Implementation & Results

This chapter covers the technical details of the implementation. First, an overview of the final system is discussed in Section 4.1. Subsequently, the working of non-trivial basic components needed for these units is explained in Section 4.2. Subsequently, the working of the processing element (PE) and the linear systolic array (LSA) is explained in Section 4.3. For the correct working of the LSA some logic is needed and that is explained in Section 4.4. Since a First in First out (FIFO) buffer is needed for communication with a computer, Section 4.5 describes the implementation of this buffer. Finally, Section 4.6 describes the achieved results.

Subsequently, the workings of some basic components needed for the PE and control are explained. How the processing element is composed of these basic components is next and also includes how the linear systolic array of PEs is implemented. Such an array implementation also requires control components to supply sequence characters to the PEs. Therefore, another section is dedicated to the implementation of this control.

4.1 Functional Unit

The functional unit, as can be seen in Figure 4.1, is a combination of control, a FIFO buffer and the PE array. It combines the array of processing elements with their required control counterparts to allow the system to work as a whole. On its input, it requires sequence data as supplied by a connected computer and on its output it delivers the absolute maximum value found by computing the Smith-Waterman matrix. This value changes over time until calculating the whole matrix has been finished.

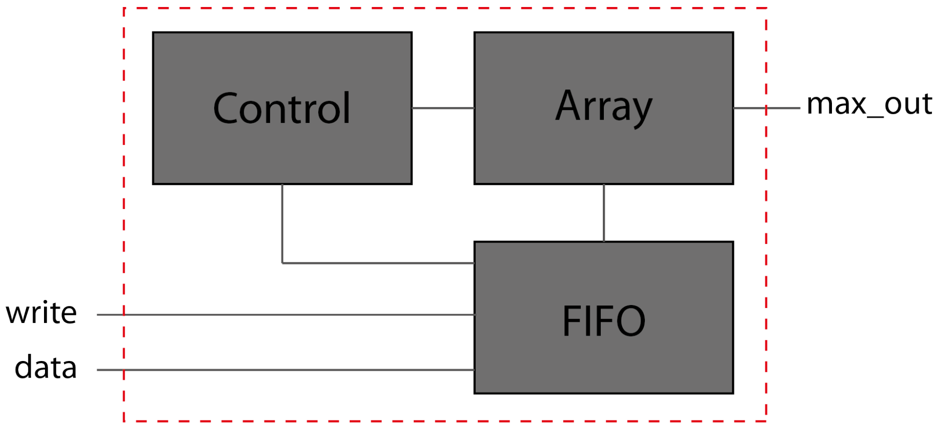

Figure 4.1: Block diagram of the functional unit

4.2 Basic Components

For the implementation of a complete functional unit, some basic components are needed. These are covered briefly below.

4.2.1 Differential counter

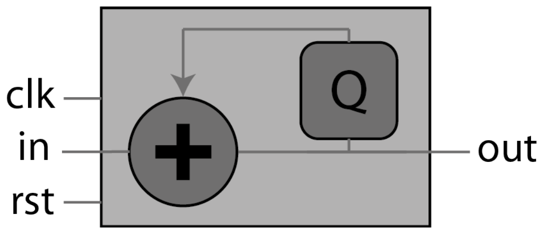

Figure 4.2: Decoder schematic

The decoder is a combination of a couple of simple components. Since differential values are used when computing matrix values, an additional operation is necessary to track what the absolute value is in every processing element. As seen in Figure 4.2, an adder is combined with a buffer (Q) to store an absolute value and add a differential value to it every clock tick. It can be reset independently of the top-level reset, which happens every time a PE jumps to a new column.

4.2.2 Zerocheck

The zerocheck component checks whether the absolute matrix value is smaller than zero and, if so, sets it to 0, like Smith-Waterman dictates (see Section 2.4 for reference). Additionally, the negative matrix value is subtracted from the differential values to

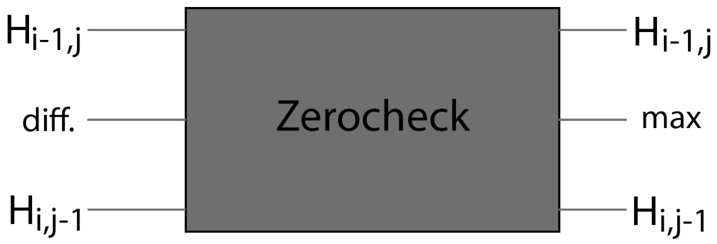

FIGURE 4.3: Zerocheck schematic

compensate this change of the absolute value. As seen in Figure 4.3, this is achieved by multiplexing the old absolute values with compensated signals.

4.2.3 Shift register

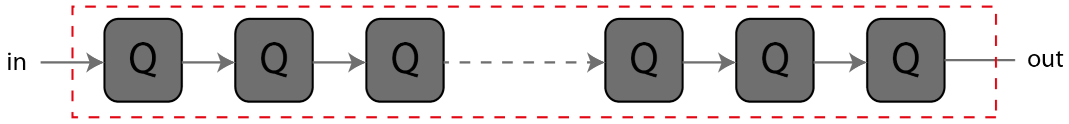

FIGURE 4.4: Modular shift register schematic

Shift registers consist of an array of buffers. A 2-bit register of 16 buffers and a 5-bit register of 12 buffers were made for different purposes. Connecting the output (the last register) to the input (the first register) results in a circular shift register that automatically refills itself. It can be filled initially by multiplexing an input with the circular connection. A modular design is shown schematically in Figure 4.4.

4.3 Processing Element

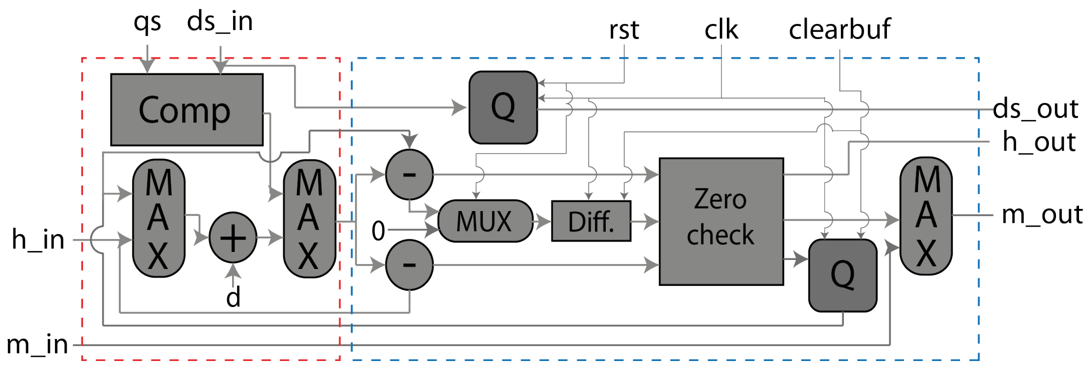

FIGURE 4.5: Processing element schematic

Out of all these basic blocks, a fundamental component for the calculation of the Smith-Waterman algorithm is built. This processing element (PE) is the core component of the alignment implementation. It calculates the outcome of matrix element $H_{i,j}$ based on the algorithm described in Equation 2.2. It is designed to function in an array of multiple PEs. Consequently it receives multiple signals from the preceding PE, and passes signals on to the next PE on its output.

4.3.1 Structural Description

The structural layout of the processing element is depicted in Figure 4.5. The element has seven inputs and three outputs. All of these will be discussed.

Inputs:

- qs: Query sequence input. two-bits vector that represents one of the four characteristic DNA letters.
- ds_{in}: Database sequence input. Of the same form as qs. This signal is connected to the ds_out through a buffer.
- h_{in}: Matrix value input. This five-bits input is connected to the output of another processing element through a buffer. The utility of this construction is discussed in Section 4.3.3.
- m_{in}: Maximum matrix value input. The 32-bits maximum value that has been encountered so far is held by this signal. It is also connected to the output of another processing element through a buffer. Section 4.3.3 provides an explanation on this connection.
- clk: Clock signal. This determines the rate at which the processing element functions. In order to achieve synchronism, every buffer is connected to this signal.
- rst: Reset signal. Top level reset, which resets every element in the entire system.
- *control_rst*: Control reset signal. A reset signal that only resets a single PE.

Components that need to be reset by both the reset and the control reset are connected to a signal called $clearbuf$. This signal performs an *or*-operation on the reset and the control reset signal.

Outputs:

- ds_{out}: Database sequence output. This signal is connected to the ds_in through a buffer.

- h_{out}: Matrix value output. This output contains one of the differential values of the preceding PE in the array, and passed it on to the next PE.
- m_{out}: Maximum matrix value output. This signal contain the highest encountered value up to that point. It is passed on to the next PE as reference.

4.3.2 Inner working

The inner working of the PE is separable in two parts. The first part executes the algorithm, and the second part provides the coupling between differential and absolute values.

4.3.2.1 Algorithm execution

This part, enclosed by the left dotted line in Figure 4.5, requires the H-matrix data, and a letter from both sequences to calculate the value of $H_{i,j}$. Since the PE uses a differential means of calculations, as described in Section 3.3, only two H-value inputs are required. One of these inputs enters the PE via h_{in}, while the other one is fed back inside the PE through a buffer. A max-operation is performed on these values, after which the gap penalty is added to the resulting value. Another max-operation is performed on this value and the outcome of a comparator that produces $w(match)$ or $w(mismatch)$, depending on the sequences. The output of this max-component concludes the algorithm execution of the PE. For future reference, this signal is labeled as alg.

The standard Swith-Waterman suggests the substraction of the gap penalty before the max operation, as seen in Equation 2.1. However, $max(a + d; b + d)$ can be reduced to $max(a; b) + d$. Using this logic, one less adder is needed in each PE.

Equation 2.1 also states that a max-operation with a zero is required. In the differential implementation this operation is moved to the end of the PE. When the operation would be performed at this point, the differential calculation could not be performed properly.

4.3.2.2 Coupling

This part, enclosed by the right dotted line in Figure 4.5, starts where the previous part ends, and wholly revolves around the mathmatical explanation given in 3.3. To transform the result of the first part of the PE into two differential results, subtractions are needed. The subtractions performed by the upper and lower subtractor are shown in Equation 4.1 and Equation 4.2.

$$Sub_{upper} = H_{i,j}1 = alg - H_{i-1,j}0 \tag{4.1}$$

$$Sub_{lower} = H_{i,j}0 = alg - H_{i,j-1}1 \tag{4.2}$$

Sub_{upper} is then connected to the *decoder* component, which turns the differential value into an absolute value. This value is then passed on to the *zerocheck* component, along with the outputs of the two subtractors. This component returns the differential values in their final form, through several logic operations as described in Subsection 4.2.2. Sub_{upper} leaves the PE, labeled h_{out}. The Sub_{lower} signal is fed back to the algorithm execution part of the PE. The *zerocheck* component also returns the final form of $H_{i,j}$, so that it can be compared to the current maximum m_{in}. The largest of these two leaves the PE, labeled m_{out}.

The letter from the database sequence that has been used by this PE is also passed on to the following PE by this part of the circuit. The usage of this will be discussed in Section 4.3.3.

4.3.3 Linear Systolic Array

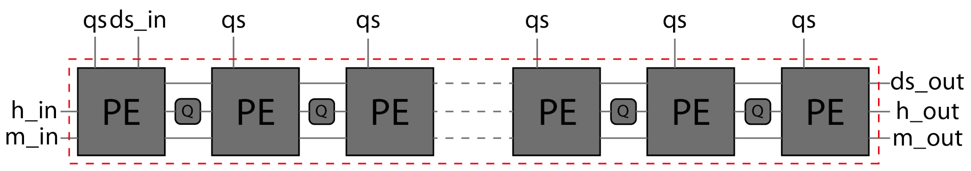

Figure 4.6: Schematic of implemented linear systolic array

In Section 3.2.1 a linear systolic array (LSA) implementation is described. This approach will be used to design an array of PEs described in 4.3. The design of these PEs enables them to work together. The interconnections between two arbitrary PEs in the array will allways be the same. The way they are interconnected is depicted in Figure 4.7. However, the first and last PE are connected differently. The first PE has a static *zero* connected to m_{in}. The h_{in} signal is connected to the output of a shiftregister, as it described in Section 4.2.3. The m_out of last PE is connected to a maxtracker that holds

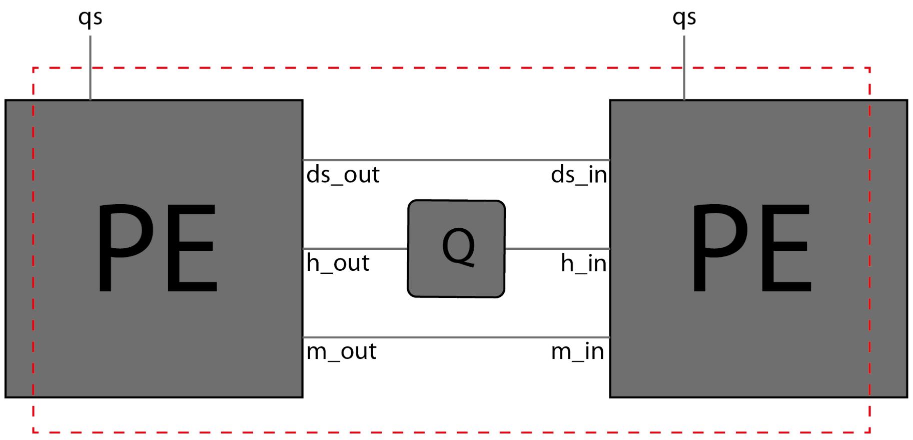

FIGURE 4.7: Close-up of interconnect in implemented linear systolic array

the maximum value of the LSA for future reference. The h_{out} signal is connected to the input of the same shiftregister that is connected to h_in of the first PE in the array. The ds_out signal of the last PE is obsolete, but will be present since all PEs have the same design. A total overview of the LSA of PEs is given in Figure 4.6.

4.4 Control

The array of PEs needs a new input every clock cycle. Every cycle, one character of the database sequence and four characters of the query sequence are needed. As soon as the first PE has reached the end of the database sequence, it should jump forward in the query sequence equal to the number of PEs. This is explained in detail in Section 3.2.1 and illustrated in Figure 3.2. The database sequence should be delivered to the array of PEs letter by letter. When the first PE reaches the end it should jump back to the beginning of the database sequence. Some control needs to be implemented for these tasks.

4.4.1 Main control unit for query sequence input

In this section, the design considerations will be discussed and an overview of the whole implemented unit will be presented. Later, the components that build up the complete unit will be explained one by one and finally, the test of the whole unit will be discussed.

4.4.2 Design considerations and overview

One of the options to solve this control problem was to implement a so called *end-bit*. This implies there would be an additional character, representing the end of a sequence.

Four characters would represent A, C, G and T, while the last character would represent the end of the sequence, telling the PE to jump. Choosing this option means that three bits would be needed instead of two. Another would be to implement a counter for the database sequence. This counter would tell the PEs when to jump.

Both options are viable options, but implementing the first one means that an extra bit is needed to representing every character. All paths in the array of PEs would have one extra line for the third bit. Implementing a single counter only for the database sequence is, therefore, more efficient in utilizing the area of the chip.

In Figure 4.8 an overview is presented of the query sequence input unit. The inputs of the unit are *clk, rst and data_in*, which is the character needed by the PE. The data comes from a FIFO buffer. The output *data_needed* requests a new character from FIFO buffer when necessary and the other output *data_out* is connected to the PE array. The whole unit consists of four smaller components. These are the controller, an adjusted shift register, a four-input OR port [47] and, finally, the PE buffers.

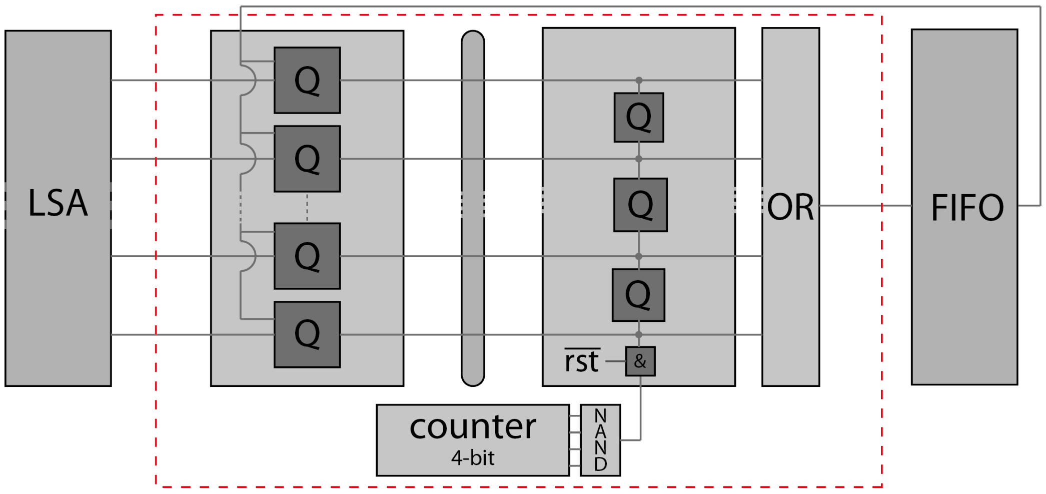

FIGURE 4.8: Block diagram of the control unit

The controller contains a counter and gives a high signal when the first PE needs to jump. It is known that the remaining PEs will need data directly after the one preceding them, so it is not needed to implement counters for every PE separately. A shift register can ensure that the signal of the counter is repeated with the number of PEs in the array. In this way, every PE gets data from the FIFO. The shift register has four outputs for the four PEs that have been implemented in the array and these function as control signals for the PE buffers to read data.

The four outputs of the shift register are also connected to the FIFO through an OR port [47] and this functions as a signal to request characters from the FIFO.

Finally, there is one adjusted buffer for each PE in the array. These buffers are connected through a bus to *data_in*, because only one buffer at a time is allowed to read a character. The PE buffers hold the data until the PE needs to jump. This is ensured by the valid bits which come from the shift register.

4.4.3 Control for jump moment

The control for the jump moment, which is depicted by a counter and a NAND in Figure 4.8 will be called *controller_hor*, consists of a counter and a NAND port [47]. The PE processes one character from the database sequence every clock cycle. The counter increments every clock cycle and is reset when it reaches the number of characters in the database sequence. Every time the counter starts from zero, a signal is given to the PEs, which tells them to jump. The NAND port [47] gives a high signal to the shift register whenever the counter starts counting from zero again.

For the proof of concept, a database sequence of 16 characters has been chosen. This means the PEs need to jump once every 16 clock cycles. For this purpose, a four-bit counter has been implemented. The counter overflows automatically and starts counting from zero again after 16 clock cycles. The four-input NAND port [47] gives a high signal whenever the counter has output “0000”.

4.4.4 Adjusted shift register

The *controller_hor* unit only gives a signal once every 16 cycles. Unfortunately, that is not enough, because four separate signals after each other are needed for the four different PEs. It is possible to implement separate counters for every PE present in the systolic array. The counter associated with the first PE will directly start counting and the counters following will all have a delay of one clock cycle with respect to the one preceding. This is not a very efficient solution, because a shift register that repeats the signal of the *controller_hor* unit three times is more efficient.

The implementation of the shift registers consists of three buffers, because, in this case, an array of four PEs has been chosen. These buffers will repeat the signal from the horizontal unit three times. The shift register unit has four outputs, which serve as control signals for the four PE buffers.

The NAND port [47], which is present in the earlier implemented horizontal control unit, also gives a high output when the reset is active. The first output of this shift register is the control signal of the first PE and decides when the PE is allowed to read data.

This output is also connected to the FIFO through an OR port [47] to request a new character. The control unit should, however, not give a signal to read in data.

Although all buffers, including the FIFO, will not function during a reset, it is logically better to take care of this problem that the shift register is giving a high output during a reset. This is done by putting an extra AND port [47] in between the input and the first output. In this way, the first output is only high when the reset is low and the NAND port [47] output is high.

4.4.5 PE buffer array

For the PEs, an array of adjusted buffers has been implemented. These PE buffers are meant to provide the PEs with the correct characters at the right moment.

In this case, an array of four has been chosen, because the systolic array consists of four PEs as well. All four PE buffers are connected through a bus to the output of the FIFO, so they should read data from this bus at the right moment and not all four at the same time. This is ensured by the signals coming from the shift register.

The inputs of the adjusted buffer are *clk, rst, cntrl* and *i*. Compared to a regular buffer it has one extra input, which is *cntrl*. This signal makes sure that the buffer does not read in a new character from the bus every clock cycle. Instead, it makes the PE buffer hold this character until the PE jumps to another character associated with that specific PE. Consequently, the PE buffer only reads in data every time the database sequence is finished, which is, in this case, every 16 cycles.

4.5 FIFO buffer

The use of a buffer is necessary when the speed of delivering and reading data is not equal. It is also necessary in cases where the speed of data delivery is not constant, because the rate of data processing is constant. Without the use of a buffer, correct data transfer is not guaranteed.

The FPGA will communicate with the computer via USB or Ethernet. In both cases the speed of the data coming from the computer is not constant. Therefore a buffer is needed.

For correct data processing, the order of the sent data is important. The first sent character should also be the first to be presented to the PE. A First in First out (FIFO) buffer is meant for this purpose. The data that enters the FIFO buffer first also leaves it first. This guarantees the correct order of receiving for the processing elements.

4.5.1 Specifications

Since the FIFO should be able to take care of a number of tasks, some specifications are needed for implementing the FIFO buffer:

- The characters can be represented by two bits and therefore the buffer should have a two-bit input and output.
- The FIFO buffer should be able to tell other units whether the buffer is full. If the buffer is full, data presented to it should be ignored.
- When other components want to write data to or read data from the FIFO buffer, they should be able to communicate this before starting to read or write.
- The written data should be placed as far as possible towards to the output of the buffer in an empty register.
- The FIFO buffer should be able to tell other units whether it is empty. This prevents other components from reading invalid data.
- When reading data, the first character as seen from the output, which is the oldest character, should be outputted and all other registers should shift their data to their neighboring registers.
- The components reading data from the FIFO buffer should be able to request data from the buffer. This way, the buffer will know when to shift data.
- To guarantee scalability, the length of the FIFO buffer should be generic in the written code.

4.5.2 Block diagram representation

For an overview, a block diagram of the FIFO buffer was made. This block diagram is presented in Figure 4.9 and the FIFO design is done with a top down approach, using the diagram for the reference.

The FIFO buffer has five input ports and three output ports:

Input ports:

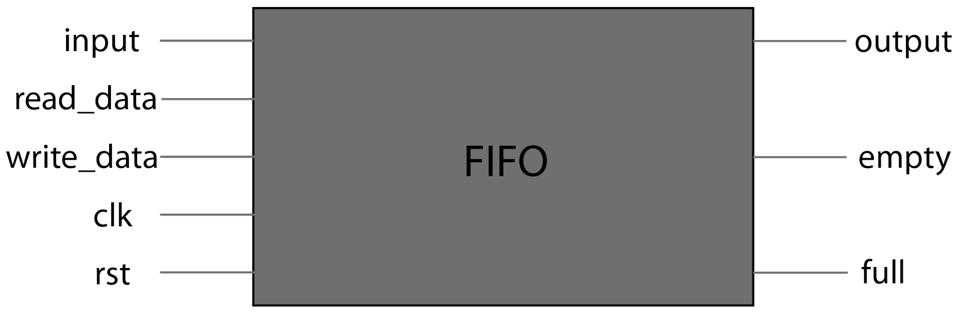

FIGURE 4.9: Block diagram of the FIFO buffer

- *clk*: The main clock which determines the frequency of the whole circuit.
- *reset*: The central reset signal. When the reset is active the buffer must be emptied and no data should be read in or read out.
- *read_data*: Signal from other components to request data from the FIFO buffer.
- *write_data*: Other components use this signal to tell the buffer that they are going to send data.
- *input* (2 bit): The characters that need to be written to the buffer are presented at this port.

Output ports:

- *output* (2 bit): The characters that need to be read from the buffer are presented at this port.
- *empty*: This signal communicates to the other units when the buffer has no data.
- *overflow*: Other units will know through this signal that the buffer is full.

4.5.3 Implementation

To implement the FIFO buffer, both a structural and a behavioral approach can be used. A structural approach would make easy adjustment of the length more difficult, because all components are already preset in such an approach. Furthermore, there are no examples in the literature of implementing FIFO buffers with a generic length from a structural approach. Therefore, a behavioral implementation was chosen to ensure that generic length would not be a problem. The behavioral description of the FIFO buffer is based on existing implementations [48].

The buffer is organized as an array of std_logic_vectors. These vectors are two bits long and serve to represent the characters of the sequence. The length of the array, which is the same as the length of the buffer, is generic.

Some logic needs to be implemented to know which was the first and which was the last character stored in the array. Otherwise, it would not be possible to distinguish valid and invalid data and delivering data in the correct order would be impossible as well. For this purpose, an idea similar to the idea of pointers in programming has been used. Two buffers combined with an adder keep track of the location of the first and last character of the array. The first buffer points to the location of the first character that needs to be presented at the output, while the second buffer points to the first possible empty place. The values contained in these two buffers point to the location of a character. From now on, they are referred to as pointers. The two extra signals *empty* and *full*, which show the status of the FIFO, are updated based on these two pointers.

Every clock cycle, the location of the first and last character might change and the status, which is represented by the two signals empty and full, can change as well. The pointee of the pointers and the value of the signals are determined in the FIFO control logic. When assigning new values, four different cases can be distinguished:

- When both *read_data* and *write_data* are low, nothing should happen. No data should be set on the output and the data presented at the input should be ignored. Since in this case the values do not change, all old values can be assigned directly to the new values.
- When trying to read data, there should first be a check whether valid data is present in the FIFO buffer. If this is the case, the pointer to the first character to be read should be adjusted. In this case, the FIFO buffer can never have the status of being full, because either the FIFO buffer is completely empty or at least one character leaves the FIFO buffer. There should also be a check whether the buffer has become empty after a read from it and, if so, the status of the FIFO must be adjusted as well. The remaining pointer to the last character stays the same.
- When writing data to the FIFO buffer, there should first be a check whether the buffer is not full already. If so, the presented data at the input should be ignored and the values of the registers should not change. However, if it is not full yet, the presented data should be accepted and there should be another check whether the buffer has become full in the meantime. If so, the value of the overflow signal should be adjusted. The pointer to the last character in the FIFO buffer should be

adjusted accordingly. The remaining pointer, which points to the first character, stays the same.

- When external components try to read and write data at the same time, the pointers to the first and last character should both be adjusted. Since reading and writing data at the same time does not change the number of data entries in the FIFO buffer, there is no need for checking whether the status of the FIFO buffer changes. Therefore, the values of the signals full and empty should stay the same.

4.6 Results

The suggested processing element (PE) consists of 185 logic elements, which is a lot less than the 384 as given by [9]. However, control overhead has not yet been included in the suggested processing element. This overhead has been found to consist of approximately 52 logic elements, leading to a total of 237 logic elements per PE.

An Artix 7 XC7A200T is assumed to be the platform that is used to implement the architecture. This platform has a total of 215360 logic elements and a maximum clock frequency of 200 MHz [49]. Since the processing elements have been designed around 5-bit adders, it is a fair assumption that the maximum clock frequency can be used without any problems. The Artix platform costs approximately 150 euros [50] when only buying the chip, and additional required components can be bought for approximately 15 euros [51]. The acquired components need to be assembled, which is assumed to cost about 15 euros. This brings us to a total cost of ± 180 euros.

By dividing the total number of logic elements on a chip by the amount used per processing element, it is shown that 908 processing elements fit on one chip. Consequently, the performance of the system can be estimated by multiplying the number of processing elements by the clock frequency. This results in a performance rating of 181 GCUPS. Dividing by the price per FPGA board leads to the performance/price ratio, as previously used in Chapter 3. The result is approximately 1.00 GCUPS/euro, which is a factor 25 improvement when compared to a current supercomputing platform, as shown in Table 3.1. It must, however, be noted that, following Moore's Law, today's FPGA solutions have about twice the amount of transistors as compared to supercomputing platforms in 2012. Therefore, this factor becomes approximately 12.

Chapter 5

Conclusions and recommendations

This chapter covers the conclusions of the project and gives recommendations for future research.

5.1 Conclusions

The goal of the project was to develop a sequence alignment algorithm implementation that is quick, cheap and highly accurate. For this purpose, we chose an algorithm and subsequently an optimal platform to accelerate it.

Although the heuristic algorithms are faster, the Smith-Waterman algorithm was chosen for acceleration. This is due to its capability to improve accuracy, scalability and implementation options. Furthermore, the current computational possibilities allow the Smith-Waterman algorithm to become feasible. The time required for this algorithm can be drastically decreased so that it outperforms other approaches. For this reason, the Smith-Waterman algorithm has been chosen to be accelerated. As for the hardware platform, FPGA was chosen. This is due to the fact that, compared to other platforms, an FPGA performs best in terms of overhead, price and scalability.

This project presented a novel approach in the literature of calculating the score matrix for the Smith-Waterman algorithm. Instead of sending the maximum value to the computer every clock cycle by comparing two digits, a buffer keeps track of the maximum value and it is sent to the computer only once. This method solves the bandwidth bottleneck by using a little bit more chip area.

The core component that performs the calculation of the Smith-Waterman algorithm, the processing element, is built according to the new differential Smith-Waterman method and is designed to function in a linear systolic array. Due to the nature of the Smith-Waterman algorithm, the array can fully parallelize the associated calculations.

The resulting implementation fits on an Artix 7 XC7A200T chip 908 times. Compared to modern FPGA supercomputing platforms, this is a twelve-fold improvement in performance/price.

5.2 Recommendations for Future Research

5.2.1 GPU implementations

With the introduction of the Cuda programming language, viability of GPU implementations of Smith-Waterman has increased a lot. It has fallen outside the scope of this project, but further investigation into GPU implementations would likely produce some interesting results. GPUs might not be as energy-efficient as FPGAs, but they can be produced on a very large scale and are thus relatively cheap. It is worth noting that a Smith-Waterman project similar to Folding@Home [52] could help medical research in the future by utilizing GPUs of many consumers who are willing to allow this. The past has shown that there is great interest in this provided that the researchers are working towards a righteous goal.

5.2.2 RVE

As discussed in Section 3.2.2, RVE can provide a speed-up at the cost of the usage of additional chip area. This can already result in a significant performance gain of 10%, but further research might increase this performance gain. Implementing RVE was too time-consuming to accomplish in this research, but must definitely be considered if more time is available.

5.2.3 Affine gap penalty

A gap penalty is used to lower the score when insertions are present. Linear gap penalty gives the same cost to every insertion. In [16] the idea of affine gap penalties was introduced. Affine gap penalty using different costs for gap opening and gap extension. It has been shown that using affine gap penalties instead linear gap penalties is a better representation of reality, because for DNA it is difficult to open a gap, but once opened, it is easy to fill that gap up. In [40] an implementation of Smith-Waterman with affine gap penalty is shown.

5.2.4 Divide and conquer

The method of divide and conquer reduces the size of the score matrix that needs to be stored. The size of the matrix that needs to stored is reduced from $n \cdot m$ to $m + n$. However, since more calculations are needed, this solution does slow down the process. Until now in the literature there are no implementations of this approach on an FPGA.

5.2.5 Differential calculation

Differential calculation is a promising new concept that has been demonstrated in this thesis. However, not much is known about its performance relative to conventional implementations. This needs to be checked by further research.

Appendix A

Work distribution

Table A.1: Work distribution - implementation

	Fatih Han Çağlayan	Roelof Willem Heij	Matthijs Geers
Implementation:			
Basic elements for Processing element	X		X
PE assembling and testing		X	
PE error detection and solving	X	X	X
Control	X		
FIFO	X		

TABLE A.2: Work distribution - thesis

	Fatih Han Çağlayan	Roelof Willem Heij	Matthijs Geers
Thesis:			
Abstract	X		
Introduction	X		
Algorithms		X	
Design Consideration			X
Differential Smith-Waterman	X		
Introduction implementation & Functional unit	X		
Basic components			X
Processing Element		X	
Control	X		
FIFO buffer	X		
Block diagrams VHDL		X	
Results			X
Conclusion	X		
Recommendations	X		X
Editing			X

The work distribution was as in Table A.1 and Table A.2. Daily meetings guaranteed good cooperation. We had regular talks with our supervisor Zaid Al-Ars.

Note: the crosses in the tables do not represent a fixed amount of workload.

Bibliography

[1] Ralf Dahm. Discovering dna: Friedrich miescher and the early years of nucleic acid research. *Human Genetics*, 122(6):565–581, 2008.

[2] Oswald T Avery, Colin M MacLeod, and Maclyn McCarty. Studies on the chemical nature of the substance inducing transformation of pneumococcal types induction of transformation by a desoxyribonucleic acid fraction isolated from pneumococcus type iii. *The Journal of experimental medicine*, 79(2):137–158, 1944.

[3] ORNL. Human genome project website, May 2013. URL http://www.ornl.gov/sci/techresources/Human_Genome/project/about.shtml.

[4] Pauline Hogeweg. Simulating the growth of cellular forms. *Simulation*, 31(3):90–96, 1978.

[5] Paulien Hogeweg. The roots of bioinformatics in theoretical biology. *PLoS computational biology*, 7(3):e1002021, 2011.

[6] Jane B. Reece and Neil A. Campbell. *Campbell biology / Jane B Reece ... [et al.]*. Pearson Australia Frenchs Forest, N.S.W, 9th ed. edition, 2012. ISBN 9781442531765.

[7] Mariana Ruiz Villarreal. Average prokaryote cell, May 2013. URL http://en.wikipedia.org/wiki/File:Average_prokaryote_cell-_en.svg.

[8] Jacques Cohen. Bioinformatics an introduction for computer scientists. *ACM Comput. Surv.*, 36(2):122–158, June 2004. ISSN 0360-0300. doi: 10.1145/1031120.1031122.

[9] Erik Vermij. Genetic sequence alignment on a supercomputing platform. Master's thesis, Delft University of Technology, 2011.

[10] Laiq Hasan and Zaid Al-Ars. An overview of hardware-based acceleration of biological sequence alignment. *Computational Biology and Applied Bioinformatics*, pages 187–202, 2011.

[11] S.B. Needleman and C.D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48(3):443–453, 1970.

[12] Temple F Smith and Michael S Waterman. Comparison of biosequences. *Advances in Applied Mathematics*, 2(4):482–489, 1981.

[13] L. Hasan, Z. Al-Ars, and S. Vassiliadis. Hardware acceleration of sequence alignment algorithms-an overview. In *Design Technology of Integrated Systems in Nanoscale Era, 2007. DTIS. International Conference on*, pages 92–97, 2007.

[14] O. Gotoh. An improved algorithm for matching biological sequences. *Journal of Molecular Biology*, 162(3):705–708, December 1982.

[15] W.R. Pearson and D.J. Lipman. Improved tools for biological sequence comparison. *Proceedings of the National Academy of Sciences of the United States of America*, 85(8):2444–2448, April 1988.

[16] S.F. Altschul and B.W. Erickson. Optimal sequence alignment using affine gap costs. *Bulletin of Mathematical Biology*, 48(5-6):603–616, 1986.

[17] T. Rognes and E. Seeberg. Six-fold speed-up of smithwaterman sequence database searches using parallel processing on common microprocessors. *Bioinformatics*, 16 (8):699–706, 2000.

[18] T. Rognes. Faster smith-waterman database searches with inter-sequence simd parallelisation. *BMC Bioinformatics*, 12(1):221, 2011.

[19] K.R. Sharma. *Bioinformatics: sequence alignment and Markov models*. McGraw-Hill, 2008. ISBN 9780071593069.

[20] RD Page. Genetree: comparing gene and species phylogenies using reconciled trees. *Bioinformatics*, 14(9):819–820, 1998.

[21] Thomas L Madden, Roman L Tatusov, and Jinghui Zhang. [9] applications of network blast server. *Methods in enzymology*, 266:131–141, 1996.

[22] Dennis A Benson, Ilene Karsch-Mizrachi, David J Lipman, James Ostell, Barbara A Rapp, and David L Wheeler. Genbank. *Nucleic acids research*, 28(1):15–18, 2000.

[23] Caroline Kovac. Computing in the age of the genome. *The Computer Journal*, 46 (6):593–597, 2003.

[24] Michael Y. Galperin. The molecular biology database collection: 2004 update. *Nucleic Acids Research*, 32(suppl 1):D3–D22, 2004. doi: 10.1093/nar/gkh143. URL http://nar.oxfordjournals.org/content/32/suppl_1/D3.abstract.

[25] A. Stivala, P.J. Stuckey, M.G. de la Banda, M. Hermenegildo, and A. Wirth. Lock-free parallel dynamic programming. *Journal of Parallel and Distributed Computing*, 70(8):839–848, 2010.

[26] T.F. Smith and M.S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147(1):195–197, 1981.

[27] E.W. Myers and W. Miller. Optimal alignments in linear space. *Computer Applications in the Biosciences*, 4:11–17, 1988.

[28] F. Zhang, X. Qiao, and Z. Liu. A parallel smith-waterman algorithm based on divide and conquer. In *Proceedings of the Fifth International Conference on Algorithms and Architectures for Parallel Processing*, ICA3PP '02, pages 162–, Washington, DC, USA, 2002. IEEE Computer Society.

[29] F. Zhang, X. Qiao, and Z. Liu. Parallel divide and conquer bio-sequence comparison based on smith-waterman algorithm. *Science in China Series F: Information Sciences*, 47(2):221–231, 2004.

[30] S.F. Altschul, W. Gish, W. Miller, E.W. Myers, and D.J. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215(3):403–410, October 1990.

[31] S.F. Altschul, T.L. Madden, A.A. Schäffer, J. Zhang, Z. Zhang, W. Miller, and D.J. Lipman. Gapped blast and psi-blast: a new generation of protein database search programs. volume 25, 1997.

[32] B. Harris, A.C. Jacob, J.M. Lancaster, J. Buhler, and R.D. Chamberlain. A banded smith-waterman fpga accelerator for mercury blastp. In *Field Programmable Logic and Applications, 2007. FPL 2007. International Conference on*, pages 765–769, 2007.

[33] A. Jacob, J. Lancaster, J. Buhler, B. Harris, and R.D. Chamberlain. Mercury blastp: Accelerating protein sequence alignment. *ACM Transactions on Reconfigurable Technology and Systems*, 1(2), June 2008.

[34] Z. Zhang, S. Schwartz, L. Wagner, and W. Miller. A greedy algorithm for aligning dna sequences. *Journal of Computational Biology*, 7(1/2):203–214, 2000.

[35] S. Manavski and G. Valle. Cuda compatible gpu cards as efficient hardware accelerators for smith-waterman sequence alignment. *BMC Bioinformatics*, 9(Suppl 2): S10, 2008.

[36] L. Hasan, M.A. Kentie, and Z. Al-Ars. Dopa: Gpu-based protein alignment using database and memory access optimizations. *BMC Research Notes*, 4(261):1–11, July 2011.

[37] E. F. de O.Sandes and A.C.M.A. de Melo. Retrieving smith-waterman alignments with optimizations for megabase biological sequences using gpu. *Parallel and Distributed Systems, IEEE Transactions on*, 24(5):1009–1021, 2013.

[38] Z. Nawaz, M. Nadeem, J. van Someren, and K.L.M. Bertels. A parallel fpga design of the smith-waterman traceback. In *Field-Programmable Technology (FPT), 2010 International Conference on*, pages 454–459, Beijing, China, December 2010.

[39] Z. Nawaz, H.E. Sumbul, and K.L.M. Bertels. Fast smith-waterman hardware implementation. In *Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, pages 1–4, Atlanta, USA, April 2010.

[40] M. Gok and C. Yilmaz. Efficient cell designs for systolic smith-waterman implementations. In *Field Programmable Logic and Applications, 2006. FPL '06. International Conference on*, pages 1–4, 2006.

[41] L. Hasan, Y.M. Khawaja, and A. Bais. A systolic array architecture for the smith-waterman algorithm with high performance cell design. In *IADIS European Conf. Data Mining*, pages 35–44, 2008.

[42] Z. Nawaz, O.S. Dragomir, T. Marconi, E.M. Panainte, K.L.M. Bertels, and S. Vassiliadis. Recursive variable expansion: A loop transformation for reconfigurable systems. In *Field-Programmable Technology, 2007. ICFPT 2007. International Conference on*, pages 301–304, Kokurakita, Japan, December 2007.

[43] L. Hasan, Z. Al-Ars, Z. Nawaz, and K.L.M. Bertels. Hardware implementation of the smith-waterman algorithm using recursive variable expansion. In *Proc. 3rd IEEE International Design and Test Workshop*, pages 135–140, Monastir, Tunisia, December 2008.

[44] L. Hasan and Z. Al-Ars. Performance comparison between linear rve and linear systolic array implementations of the smith-waterman algorithm. In *Proc. 20th Annual Workshop on Circuits, Systems and Signal Processing*, pages 451–456, Veldhoven, The Netherlands, November 2009.

[45] L. Hasan and Z. Al-Ars. An efficient and high performance linear recursive variable expansion implementation of the smith-waterman algorithm. In *Proc. 31st Annual International Conference of the IEEE Engineering in Medicine and Biology Society*, pages 3845–3848, Minneapolis, USA, 2009.

[46] B. Strengholt and M. Brobbel. Acceleration of the smith-waterman algorithm for dna sequence alignment using an fpga platform, June 2013.

[47] Wikipedia. Logic gate - wikipedia, the free encyclopedia, May 2013. URL http://en.wikipedia.org/wiki/Logic_gate.

[48] Pong P Chu. *FPGA prototyping by VHDL examples: Xilinx Spartan-3 version*. Wiley-Interscience, 2008.

[49] Inc. Xilinx. 7 series fpgas. Technical report, San Jose, USA, 2013.

[50] Digi-Key Corporation. Price list. URL http://www.digikey.com.

[51] Premier Farnell plc. Price list. URL http://www.farnell.com.

[52] Stanford University. Folding@home. URL http://folding.stanford.edu/.