An FPGA acceleration method for DNA sequence short-read alignment

Vytautas Mizgiris

4th Year Project Report
Computer Science
School of Informatics
University of Edinburgh

2018

Abstract

Nowadays, improvements in DNA sequencing and analysis are reaching new heights. With the introduction of High-Throughput Sequencing - technologies capable of efficiently processing large amounts of genome data - and improvements in hardware - development of billion transistor architectures and an increasing interest in parallelism - new and optimized ways are sought to speed up sequence alignment process. Previous research in the area of DNA alignment has mainly been focused on parallel execution using accelerating hardware (GPUs and FPGAs), and great improvements in performance and cost have been observed. I have explored the sequence alignment optimization problem further and developed a hardware architecture for the Smith-Waterman dynamic programming algorithm, which I deployed on the Zynq-7010 FPGA to drive the alignment process. The alignment scores are calculated on the FPGA, streamed to the on-chip processor running Linux (host), and processed by a C application to obtain the score matrix. The obtained score matrix can be used to find optimal alignments; furthermore, I have proposed an improved architecture design for optimal alignment traceback in hardware.

Acknowledgements

I would like to thank my supervisor for each bit of help and guidance, as well as all of my family and loved ones for the support during my last university year and the development of this project.

Table of Contents

Chapter 1

Introduction

Today's technology industry is driven by huge amounts of data and the need to process it. Hardware and software optimizations are constantly being rolled out to source and deal with data at increasing speeds. In the following chapter, I will present some modern parallel processing approaches and how they can help with the problem of sequence alignment. I will also draw out a plan for the project goals, report structure and list contributions.

1.1 General purpose hardware

When it comes to data intensive, highly parallelizable tasks, the core areas to look into boil down to the lack of optimization in general purpose hardware. Problems involving computation for millions of objects independently (with no data dependencies), such as audio and video processing and matrix calculation, can utilize graphics processors (mentioned later) much better than scalar processors, yet the former can be expensive and not very scalable.

Another matter is the storage requirements for tasks that involve dealing with processing and / or producing quadratic amounts of data. High speed data transfers along with storage are crucial elements to streaming data applications. Reducing the amount of data with as little impact on the output as possible is one of the goals when dealing with storage constraints.

Figure 1.1 displays *Moore's Law*, which states that the number of transistors in processors roughly doubles each year (later revised to each two years), a trend that has been followed closely throughout the past few decades. Processors that equip transistors that decrease in size and tightly packed transistor dies also introduce longer delays for data to travel across the processor and higher heat-dissipation, creating the need for development of highly-optimized parallel architectures by having multiple processing elements instead of a single unit.

One of the better known ways of dealing with parallelism in computation of arbitrary parallelizable problems is by employing processors that are designed to

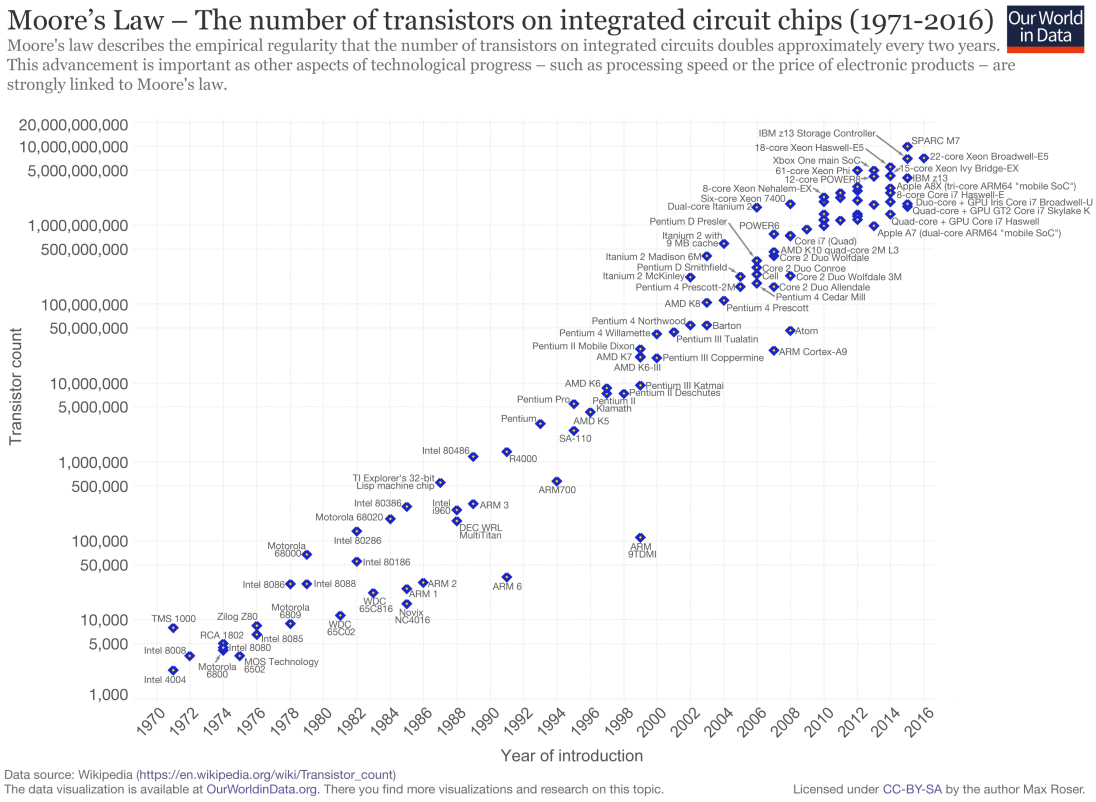

Figure 1.1: The processor transistor increase trend behind Moore's law. (Image attributed under CC-BY-SA to author Max Roser)

work with multiple data objects at once. Unsurprisingly, graphics processing units (GPUs) are a popular choice for many modern image, video, neural network processing problems. In addition, hardware vendors have released extensions, such as intel SSE, that are applied to general purpose CPUs, letting them operate partly as SIMD (single-instruction multiple-data) processors, capable of hardware-level parallelization of tasks with limited to no data dependencies.

However, all the architectures mentioned above are considered *general purpose*: hence, they will likely not scale well for bigger problems and may suffer from overhead occurring from communication in *embarrassingly parallel* problems. A processor that employs a task-specific architecture has potential to outperform a GPU or CPU used for the same task, and research has indeed indicated that this is the case [1, 2].

1.2 Reconfigurable architectures

The idea of reconfigurable architectures has been around in the scientific community for quite a while, along with the introduction and development of general purpose processors. Xilinx, the vendor of the Zynq-7010 processor, developed their first reconfigurable processor in 1985. It was also the first product to suit commercial demand because of its relatively low price and mass production.

A *field-programmable gate array* is appealing for many data-intensive tasks that involve fine-grained parallel computation, and allows the architect to write implicitly parallel code. A reconfigurable architecture that can utilize hardware-level parallelism has an edge in the parallel era, and is cost-effective for certain tasks when compared with standard multithreaded or superscalar processors. Regrettably, due to the lack of open-sourced development effort and high cost of the actual development and testing process, such and similar architectures have not faced a major take-off like other types of processors.

On a side note, reconfigurable processors were, in fact, considered in the research for future mainstream architectures before multi-core computing took place. Around 20 years ago, speculations were made on what would replace the existing standard single-core processors, and discussed in the *IEEE* Computer magazine [3]. Many ideas were then presented, together with some that accurately predicted the trend, and others that eventually were left out as a field of research only. One such guess was the idea of a *raw processor* [4], a compiler reconfigurable processor that resembled a user reconfigurable processor in terms of the underlying structure: a die consisting of a number of tiles that were supposed to be programmed by the compiler. A high amount of utilization was expected to be achieved as each tile on the processor would be responsible for a separate (type of) task, resulting in significant performance gains.

1.3 Aim and structure of project

In my project I will propose a way of using an FPGA reconfigurable processor for the problem of sequence matching. The field I will be exploring is DNA alignment, most commonly represented as a matrix score computation problem, and very commonly used in the field of computational biology.

I will cover the background of DNA sequencing and the string matching problem before diving into methods of sequence alignment and its optimization. The problem of DNA alignment is discussed broadly in a range of articles published in the recent past. It can be a highly data-independent task, well-suited to processors such as FPGAs and ASICs. Because of the increasing size of databases of DNA sequences and rapid advances in technology, there is a demand for highly scalable and fast solutions to be researched, implemented and verified to speed up DNA analysis and carry on with other tasks in computational biology.

The aim of my project is to critically evaluate research done in the field of DNA alignment architectures as well as design, implement and benchmark a suitably parallel approach for DNA matching in the fabric of an FPGA processor, coupled with interfaces to communicate with an embedded Linux system. The basis of the implementation is to choose a suitably parallel algorithm and, with reference to architectures developed and written about in literature, design an alignment accelerator. The final task is to adapt it to the hardware available and measure its performance, comparing it to a software-only solution. The closing chapters will consist of results, inferences and *proposals* to future systems, as well as how the work done can be used for future research.

1.3.1 Contributions

The following is a list of contributions that I have made to complete the project:

- Critically evaluated past research on the alignment of DNA sequences using common algorithms, identified bottlenecks and optimizations;
- Designed and implemented a parallel architecture on an FPGA processor for DNA sequence short-read alignment using the Smith-Waterman algorithm, with a parallel streaming sequence matching in hardware and score matrix reconstruction application in C;
- Discussed and proposed improvements and alternative approaches to build upon the implementation.

Chapter 2

Background

The DNA, or deoxyribonucleic acid, is an important part of any living organism that essentially defines the way it grows and functions. It is code that is supplied to create genes in cells, and to provide vital information on how the cell should reproduce. Due to DNA's uniqueness, it is one of the most important defining features of every human being. It is very commonly studied in the fields of computational biology, or bioinformatics, where DNA analysis is done by examining the cells and, most often, finding patterns and / or aligning the extracted code to another set of DNA. A few fields that such analysis helps significantly are: genetic disease detection, paternity tests and genealogy tree building, proving the evolution of organisms and determining the origin and history of species [5], identifying criminals by observing the DNA from fingerprints left at a crime scene and comparing it to a reference sequence. All of the latter involve DNA pattern matching, for which an efficient solution is needed, due to it being expensive both in time and space. In the following chapter, I explore this topic in more detail and present the background of the project.

2.1 DNA sequencing

The first and foremost part of DNA analysis is sequencing. The goal of DNA sequencing is to find, within the molecule, the exact ordering of the four nucleotides - **a**denine, **c**ytosine, **g**uanine and **t**hymine. Using state-of-art sequencer machines, short parts of the sequence are obtained, called *short-reads* - sequences of length anywhere from roughly 25 to 300 base pairs[1]. These obtained sequences must overlap a certain amount of times to make sure the ordering is accurate, to then build the final sequence. Alternatively, short-reads are used to align to a reference sequence to obtain information about the reference sequence. A modern approach to DNA sequencing is illustrated in Figure 2.1.

[1]Due to the fact that nucleotides form fixed pairs in DNA strands, in a textual representation they are called base pairs. A DNA base pair is equivalent to a single character in terms of sequence alignment; base, base pair and character will therefore be used interchangeably.

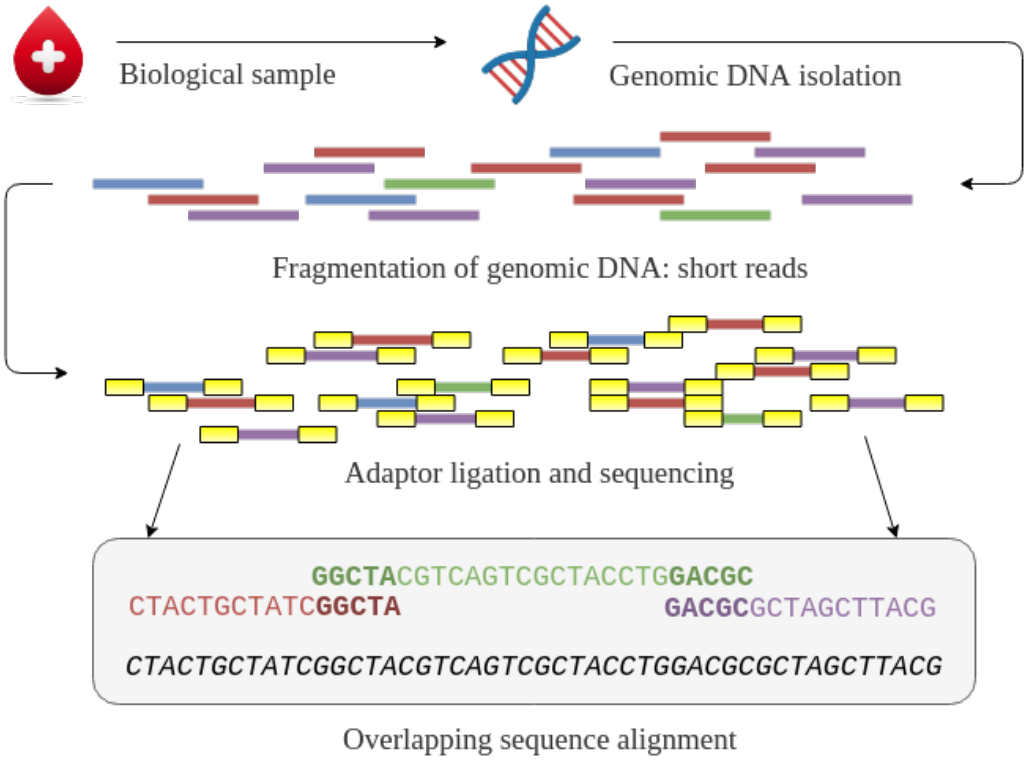

Figure 2.1: Next Generation Sequencing DNA sequencing process, during which short reads are obtained and reconstructed into a full nucleic sequence. The illustration of process is based on an image by Ranjan et al. [6]

2.1.1 Methods

One of the first methods, Sanger sequencing [7], introduced in 1977, has been one of the core approaches for DNA sequencing for at least 30 years after. Despite its high accuracy and detailed outcome, it involves expensive machinery and is only used in specialized genome analysis centres, and is unsuitable for a more widespread sequencing. The situation has changed quite recently, when Next Generation Sequencing (increasingly known as High-Throughput Sequencing) was introduced, described in more detail in the following articles [8, 9]. NGS is a breakthrough of variety of methods that were proposed and invented in the last decade, making DNA sequencing a much faster and cost-efficient procedure. Advances in technology allowed for a parallel DNA sequencing for billions of short-reads. Different technologies with varying machinery are used today[2], depending on the complexity of the genome and outcome requirements, replacing the need for dedicated sequencing centres. The Sanger method is still being used today as a reference alignment procedure, to sequence longer reads as well as ensure the accuracy of an NGS method.

2.1.2 Sequence database

One of the indirect outcomes of NGS is the Sequence Read Archive, or SRA. It is a database, published by the National Center for Biotechnology Information, US (NCBI), which, as of 2014, contains a massive amount (roughly 100 terabytes) of reads and full sequences obtained using NGS technology. The database provides tools to search and download a variety of already sequenced data, upload newly processed reads using high speed (up to 400 Mbps) links and supports many major sequencing technology formats [10, 11].

In terms of the end of goals of the project, the database is a great tool to obtain and use short-reads to benchmark DNA alignment architectures. I will elaborate more on this in the implementation chapter.

2.2 Introduction to biological sequence alignment

Sequence alignment can be categorized in two core techniques: *global* and *local* alignment procedures. The methods for defining similarity in the past were firstly using global alignment. A global alignment algorithm uses the whole space of two DNA sequences to find similarity, that is, every character from sentence A is aligned to every character from sentence B. Clearly, the time required to align each character from one sequence to another grows exponentially as the length of search sequences increases. Such procedure is obviously inefficient on its own,

[2] A more in depth coverage can be found at https://www.illumina.com/techniques/sequencing/dna-sequencing.html, one of today's common NGS technology manufacturers' website.

$$\begin{array}{cccccccccccc} C & T & C & T & A & L & I & L & L & A & V & A & V \\ | & \cdot & \cdot & | & | & | & \cdot & | & | & | & \cdot & | & | \\ C & - & - & T & A & L & - & L & L & A & - & A & V \end{array}$$

Figure 2.2: Global alignment result. Characters are matched by considering the whole length of both sequences.

$$\begin{array}{cccccccccccccc} C & T & C & T & A & L & I & L & L & - & A & V & A & V \\ \cdot & \cdot & | & | & | & | & \cdot & | & | & | & | & | & \cdot & \cdot \\ - & - & C & T & A & L & - & L & L & A & A & V & - & - \end{array}$$

Figure 2.3: Local alignment results. Sequences are aligned by considering optimal local alignment scores.

so optimizations are desired. Global alignment works well for problems where a high rate of similarity between two similar length sequences is expected.

Newer algorithms employ local alignment, a technique that, when it first appeared, relied on the fact that for some types of protein sequences separate regions were found to be similar instead of the whole sequences [12]. Algorithms that use local alignment reduce the search space significantly and increase flexibility. Local alignment usually results in improved performance, but can also be less sensitive to more complex patterns, and is more suited to problems where, for length α of a base sequence and length β of the sequence being searched for, $\alpha \gg \beta$ [13].

Example: consider two sequences, $\{x = CTCTALILLAVAV, y = CTALLLAAV\}$, with a character set $\Delta = \{A, C, I, L, T, V\}$. Then, the two types of alignment are visually represented in Figures 2.2 and 2.3, with sequence x being the one y is aligned to.

2.3 Alignment algorithms

A number of techniques have been explored in the past to optimize the problem of string search and pattern matching. These algorithms are used in areas such as machine learning, computational linguistics or biological sequence analysis, and have been developed taking in account a variety of expected outcomes.

A good example is the Knuth-Morris-Pratt string prefix-suffix search algorithm. The algorithm tries to optimize the search space by taking in account previous matches and skipping an amount of trivial comparisons [14], exposing linear runtime and space complexity, making it an efficient solution in the general case. It is one of the earliest and widely used optimizations of a nave search solution.

The algorithm is general enough to be used for DNA alignment, since the core functionality is relevant to DNA analysis and will produce a final character comparison score which can be used to determine sequence similarity. In addition, recent research has also found that the algorithm can outperform more sophisticated approaches in cases where sequences are suspected to be highly similar [15]. On the other hand, newer, highly optimized and more accurate problem-specific algorithms have been described and implemented, as shown in [16].

Next, I will present the most widely-used procedures that are used to align DNA

sequences. Many approaches commonly apply the concept of *dynamic programming* rather than use general-case string matching algorithms and their derivatives.

2.3.1 Dynamic programming

Dynamic programming is a term that refers to an optimization of a larger problem by splitting it into several sub-problems. For sequence matching, dynamic programming is defined more precisely as: building a score matrix for two given sequences, calculating the score for every point in the matrix given the previous alignment sub-problem score, and tracing back to locate the global optimum alignment by following along the highest neighbouring scores at each step. The recursive score formula in its bare form is defined as the following:

$$S(i,j) = max \begin{cases} S(i-1,j-1) + \sigma(x_i, y_j) & match/mismatch \\ S(i-1,j) + \gamma & deletion \\ S(i,j-1) + \gamma & insertion \end{cases} \tag{2.1}$$

where i and j are the respective sub-sequences, σ is the alignment score and γ is the gap penalty. The procedure is described in great detail along with discussion by S. R. Eddy [17].

In any dynamic programming approach, a gap penalty function is used to favour matches with higher similarity in terms of distance between matching subsequences. Linear and affine are the two most commonly-used gap penalty functions. The former uses a constant negative factor that, in effect, decreases the match score as the gap between two matching regions increases, and is defined as:

$$G = \gamma \times L \tag{2.2}$$

where γ is the fixed insertion / deletion penalty and L is the length of the gap so far. The affine gap penalty adds a constant gap opening penalty that can essentially determine how close the matched regions can reside:

$$G = C + \gamma \times L \tag{2.3}$$

where C is the fixed gap opening penalty, γ is the fixed insertion / deletion penalty and L is the length of the gap so far. The latter is more commonly used due to added flexibility and to improve accuracy where the structure of a sequence is known, e.g. a lower gap opening penalty will be useful for sequences that have close-matching regions and efficiently discard matches with wide gaps.

2.3.1.1 Needleman-Wunsch

One of original methods that uses dynamic programming and is the basis for many later approaches is the Needleman-Wunsch algorithm. It was initially proposed in

1970, with an application to computational biology in mind, to align amino acid sequences of proteins [12]. A few years later, more efficient alternatives were suggested, including improvements in runtime performance from cubic to quadratic time [18], space optimization [19] and with application to DNA matching and fields other than bioinformatics [20]. The NW algorithm is also the core method for the later development of the Smith-Waterman algorithm, which employs, in some cases, the more favourable local alignment technique. Nevertheless, the NW algorithm is still used for global alignment tasks in its original form. The procedure can be found embedded in widely-used tools for computation biology including NCBI BLAST [21], EMBOSS, MathWorks Matlab products.

2.3.1.2 Smith-Waterman

The Smith-Waterman algorithm was proposed as an alternative to the Needleman-Wunsch method, focusing on the more often pronounced need to calculate local alignment scores in larger sequences than the less-used global alignment [22]. It builds upon the dynamic programming approach, introducing a new concept: instead of negative single alignment scores, a score of 0 is assigned. This way, optimal local alignments can be found by tracing back from the position with the highest score in the substitution matrix, to the first 0 that appears in the way. Any subsequent alignments are found by tracing back from the corresponding second-highest score. The modified dynamic programming algorithm for the Smith-Waterman alignment algorithm is as follows:

$$S(i,j) = max \begin{cases} S(i-1,j-1) + \sigma(x_i, y_j) & match/mismatch \\ S(i-1,j) + \gamma & deletion \\ S(i,j-1) + \gamma & insertion \\ 0 \end{cases} \tag{2.4}$$

given that $S(0,j) = 0$ and $S(i,0) = 0$. If a fixed (linear) gap penalty is defined, the algorithm runs in quadratic time.

Example: consider two DNA sequences, $\{x = ACACACTA, y = AGCACACA\}$, match score $+2$, mismatch penalty -1, gap penalty -1. Suppose a few best optimal alignments are found using the Smith-Waterman dynamic programming equation, the results are then shown in Figure 2.4.

2.3.2 Other algorithms

Among other algorithms, commonly used for DNA, RNA or protein sequence matching, BLAST is a notable competitor to the dynamic programming based algorithms. BLAST, name of which stands for **B**asic **L**ocal **A**lignment **S**earch **T**ool [21], uses a heuristic approach to find a close-to-optimal matches between two sequences. It consists of a few stages: *seeding*, during which all words (subsequences) of some fixed length are located in the base sequence and aligned to

$$S = \left\{ \begin{array}{ccccccccccc} & - & A & C & A & C & A & C & T & A \\ - & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ A & 0 & 2 & 1 & 2 & 1 & 2 & 1 & 0 & 2 \\ G & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 \\ C & 0 & 0 & 3 & 2 & 3 & 2 & 3 & 2 & 1 \\ A & 0 & 2 & 2 & 5 & 4 & 5 & 4 & 3 & 4 \\ C & 0 & 1 & 4 & 4 & 7 & 6 & 7 & 6 & 5 \\ A & 0 & 2 & 3 & 6 & 6 & 9 & 8 & 7 & 8 \\ C & 0 & 1 & 4 & 5 & 8 & 8 & 11 & 10 & 9 \\ A & 0 & 2 & 3 & 6 & 7 & 10 & 10 & 10 & 12 \end{array} \right\}$$

$$\begin{array}{c} A\ -\ C\ A\ C\ A\ C\ T\ A \\ |\ \cdot\ |\ |\ |\ |\ |\ \cdot\ | \\ A\ G\ C\ A\ C\ A\ C\ -\ A \end{array}$$

$$\begin{array}{c} -\ -\ -\ A\ C\ A\ C\ A\ C\ T\ A \\ \cdot\ \cdot\ \cdot\ |\ |\ |\ |\ |\ \cdot\ \cdot\ \cdot \\ A\ G\ C\ A\ C\ A\ C\ A\ -\ -\ - \end{array}$$

Figure 2.4: Example best (red) an arbitrary n-th best (blue) local alignment using Smith-Waterman algorithm. The alignment is found by following coloured entries bottom-up.

the hit sequence(-s) by using a scoring matrix, *extension* where the sub-sequences are extended to both sides to find the rest of the alignments. The algorithm relies on a specified threshold that determines how high a subsequence match needs to score to be included in the results.

Because of its efficient nature, it is common to use the algorithm for large databases of sequences as well as for single alignments that do not need the most optimal match. The runtime of the algorithm depends on the length of the sequences being compared, the subsequence length at the seeding stage, and the amount of sequences being compared, therefore it is hard to estimate a common big-O runtime for the algorithm. It is estimated by some to be linear [23], which is big improvement over dynamic programming approaches.

The algorithm was developed with the trade-off between performance and optimal alignments in mind - it is therefore a very fast method to run on a single-threaded system. Parallel software implementations are available using threading libraries in common operating systems, such as the open-source `http://www.mpiblast.org/`, and the whole or parts of the algorithm can be parallelized in hardware [24].

2.3.3 Overview

Despite the speed of the BLAST algorithm, due to its inexact nature and lack of straightforwardness to develop a parallel hardware architecture for, I will be focusing on dynamic programming methods for the scope of my project. The latter have a very suitable form for parallelization and have potential to match BLAST in terms of performance, while also producing globally optimal alignments. Due to NGS, cheaper sequencing methods are becoming employed, some of which trade off performance for accuracy (such as 97.5% accuracy of an Ion Torrent method versus traditional, expensive Sanger-based CE sequencing, yielding 99.99% [25, 26]). Because of this, additional small inaccuracies can contribute

to a worse analysis of a DNA sequence.

Dynamic programming problems are commonly solved using a systolic array architecture, where previously-computed values are used for subsequent computation, filling a matrix-like structure. Using such a procedure, a number of values can be calculated in parallel by employing suitable software and hardware techniques. In the next section, I will present hardware that can be efficiently used to implement a DNA alignment core using one such approach.

2.4 Introduction to FPGA

A field programmable gate array, which I will refer to as an "FPGA" in this paper, is a processor that employs a user re-configurable architecture. In most cases, it is a separate integrated circuit or a part of a chip containing other types of processors. The chip is usually attached to a board that has various types of input and output ports, used for communicating with a host interface to transfer data and debug. A modern FPGA will likely also contain a pre-configured multi-purpose processor on the side, some amount of memory designed as flip-flops or block memory, and a variety of communication buses that connect the FPGA fabric with the processor, inputs and outputs and in-between.

```
23 module half_adder(
24     input A,
25     input B,
26     output S,
27     output Cout
28     )
29
30     assign S = A ^ B;
31     assign Cout = A & B;
32
33 endmodule
```

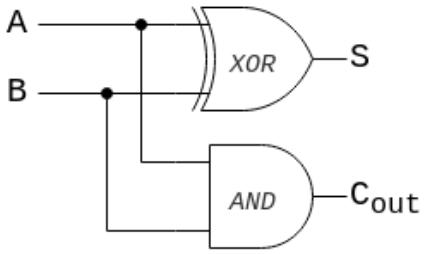

Figure 2.5: Example Verilog translation of a half adder logic circuit. The `assign` statements represent the assignment of logic to a particular output variable. In this case, `S` represents the sum bit of A and B using an `XOR` gate, whereas `Cout` calculates the carry bit using an `AND` gate.

An FPGA is configured using a hardware design language, such as Verilog (see Figure 2.5), SystemVerilog or VHDL. The code defines a gate-level structure and the preparation of the binary code for the architecture consists of three or four stages: simulation (debugging), synthesis, implementation and bitstream generation. The first stage involves a simulated run using test HDL code to identify the processor behaviour at the granularity of a single clock cycle. Once the developer is happy with the results, code synthesis lays out the code as a logic circuit, and implementation is used to route the circuit onto the specific FPGA's architecture core. The bitstream (binary code) can then be generated and loaded onto the FPGA using suitable development board interfaces.

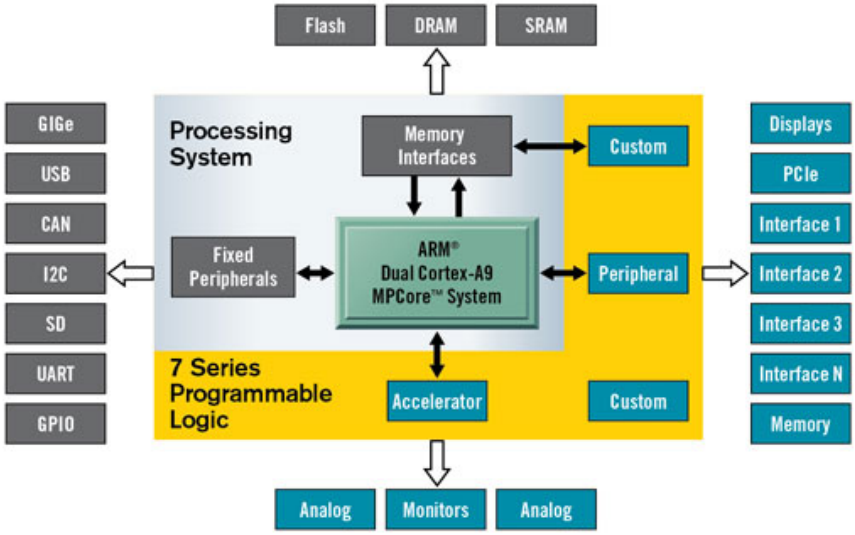

Figure 2.6: An overview of the components inside the Xilinx Zynq-7000 All Programmable System on a Chip [27].

The reconfigurable nature of an FPGA processor allows for the development of some very power-efficient, high-speed architectures for specific tasks. An architecture deployed on an FPGA will be inherently parallel provided non-blocking assignments are used, i.e. values will be passed between multiple modules and registers at once, greatly reducing the amount of time it takes to process data.

An example of a modern FPGA is the Xilinx Zynq-7000 processor (see Figure 2.6). Integrated with a mobile ARM Cortex-A9 dual-core processor and fitted with high-speed buses, it provides high levels of flexibility and efficiency.

2.4.1 Fields of application

In recent decades, there has been an increased interest in taking advantage of the growing variety of computational tools and resources to increase performance and reduce power consumption by exploiting task parallelism in reconfigurable devices. Below are a few today's examples of exploitation of efficiency using an FPGA:

- **Deep packet inspection**, a method to determine if a packet sent over the network contains potentially malicious, spam or otherwise unwanted data, is increasingly important in internet traffic control and cyber-security. A number of solutions using FPGA processors are tested and proposed as a cheaper, efficient alternative [28, 29, 30].
- **Bitcoin**, a cryptocurrency, had recently risen in popularity and use, and so methods to process the transactions efficiently were sought. Initially, traditional computational resources were used, such as general purpose and graphics processors. As the popularity and demand for bitcoin increased, quicker and less power-demanding solutions were researched, and

that was where ASICs (application-specific integrated circuit chips) and FPGAs showed an advantage [31].

- Focus of the project: a prolonged interest in using FPGA systems in **bioinformatics**, more commonly **DNA matching** applications, can be observed. The field is described in more detail in the next section.
- FPGAs are also used in digital signal processing, computer vision, automobile, aviation and military technology and others, including **ASIC prototyping**, which is an cost-efficient step towards a bigger scale implementation of a design [32, 33].

2.5 FPGAs in computational biology

The use of FPGAs in computational biology and the design techniques to employ them are comprehensively explored in the following article by Herbordt et al. [34]. There are a number of drawbacks that software-only solutions present that contribute to under-utilization of a general purpose processor and ultimately limit the potential performance. For example, a computation process that involves calculating entries in a matrix will involve a lot of memory accesses, which in turn will limit the speed in which the data can be processed at once. In addition, limited software support for fine-grained bit-level data incurs wasted bandwidth between memory and the CPU. A configurable FPGA architecture can solve both problems. There are many more advantages of using an FPGA for tasks of computational biology, many of which involve parallelism and scaling as the core carry-forward; I will therefore focus on a few that expose the best way such a processor can be exploited for the task.

A brief and concise overview of the way *dynamic programming* algorithms are implemented in FPGAs is given by Boukerche et al. [35]. Given the fact that an FPGA suits the task of a parallel implementation of such algorithms very well, I will present a few points to indicate how an FPGA architecture solves the problems discussed above:

- The most common approaches use a linear systolic array: a tightly interconnected mesh of processing modules. At each point in time, they process values coming in from the neighbouring modules together with saved outputs. The advantage of an FPGA employing such array is the ability to operate all processing elements in parallel, eliminating the time spent idle for each of the elements in an analogous software implementation. A more detailed overview of a systolic array follows.
- In a clock-driven implementation, the storage requirements for intermediate products is greatly reduced: many values travel from each of the processing elements for the following computation during a single clock cycle, eliminating the need to store intermediate values which is required for a sequential implementation.

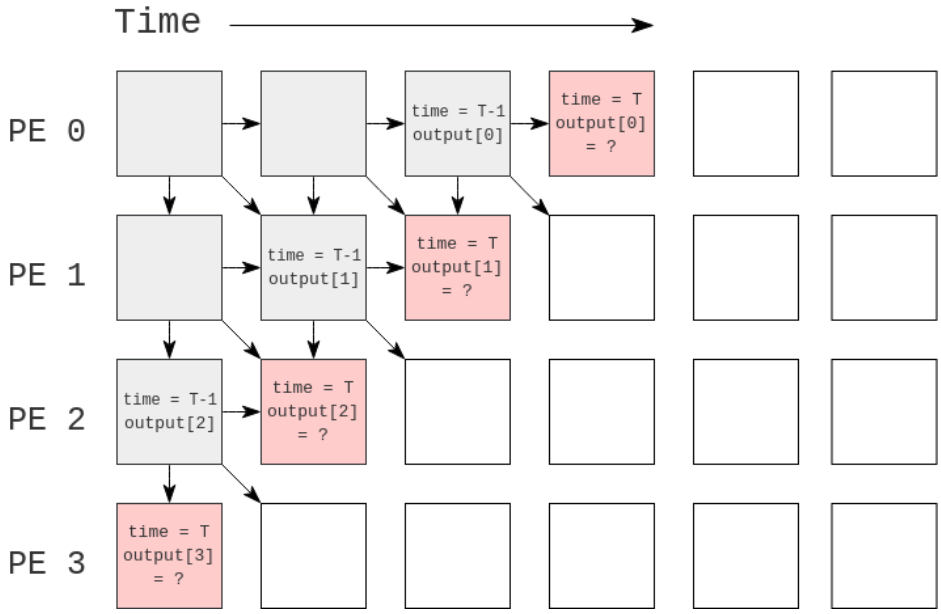

Figure 2.7: Systolic array in a dynamic programming parallel execution setting. The processing elements all operate in parallel once the pipeline is filled, calculating results for scores[0] to scores[3]. Results for the red entries are being calculated at the current time T, using values from previous time T-1 and T-2, which are saved in each cycle to build a resulting score matrix.

- In the case of DNA alignment, DNA bases can be coded in as little as two bit values, whereas depending on the constraints for the sequence lengths, the widths for scores and intermediate values can be specified, reducing the overhead incurred from wasted bandwidth in the FPGA fabric as well as between FPGA and host communication.

2.5.1 Systolic array architecture

For dynamic programming-based sequence matching implementations, a systolic array (synchronous or asynchronous, monolithic processing modules) or a wave-front processor (asynchronous, independent) is used to parallelize the biggest chunk of DNA alignment work. The procedure, in the setting of FPGA architecture design, is depicted in Figure 2.7.

Each processing element (further on referred to as PE) is attached to a space in several input and output data pipes, such that PE_0 gets initial values and emits results to `output[0]`, PE_1 gets values from `output[0]` and emits results to `output[1]`, and so forth. For values that need to be used by each PE (constants, such as DNA sequence characters), shifting a value into the first place of an input pipe results in subsequent values begin pushed in and propagating into the pipeline - that results in second PE processing the value used by first PE during previous cycle.

A synchronous systolic array processes values on the rising edge of the clock, implying fast processing and tight synchronization. The processing of input values, on the other hand, needs to fit into a single clock cycle, which can require decreasing the clock frequency. With an asynchronous architecture, values are propagated on input change and results sampled on the rising edge of the clock, which implies no constraints on the clock speed provided the computation is split into stages. The downside - an asynchronous approach usually requires considerably more register memory on the FPGA, since partial results need to be stored before continuing.

2.6 Related work

A significant amount of research has been done in the field of hardware acceleration of pattern matching. Many of the previously discussed algorithms have been implemented in some way in FPGAs with varying resources and modifications, and described together with quantitative findings in literature. The majority of dynamic programming approaches are straightforward to parallelize - using systolic array design - on a reconfigurable system such as an FPGA. In addition, many other approaches were taken to combine elements of different algorithms to better utilize hardware resources and come up with a solution that yield a better performance / optimal alignment ratio, such as [36].

Several methods based on the dynamic programming procedure were described in the past and some of those have influenced the way I approached the sequence matching problem. I will be going into detail discussing the optimizations proposed, advantages and drawbacks of each approach.

2.6.1 Method for short-read mapping

A method for short-read alignments was described by Shah et al. [37], presented as "a step to personal genomics". The core idea of the approach is to store the comparison sequences in dedicated memory, fetch and process them in parts as computation progresses, with several passes of comparison. The scores are stored in the block RAM of the device and sent to the host computer at each comparison stage via Ethernet link. All of the logic is implemented on a device of low cost and limited resources (Spartan-3 FPGA).

The architecture uses a systolic array of processing modules as the alignment core. The novelty method proposed is a so called next-pass and full-alignment algorithm: by exploiting the increasingly larger resources of block RAM on an FPGA, a filled score matrix is transferred to the host only after multiple passes of characters are made through the processing elements. Most modern day FPGAs, including Digilent ZYBO, have a considerable amount of block RAM, which can be efficiently utilized. The core can also essentially align any size sequences

because of the next-pass strategy, whereas approaches that store the score matrix on-board suffer from very limited sequence length, due to lack of resources.

Bottlenecks for the approach, as pointed out in the conclusion section, are the time required to transfer scores to the host and space consumed for the intermediate score matrices.

2.6.2 Optimized method for alignment score only

Significant savings in performance and space are achieved by greatly reducing the complexity of the processing cell for calculating the score only. The approach is described by Khan et al., extending upon previous work by Yu et al. [38, 39]. The article describes a method based on a simplified Smith-Waterman algorithm for comparing two sequences, yielding a final alignment score. Findings by Lipton et al. [40] are used to minimize the amount of data stored in each processing element and eliminate the need for a score matrix, which I find is the biggest advantage of the described method. As opposed to the approach discussed before, FIFO queues coupled with shift registers are in place to propagate the DNA sequences into the PE pipeline.

At any point in the alignment procedure, the space complexity is equivalent to the length of the base sequence times the amount of storage a single PE uses. It is beneficial to have a system with very little requirements for space, although it is important to note that the alignment costs are reduced by constraining the problem to finding only the comparison score.

2.6.3 DIALIGN, wavefront processor

The DIALIGN algorithm is a variant of dynamic programming based algorithms. It uses a similar method to the Smith-Waterman algorithm, but only aligns regions that do not contain gaps, and therefore is better suited for many regions of high similarity as opposed to SW, which is more useful for a single one. The authors for a hardware DIALIGN accelerator [35] have summarised the previous work and reported the speedup for a number of different approaches, and proposed a way of aligning sequences using a wavefront processor, due to peculiarities of the algorithm introducing a non-uniform distribution of workload across some PEs. Speedup measures of between 126 and 383 have been achieved using two accelerators, DIALIGN-Score and DIALIGN-Alignment, which is a very impressive improvement over software-only solutions.

2.6.4 Partitioned problem space

To reduce space complexity and increase the amount of parallelism, Hu et al. [41] suggested partitioning the problem space into blocks, and operating on many distinct sub-sequence comparisons in parallel, based on the assumption that dynamic

programming approach of prefix-suffix matching is mostly used for sequences with high degree of similarity. The latter implies that a global optimal alignment is not guaranteed to find, but results in a greatly decreased computational space (only calculating scores near the main diagonal of the score matrix). The approach was simulated, and a 98.8% accuracy was reported when compared to reference alignment results.

2.6.5 Flexible alignment parameter setting

A "highly parametrized" sequence alignment architecture has been developed by Benkrid et al. [42], that uses Handel-C high-level programming language to generate HDL code for a sequence alignment architecture. This way the authors give the flexibility of setting a variety of parameters for the process, which is useful both for testing - feasibility of such architecture for real-world applications - and end users, who are interested in comparison of sequences with diverse outcomes. A multitude of benchmarks were carried out, including a test resulting in an improvement of $\tilde{2}$-3 times the speed of a highly-parallel task-specific SIMD processor. The simulated architecture has certainly showed a significant improvement over more traditional hardware, but it is important to note that generating hardware code using high-level constructs may take away some of the fine-grained parallelism that can only be achieved by writing in pure HDL.

Chapter 3

Resources and set up

In the following section, I will cover the setup that has been executed before the research and implementation. A variety of tools and code-bases are increasingly available online as either open-source projects or free-to-use in an academic context software, to accelerate FPGA-specific development. FPGAs, as processing units, are found across various platforms and configurations - FPGA logic design is used to solve a number of mathematical problems in many scientific fields, while there are also a number of ways to describe that logic. Therefore, it is important to have the possibility to focus on a selected field of research without having to learn every peculiarity of FPGA programming and interfacing. That said, knowledge of the specific area one is targeting is a must. I will now proceed to present the components and tools I have used to progress with the project.

3.1 Equipment

With the help of School of Informatics, I have sourced and used the following equipment throughout the development of the project:

- **Digilent ZYBO Zynq-7000 Development Board**. The board is fitted with an entry level, Xilinx Zynq-7010 FPGA processor, fitted with a general purpose, 650 MHz dual-core ARM Cortex-A9 processor, programmable logic containing 28,000 logic cells and 240 kilobytes of block RAM, 512 megabytes of dedicated distributed RAM and a wide set of input and output interfaces. Full specifications can be found in the Appendix.
- **Digilent PmodSSD**. It is a 14-line, two-digit LED display assembly that can be attached to the ZYBO board and programmed to visually represent data that is being output from the programmable logic. The unit is very useful for debugging purposes while developing an architecture for the FPGA, e.g. to display temporary variables and results.
- **32GB Class 10 microSD card**. To utilize the possibility of interfacing the programmable logic of Zynq-7010 with an operating system, internal

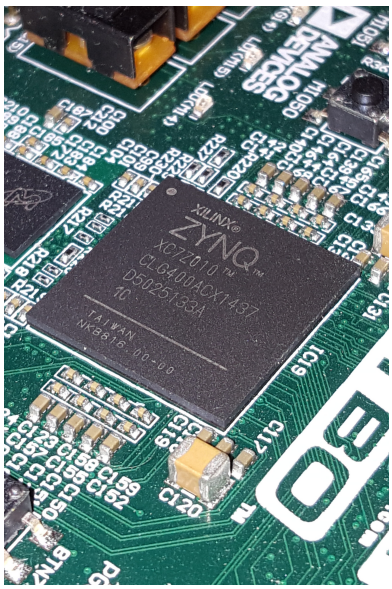

Figure 3.1: Zynq-7010 FPGA on the equipped Xilinx ZYBO board

memory is required. The ZYBO board provides a storage interface in the form of a microSD memory card slot. A distribution of Linux is deployed on the card using a suitable filesystem. I explore this in more detail in Section 3.2.

I have used an HP ProBook laptop to source the required tools, design FPGA logic and benchmark the performance. The laptop is equipped with an intel i5-4210M 2.6GHz dual-core, four-thread processor, 16 gigabytes of DDR3L RAM and a fast 256 gigabyte solid state drive, running Ubuntu 17.10.

3.1.1 Development software

To design and develop the logic for DNA matching in the hardware mentioned before, I have used Xilinx Vivado v2015.4 WebPACK Edition (displayed in Figure 3.2), a programmable logic design suite targeting various Xilinx FPGA hardware, including Zynq-series processors and platforms. The software provides a broad selection of tools for writing hardware code, simulation, statistics and debugging.

3.2 Running a Linux distribution on the FPGA board

The Zybo development board exposes common, useful access interfaces to the user. The on-chip mobile processor, dedicated RAM and a memory card slot, as well as the addition of video output and ethernet ports make it relatively easy to set up a on-board operating system to interface the Zynq processor using a suitable driver. Because of this, a multitude of guides and instructions on how to set up the Linux kernel and run a command-line or graphical interface of a

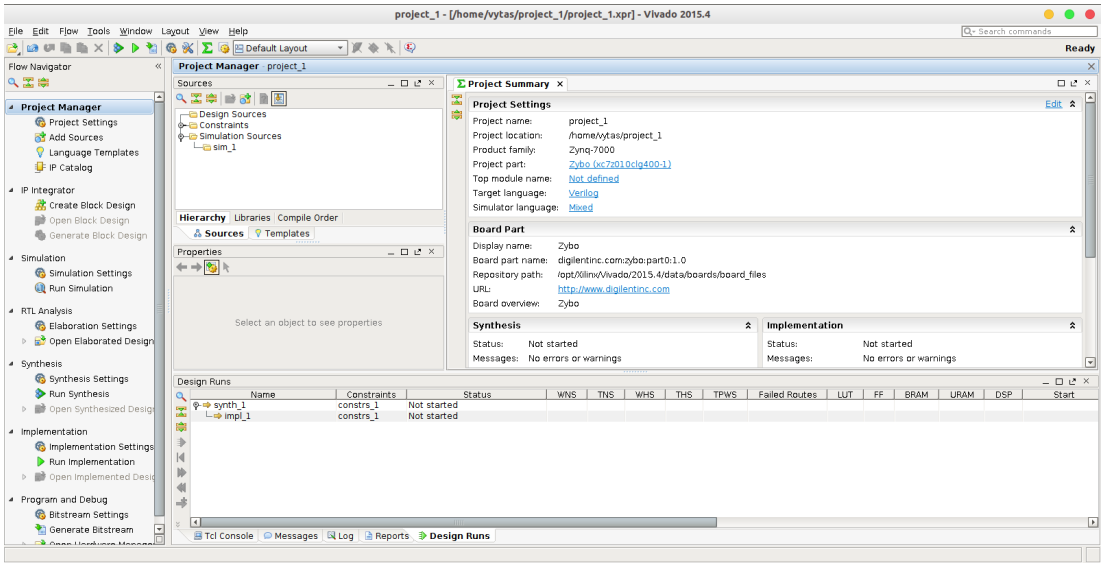

Figure 3.2: Xilinx Vivado v2015.4 main window.

flavour of Linux are available, with various levels of configuration flexibility and suited for different needs.

I have gone through a few preparation materials [43, 44, 45], mostly open-source projects, to build Linux kernel and load the system image onto the memory card. Unfortunately, it did not prove to be an ideal solution: most of the customizable kernel parts were not going to be used, and there was too little time to focus on learning just how to set up a custom environment to implement a single architecture. After a few successful runs, upon having multiple issues with trying to find a baseline interface driver to the Zynq processor as well as lack of resources explaining the system that is built by following the deployment guides, I had to look further.

3.2.1 Xillybus FPGA interface suite

Xillybus [46] was a promising start. As detailed on the website, it is an IP core and driver suite, which provides a predefined interface to develop one's own logic on the FPGA as well as drivers that expose a simple device read / write interface for both Windows and Linux operating systems. For the scale of the project, it is a toolset that is more than sufficient to achieve the goals and performance expected for an implementation of a sequence matching accelerator: the Xillybus drivers interface with standard FIFO (First-In-First-Out) queues on the FPGA side, data transfer rate of which are only limited by the amount of data the Zybo development board is able to process in one go. The rates are discussed in more detail in Section 3.3.

Upon launching, the Xillybus 2.0c Lubuntu 16.04 distribution provides a bash interface and an optional graphical KXDE interface. The IP package contains a

starter Vivado project with sample code and documentation on how to communicate with the FPGA processor.

The Xillybus core is a very efficient way of interfacing and testing infrastructures on an FPGA processor, greatly reducing the development and testing time, which is one of the main problems currently faced by FPGA and ASIC architecture developers.

3.2.1.1 Note on Xillybus and openly available tools

I have used the Xillybus suite together with the *Educational licence* that covers "assignments and student projects as well as in research projects with limited or no budget"[1]. All of the provided software and FPGA interface core are therefore free to use and demonstrate.

A significant amount of hardware tools, platforms and code-bases for FPGAs are still of commercial origin, and many need an expensive licence to be obtained to use, which ultimately impacts the speed of development and deployment of the required architecture as a whole, and sometimes is infeasible at all. Some work is being put into some open-source developments, such as `https://netfpga.org/`, `https://opencores.org/`, and most of them appear as mixed hardware and software projects, using custom FPGA development boards with open-source toolsets. My aim is to therefore present a sufficiently generalized DNA matching architecture instead of focusing too much on board-, FPGA- or vendor-specific software and hardware.

3.2.2 Host to FPGA interface

The FPGA core logic inputs and outputs can be accessed from the host operating system using FIFO queues over Direct Memory Access interface (DMA) that is mapped to device files, such as `/dev/xillybus_stream_dna_y`, and written to / read from using standard IO libraries in a variety of high level programming languages. The interface architecture is a part of the Xillybus core, but communication over DMA for Zynq-based systems is a common way to access the fabric logic, since the Linux system running on the on-chip general purpose processor allows for very efficient communication.

According to the Zynq-7000 AP SoC Technical Reference Manual, the FPGA PL (programmable logic) to PS (processing system) port with a maximum throughput can achieve up to 600 MB/s using 32-bit wide interfaces, whereas Xillybus supports speeds up to 3,5 GB/s. The speed is further limited by the overhead of synchronization and passing data between FPGA and the Linux kernel, as well as FIFO buffering, resulting in a theoretical downstream limit of 350 MB/s, indicated on Xillybus IP Core Factory page. More information specific to Xillybus can be found in the product brief [47].

[1] More information can be found at http://www.xillybus.com/licensing

3.2.3 FPGA as a peripheral

One of the side ideas for the project was to suitably benchmark the performance of the FPGA, running sequence comparisons by queueing jobs and sequence streams remotely, using a high-speed Ethernet interface. I would have completed that given more time, but for the quality of research and results, I only used the FPGA to on-chip processor communication interface to queue reads and writes, which involved copying the base and target sequences to the device before the score fetch procedure begins.

Chapter 4

Implementation

I will describe, in the following chapter, the details of an implementation of the Smith-Waterman algorithm for up to 160 base pair short-read alignment within arbitrarily long sequences on the Zynq-7010 FPGA. I have explored a number of proposals in literature and identified advantages and problems of each model, which paved the way for a modified architecture I will be describing here. The DNA short-read alignment accelerator I have created employs a linear systolic array of processing modules, coupled with 32-bit FIFO communication queues, and sequence streaming, output and synchronization logic. A host processor application is needed to process the scores from the FPGA into a Smith-Waterman score matrix, which I have implemented in C, a high-level programming language, using POSIX threads and Linux kernel IO library.

4.1 Accelerator model overview

I have developed the processing element with reference to the Smith-Waterman algorithm, and followed the idea of Khan et al. [39] for the streaming sequence design; the IO logic of the alignment accelerator is presented in Figure 4.1.

For simplicity and without the loss of generalization, I used a linear gap penalty $= -1$ for both deletion and insertion of a character, a mismatch penalty $= -1$ and a match score $= 2$. This, as I have indicated in later sections, allowed having packed score values and saving space for the score matrix storage. Such an approach constrains the parameters that the user can specify, but ultimately is unavoidable due to limited resources on the Zynq 7010 FPGA. A proposal using the Gotoh extension to the Smith-Waterman algorithm, discussed by Faes et al. [48], and the previously-mentioned approach developed using Handel-C [42] are methods that allow flexibility in the setup of parameters, and are examples of possibilities to extend a simple alignment accelerator to a more portable, feature-packed product.

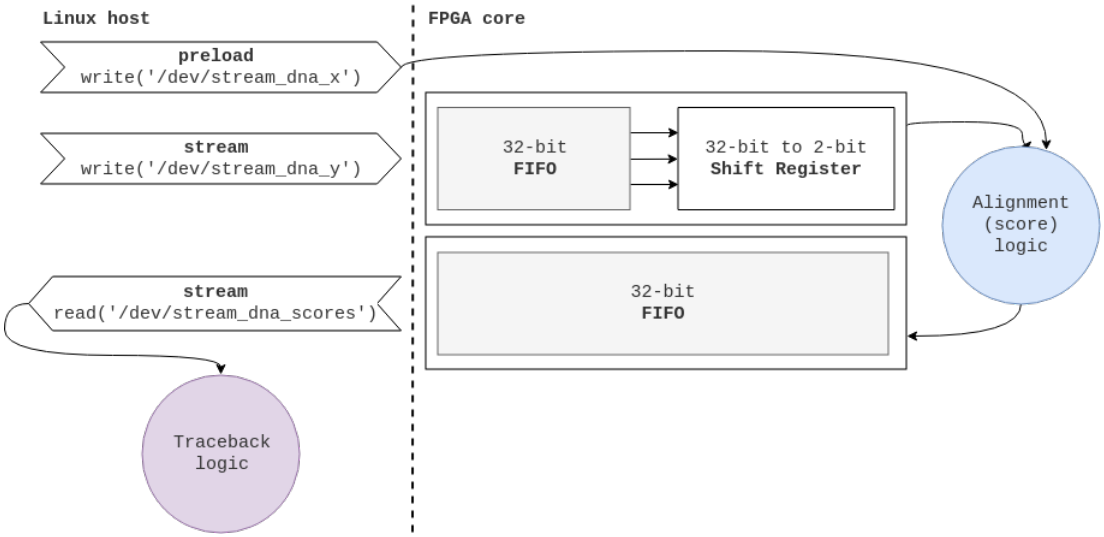

Figure 4.1: The sequence alignment accelerator interface diagram.

4.1.0.1 Storage-based versus streaming IO architecture

A computation for DNA alignment using virtually any dynamic programming algorithm requires a significant amount of space to store scores between characters of two sequences. Therefore, an approach that uses the dedicated memory resources on an FPGA board would require sufficiently large and low-latency storage. For example, consider a DNA alignment problem where, for sequence X, $X_{len} = 160\,bp$ and for sequence Y, $Y_{len} = 1,200,000\,bp$. Then, assuming 16-bit (2-byte) score values for Smith-Waterman score matrix, to store the computed matrix device would need:

$$S = 160 \times 1,200,000 \times 2B \approx 366MB \tag{4.1}$$

Storage requirements would impose strict limits for the sequence length, and might not be feasible for systems such as Zynq-7010, which already uses a portion of the dedicated RAM to run embedded Linux.

A streaming input sequence and output score architecture alleviate the requirements, allowing the score matrix to be handled by the host. This way, using high-speed IO, e.g. Ethernet link, it is possible to extend the solution to support reading data off the FPGA core and streaming it via LAN to an external host that then stores score data in a suitable medium.

4.2 Notation

In the following section I will review some of the core notation in FPGA architecture design that I will later refer to with regards to specific implementation details.

4.2.1 Verilog: types of assignments

In Verilog, two types of variable assignment are used: blocking (`=`) and non-blocking (`<=`). Starting with the latter, non-blocking assignments are used where parallel execution is desired, under synchronization with the system clock. Each value in non-blocking blocks of code is set in conjunction with every other non-blocking assignment that follows. Using non-blocking logic, the architect needs to make sure that every value can indeed be set (calculated) in a single clock cycle, which may require bringing down the clock speed, and need extra resources on the chip.

To leverage tight clock synchronization and data dependencies, blocking assignments are used. The variable that is being set blocks execution of any following logic before the assignment has performed fully. In effect, this allows for asynchronous execution as well as for some complex computation to be split into stages, as I will discuss later.

4.2.2 Verilog: FIFOs and shift registers

In my implementation, the FPGA core communicates with the host (Linux kernel) using FIFOs. I refer to a FIFO as a data buffer that operates under the First-In-First-Out policy. A common type of FIFO queue in Verilog contains input and output ports with signals indicating the status of the buffer, such as `full` or `empty`. It is used to obtain and send data from and to the host in a synchronized manner, and to make sure that no data is lost in the process.

Shift registers are a type of buffers that handle incoming data and sample it at an arbitrary granularity. As the name suggests, input data is stored in a register that is shifted across by a number of bits on each clock cycle, while the last (oldest) element is sampled before being dropped. In effect, it abides by the FIFO policy and is similar to a FIFO queue, but has limited buffer memory and is meant to serve a slightly different purpose. Varying types of shift registers are available, but the implementation presented will only concern a *parallel-in, serial-out* shift register, which loads an array of data in parallel and sieves it out serially in smaller values.

4.2.3 FPGA: building blocks

Lookup tables (LUT), flip-flops (FF) and block memory (BRAM) form the main resource pool on an FPGA. A brief overview follows:

- **LUT**s are, in essence, memory blocks that store truth tables (multiplexers with predefined values for all input combinations) generated from combinational logic. In logic synthesis, they are inferred by writing stateless HDL code, i.e. assigning gate logic to a variable, or sometimes defined manually

as an output multiplexer; the Zynq-7010 FPGA contains 17,600, 3 to 6 input LUTs;

- **FF**s and **BRAM** are the main storage units. FFs are defined in the HDL code as registers, and hold state or other small amounts of data and reside near LUTs. BRAM is different from FFs, as it is standalone, strictly addressable memory that resides in the programmable logic fabric, but provides a handful of space for storing input and output. It is also used to construct FIFO queues as they can require ample storage space. The Zynq-7010 provides 240 KB of block memory and 35,200 1-bit flip-flops.

4.3 Hardware architecture

I have conducted the modelling of the parallel architecture for DNA matching in stages, designing and testing each component as a unit before proceeding to implement the rest of the model. In the following section, I will present the detailed implementation of parts of the architecture, from the beginning to the finished system.

4.3.1 Processing element

The core component of any systolic array based approach is the processing element, or PE. A variety of designs for the PE have been proposed in previous work, some of which I have already discussed previously [38, 37, 49].

Before going into detail, I will briefly present the operation of Smith-Waterman algorithm in a systolic architecture:

- An array of processing elements is initialized, each operating individually in parallel (in a parallel system; or sequentially in a single-threaded environment); initial score values are set to 0;
- At time T, each processing element takes the score stored from previous computation at time $T-1$, as well as scores from previous processing element at times $T-1$ and $T-2$ (refer to Figure 2.7);
- Previous scores at each processing element are saved, current scores are output and stored in the corresponding *diagonal* of the score matrix;
- The obtained values are then compared to find the maximum value (score at time T); if the maximum value is negative, 0 is substituted instead;
- The process is repeated for $X_{len} + Y_{len}$ cycles, where X_{len} is the length of base sequence and Y_{len} is the length of the streaming sequence;
- At the end of the comparison procedure, the alignment and total alignment score are found by navigating along the matrix entries starting from the

highest score(-s) found at the bottom right of the matrix, bottom-up (refer to Figure 2.4).

4.3.1.1 High-level design

To develop an understanding of the way systolic architectures work in more detail, I first started working with a high-level programming language, C++, to design a prototype processing element. The model that I have implemented works in a similar way to the general procedure described above, and the algorithm to compare base sequence X character X_C to stream sequence Y character Y_C for a single PE is displayed in Algorithm 1.

Algorithm 1 Smith-Waterman sequential processing element

```
1: procedure ALIGN(char Y_C, integer top_in)
2:     diag ← top
3:     top ← top_in
4:     if X_C == Y_C then
5:         ms ← diag + match_score
6:     else
7:         ms ← diag − mismatch_penalty
8:     end if
9:     i1 ← left − gap_penalty
10:    i2 ← top − gap_penalty
11:    score ← max(ms, i1, i2, 0)
12:    score_old ← left
13:    left ← score
14:    return score
15: end procedure
```

An array of processing elements is initialized, each with a single character from the base sequence X. The stream sequence Y is then "propagated" across the PE array, using a nested loop to feed a shifting range of $k = X_{len}$ characters, to simulate a pipelined execution.

It is important to note that for the first few and last few iterations, only a portion of processing elements is enabled. The relation is more clearly depicted as the following, where: X_{len} and Y_{len} are base and stream sequence lengths, $P_{enabled}(t)$ is the processing status of PE_i, and $P_{proc}(i)$ is the total amount of iterations or characters from sequence Y processed so far by PE_i:

$$P_{enabled}(i) = \begin{cases} 1 & if \quad (P_{proc}(i) = P_{proc}(i-1) - 1 \\ & and \quad P_{proc}(i) \leq Y_{len}) \\ & or \quad i = 0 \\ 0 & otherwise \end{cases} \tag{4.2}$$

The core difference between the software and hardware model is the fact that sequential operations for each item in score matrix diagonal (every and each processing element score at each point in time) can be calculated in parallel in the final architecture since they only depend on the values from the previous iteration.

4.3.1.2 Conversion to HDL

I have done a few experiments with the design of the PE in Verilog. Using the Vivado simulation tool, I have sketched and simulated a design for PE that incorporates purely non-blocking assignments, i.e. no state definitions, with output values driven each clock cycle. Such design covers an ideal scenario, where a computation is completed by each PE in every cycle, and the time required to process sequences X and Y is equal to a very low $X_{len} + Y_{len}$ clock cycles.

Unfortunately, it is unrealistic to achieve this under real-world conditions. Such approach implies the use of a great amount of LUT resources to store combinational logic and may require a sufficiently slow system clock. During testing, the synthesis stage in Vivado reported the use of more than 49,000 LUTs for an interconnected array of 200 PEs, which is impossible to achieve using relatively low-cost FPGAs.

I have finalized the PE design using state registers and a finite state machine (FSM) for the logic of a single PE. The computation of two character alignment score is done using 2-bit values for characters from each sequence (corresponding to one of A, C, T or G bases), and 10-bit values for intermediate values and final score, based on the following observation: given $score_{match} = 2$, the maximum score at any position in the score matrix will never be higher than $X_{len} \times 2$, that is, $score_{max} \leq 320$, which requires 9 bits to store as an unsigned integer and 10 bits signed. The state logic is given in Figure 4.2. I have chosen 3 states to process the comparison result incrementally, to deal with limited LUT resources on the circuit and meet strict timing constraints.

4.3.2 Array of processing elements

The next step involved connecting an array of PEs. I have done this by using registers to hold the base sequence X, and initial registers for streaming sequence Y and score pipeline register. A PE_i is tied to `sequence_X[i]`, `sequence_Y[i]` and `score_out[i]`. The register holding sequence X is constant, whereas elements of streaming sequence are shifted across Y register. At each iteration, PE_i stores its computed score at a particular time T in its slot in the score array. Each PE is also provided with a `valid` signal to enable stalling the computation while score array is transferred to the host.

```
94    case (state_nxt)
95        RUN1:
96            if (valid_i)
97                begin
98                valid_nxt = 1'b0;
99                diag_nxt  = top_nxt;
100               top_nxt   = top_i;
101               Y_nxt     = Y_i;
102               state_nxt = RUN2;
103               end
104       RUN2:
105           begin
106           i1_nxt    = left_nxt - gap;
107           i2_nxt    = top_nxt - gap;
108           ms_nxt    = (X_i == Y_nxt) ?
109                       diag_nxt + match :
110                       diag_nxt - mismatch;
111           state_nxt = RUN3;
112           end
113       RUN3:
114           begin
115           ms_nxt = ms_nxt[9] ? 10'd0 : ms_nxt;
116
117           if (i1_nxt >= i2_nxt && i1_nxt >= ms_nxt)
118               score_nxt = i1_nxt;
119           else if (i2_nxt >= i1_nxt && i2_nxt >= ms_nxt)
120               score_nxt = i2_nxt;
121           else
122               score_nxt = ms_nxt;
123
124           left_nxt  = score_nxt;
125           valid_nxt = 1'b1;
126           state_nxt = RUN1;
127           end
128   endcase
```

Figure 4.2: 3-state FSM logic for the Verilog Smith-Waterman processing cell.

4.3.3 Streaming data IO interface

I have used two FIFO queues, of 32-bit word width and 512-word depth, to handle the incoming stream sequence Y and output scores. A pre-load interface is used for the base sequence X before the comparison process begins, so no additional buffer resources are used (see Figure 4.1). A status variable tied to the success of a pre-load indicates when the core is ready to accept the incoming stream sequence and begin comparison procedure. This variable gets reset when end of stream is reached and comparison has finished.

There is a reason why I used 32-bit width FIFO queues instead of, say, 10-bit queues to stream output: Xillybus data communication happens at the same rate whether it is connected to an interface of 8-bit width or 32-bit width. The only difference between these two is if, hypothetically, an architecture sent and received data as single units of particular length (i.e. 16-bit integers), making them easier to work with. The high-speed data transfer buses that are present in the Zynq-7010 FPGA have a native 32-bit interface, and sending two 32-bit values consumes the same amount of bandwidth as sending two 8-bit values.

For the streaming sequence, 16 single DNA bases are converted into 2-bit binary values and assembled into a 32-bit word, then written to the device file of sequence Y. I have implemented a 32-bit shift register coupled with a FIFO queue to handle such packed data. Data that comes in from the host is stored asynchronously in the FIFO, with regards to the operation of shift register. When FIFO is not

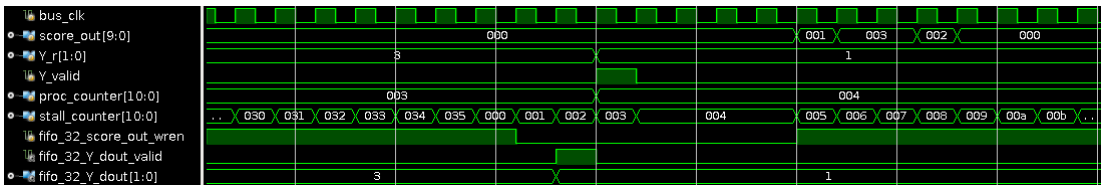

Figure 4.3: Design simulation waveform. The `stall_counter` register is incremented while the scores are written to output FIFO, before proceeding with the processing of the next character from sequence Y, which, in the simulation above, happens when `stall_counter` = 4.

empty and shift register is idle, a new 32-bit output value is stored in the shift register array, and 2-bit values are sieved out every time the module is enabled, to be used by the comparison logic.

4.3.4 Score output logic and PE stalling

To achieve streaming score output, after each parallel computation the PE systolic array and shift register needs to stall to send the score array to the host. This incurs, for each parallel computation of diagonal entries, an additional $X_{len} = 160$ clock cycles, one for each entry in the score array. Part of the timing (waveform) diagram for a simulation of the logic is displayed in Figure 4.3. The 10-bit values are 0-extended to 32-bit values and sent to the output FIFO. A few optimizations can be thought of in this case, I am therefore leaving some space for discussion in the next chapter.

4.3.5 Synchronization of asynchronous data streams

An important piece of functionality for an accelerator that works with streaming data is to synchronize the asynchronous input from host to avoid unexpected results such as data overflows and premature or non-termination. I have defined a set of flags and used Xillybus FIFO data transfer enable signals to mark the start and end of a several processes that happen before and in the middle of the computation process. An example timeline of events is displayed in Figure 4.4.

The only strict requirement for the host is to send the base sequence X to the core before sequence Y to initialize processing elements and prepare core for the comparison process. The `write(Y)` and `read(scores)` host IO calls (abbreviated) can be executed in any order once the sequence X is transferred, this way allowing for a multithreaded operation on the host side (e.g. reading and processing partial score output while transferring the rest of sequence Y).

To deal with communication overhead, the computation is stalled while the host has its write interface open, and continued when new sequence Y data arrives. Once the write interface is closed (meaning the last bases from sequence Y were transferred), a signal (`final_flag`, ref. Figure 4.4) is asserted to indicate the end

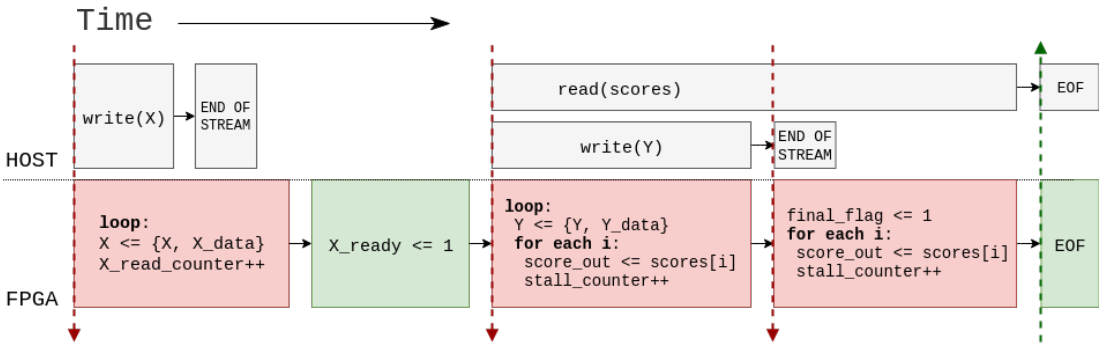

Figure 4.4: Timeline of a sample accelerator access pattern. The red arrows indicate a write request propagated from host to FPGA core, whereas the green arrow indicates core informing host when the last score is transferred.

of input stream. At this point, the core computes the rest of the scores and sends them out, before asserting the end-of-file (EOF) signal, ending the procedure.

4.3.6 Testing and debugging

The final steps to developing a Smith-Waterman DNA alignment accelerator were to simulate a few different use cases and verify that the output is as expected under different access patterns. Simulation of the design and its components involves setting arbitrary values to signals that are asserted by the host in a working design, which allows efficient testing of some real-world scenarios. In addition, delays of arbitrary length can be generated. The following are some of the cases that have been tested to simulate host and FPGA communication:

- Asserted write and / or read interfaces *open* on the host side before the base sequence is transferred: the accelerator stays idle;
- Loaded sequence X and asserted the start of sequence Y stream, leaving host read interface closed: the accelerator stays idle until the host is ready to process output;
- With all host to FPGA interfaces open, in the middle of sequence alignment host stopped reading: accelerator proceeds until output score FIFO is full, then stays idle and waits for the host to free the queue before proceeding with the rest of the computation; a similar situation was tested when host stops writing sequence Y;
- Host closed sequence Y stream interface: accelerator processes the rest of the score matrix and correctly asserts `final_flag` and EOF.

During the actual design process, I initially tested an architecture for a smaller problem size (base sequence X (PE array) length limited to 60 units), running IO accesses from the host using Python scripting interface on Linux bash. This helped me identify synchronization problems, such as a premature end-of-file

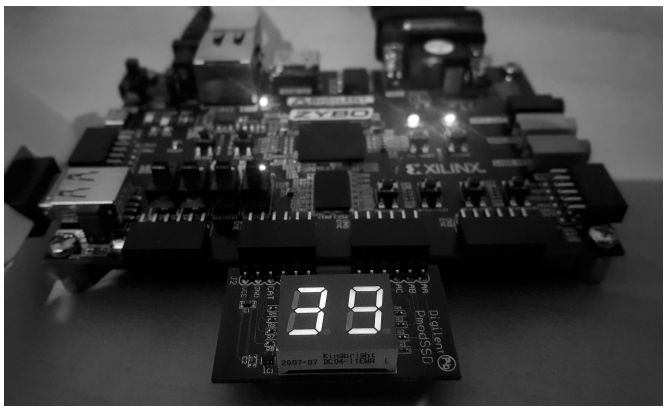

Figure 4.5: Digilent PmodSSD module attached to the ZYBO development board.

signal being asserted on the read interface, non-termination upon closing write interface, as well as other bugs, some that caused garbage or no data to be read from the core, lost data, and similar issues. More on this in Section 4.4.

Hardware design debugging in real-time is usually a very time-consuming and expensive process, therefore building extensive simulation models before deployment of the system is very important. Due to the overhead from communication between FPGA core and host, on the other hand, some cases are hard to identify without seeing the system behave live. I have used the Digilent `PmodSSD` module (refer to Chapter 3), a two-digit LED display, attached to analog IO pins on the ZYBO development board (see Figure 4.5), to help me identify problems in a running system. The setup of the module for debugging followed the steps below:

- Using Vivado tools, I adjusted the board-specific pin layout file to attach free `Pmod` pins to arbitrary registers that I then connected to the top Verilog module;

- I have attached the registers, defined in the top module, to a digit display logic module [50], and tied each bit of the 8-bit SSD input register to status registers on the core, such as FPGA to host FIFO status signals, indicating *write* and *read* interfaces open on the host side, `final_flag`, EOF and several other internal core signals;

- When different status signals are asserted on the core, the SSD displays an arbitrary integer in the range 0 to 99, corresponding to a binary sequence indicating the presence of those signals or lack thereof, as displayed in Figure 4.6.

For example, upon closing the write interface on the host, I expect both `final_flag` and EOF to be asserted shortly after, such that two lower bits of the binary representation of the number displayed are both 1.

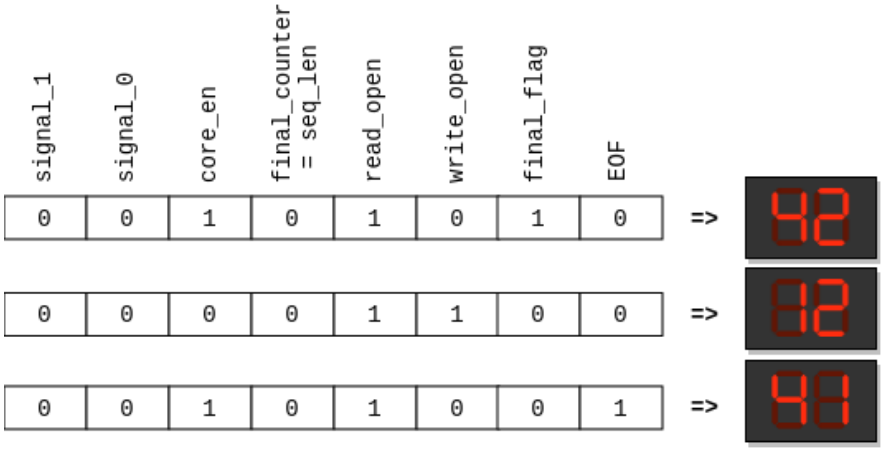

Figure 4.6: SSD input status array, attached to various internal signals and corresponding integer on the LED display, to monitor the current state of the FPGA core.

4.3.7 Synthesis and implementation

As I mentioned in Chapter 2, the synthesis and implementation stages in Vivado hardware design cycle are responsible for generating a logic circuit for the written code and mapping it virtually onto the targeted hardware.

The final implementation run resulted in resource utilization depicted in Figure 4.7. I observed a high use of LUTs, nearly topping the resource pool; that results mostly from the sum of combinational comparison logic contained in each PE. Due to such complexity of the PE, my accelerator design initializes an array of a total 160 processing elements, effectively limiting the maximum length of a short-read (base sequence) to 160 base pairs. An FPGA with more LUT resources would allow the size of the PE array to increase to align longer base sequences. It should, of course, be noted that this problem has been considered in previous work and alleviated or avoided altogether by using approaches such as: trimming comparison logic to process single and two-bit values, for total comparison score only [38], using a multiple pass strategy with complex sequence access patterns to reuse PEs, in favour of only initializing a relatively small amount of PEs [37]. Finally, Xillybus IP core (the design wrapper for communication with the host running on the ARM processor) uses a fair amount of logic (up to 4,000 LUTs) to drive the default peripherals on the ZYBO board, such as VGA interface, which can be considered omitted in an implementation of such accelerator in a fully custom hardware design.

4.3.7.1 Timing

Worst Negative Slack (WNS) reported a positive 1.126 ns delay, which indicates that every signal path in the design meets the timing constraints. To meet these, each connection between two distinct endpoints on the FPGA must be possible to route in such a way that an asserted signal is guaranteed to have sufficient

Resource	Utilization	Available	Utilization %
LUT	15027	17600	85.38
LUTRAM	193	6000	3.22
FF	14891	35200	42.30
BRAM	3.50	60	5.83
IO	93	100	93.00
BUFG	2	32	6.25
MMCM	1	2	50.00

Figure 4.7: Final architecture FPGA resource utilization report.

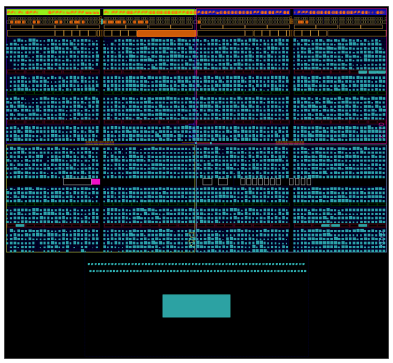

Figure 4.8: Final architecture resource layout on the physical FPGA die.

time to travel across that route during a clock cycle. If part of the design logic is impossible to execute in a single clock cycle, resulting in a negative WNS, the clock frequency should either be scaled down or the operation needs to be staged (pipelined).

4.4 Software interface

After completing and testing the FPGA accelerator architecture, I came to the software part of the alignment processor. I used both Python and C to write code; the former was useful to write short test code sequences, whereas I have written the final application in C, using POSIX threads library and Linux kernel IO functions.

Unfortunately, upon testing the accelerator, I ran into data manipulation problems that I did not have enough time or resources to fix during the development, which ultimately led to a lack of sound performance comparison results; I will elaborate more in this section.

4.4.1 Testing data flow

To start, I have written a method in Python that converts a DNA sequence provided as an argument into a binary representation, optionally saving it into a binary file. The code parses the given data - a sequence of DNA bases, in a single line or across multiple lines. The bases are then converted into a 2-bit representation, such that $\{A \equiv 00_2, C \equiv 01_2, G \equiv 10_2, T \equiv 11_2\}$. The output is given in Figure 4.9, with the sequence being converted shown on the left and the corresponding binary entries on the right.

In addition to interfacing the accelerator on the FPGA, the script has helped me to proceed with the coupling of processing elements into a systolic array and

tcccatagtcgcctgc...	00000000 | 03 01 01 01 00 03 00 02 03 01 02 01 01 03 02 01
...cacagggaaccaggga...	00000010 | 01 00 01 00 02 02 02 00 00 01 01 00 02 02 02 00
...actttatcccatacta...	00000020 | 00 01 03 03 03 00 03 01 01 01 00 03 00 01 03 00
...acaatttctgtaacaa	00000030 | 00 01 00 00 03 03 03 01 03 02 03 00 00 01 00 00

Figure 4.9: Sample DNA sequence conversion to binary results, obtained by running `hexdump -C` on the binary DNA file.

to test the correctness of the implementation in Verilog, by filling up sequence registers and testing the architecture using preloaded values, in simulation mode.

I have initially tested my implementation by launching two separate scripting processes (`bash $ python`), one waiting for the sequences to be written to device and outputting scores, another to open sequence write interfaces and write data. The reading side blocks until any sequences are written into the device and outputs scores once they are calculated on the FPGA and outputs the rest of the score matrix entries once all FPGA interfaces are closed on the writing side.

4.4.2 Application to obtain and process scores

Once I familiarised myself with basic data synchronization rules between Linux and FPGA FIFO interfaces, I developed an interface in C. I have made the sequence input format similar to the software application described in Section 4.3.1.1, to compare both software and hardware-accelerated results and runtime.

The process flow of the software interface is as follows:

- Sequences X and Y are loaded using `fread()` command, and stored in arrays of characters; length of sequences is retrieved using the `stat.h` library;
- Space is allocated for the score matrix. Since the ZYBO development board is limited to 512 MB memory, a part of which is used by Linux host, the Y sequence is to limited 1,200,000 bp (characters), I discuss potential improvements to alleviate use of memory in the next chapter;
- The FPGA FIFO interfaces are opened using Linux kernel IO libraries as suggested in the Xillybus host programming guide[1], a POSIX thread is spawned to read FPGA output and base sequence X of up to 160 bp length is written to the device;
- The stream sequence Y write process begins, operating in parallel with score read process, which reads 640 bytes (160 4-byte scores) at a time and stores received data into the previously allocated score matrix;
- Interfaces are closed upon finishing and total runtime is reported.

[1]Found at http://xillybus.com/downloads/doc/xillybus_host_programming_guide_linux.pdf

4.4.2.1 Sequence transfer

As I covered previously, I have optimized the internal 32-bit FIFOs in FPGA core to work with DNA sequences of 16 bp length in a single transfer. This has saved communication bandwidth, but ultimately complicated the design of sequence stream logic, both on the hardware and software side.

In the software application, I process sequences in split parts of 16 or fewer base pairs (if the sequence length is not a multiple of 16), and convert to binary values before passing into the write interface. A `long` (4-byte) type value `partseq` is initialized to 0, and is filled iteratively according to the following:

```
partseq = (partseq << 2) OR (binary base)
```

where `binary base` is a 2-bit value, indicating the next DNA base to be added, one of 0, 1, 2 or 3. Once `partseq` is filled, it can then be written to the FPGA core, indicating a 4-byte transfer. This way, the two least significant bits (LSBs) indicate *last* DNA base of the partial sequence.

In the hardware side of things, the LSB position indicates the *first* base pair of the sequence, and requires the software to send sequences in such an order as well. Since a right shift operation in C is well-defined and intuitive to use, and the FPGA alignment processor is fundamentally dependent on sequences that start from the LSB, I have chosen to process sequences backwards on the software side, by reading the DNA base pair arrays from the end to beginning. Such approach can incur a performance penalty depending on the system the application is run on, due to lack of cache prefetching, thus forward-iterating loops may be preferred. To alleviate this, the DNA base conversion can be substituted to one that stores binary bases in the order from LSB to MSB.

4.5 Setbacks

There have been a number of problems associated with coupling software and hardware architectures together to produce correct and sound results. I found that it is crucial to take into account the peculiarities of synchronization and correct indexing on the software side, the lack of which can very easily produce unexpected results.

In particular, it was a complex process obtaining and storing the scores that are sent from FPGA to the host. The scores are piped from the internal score FIFO in sequences of 32-bit values, representing arbitrary *diagonals* of the score matrix. Depending on the input sequence length, the software interface is only be employed to fetch a fixed amount of the scores from the output sequence, whereas the FPGA is initialized to strictly 160 PEs and sends 160 values at each iteration, complicating the indexing of resulting score matrix.

Despite my architecture working correctly and soundly in the simulation stage, there happened to be issues when it was employed in real-time. Bugs in the

hardware and software architectures caused multiple reviews of the code and required extensive testing, including numerous covers of corner cases. I have gone through multiple iterations inspecting C and Verilog code, printing values that were being communicated between software and hardware and indices indicating the space for score storage. The latter has halted, to some extent, the evaluation process; I elaborate more on it in the next chapter.

Chapter 5

Evaluation and proposals

Upon developing the accelerator architecture, I have tested it against a reference Smith-Waterman algorithm performing as a software-only processor in C++. Regrettably, during evaluation, my design did not manage to handle stream sequences longer than roughly 15,000 base pairs due to an unidentified cause in the data transport layer. I was therefore not able to produce verifiable performance comparison results. Instead, in the following chapter, I will present the evaluation methods I was prepared to use for the processor and an exploration of the problem with the design. In a fully working system, a number of factors are in the performance equation and the overall efficiency is greatly limited by data storage resources and communication overhead; I will therefore also propose a number of possible improvements for a Smith-Waterman hardware alignment accelerator.

5.1 Evaluation

As I have mentioned, I had to stall the evaluation to solve problems in score matrix indexing and transmission of sequence code. During final development stages, the accelerator worked correctly most of the time, and was tested against a variety of cases using arbitrary length sequences. The problem surfaced when sequences of more than 120,000 base pairs was being streamed to the device's `xillybus_stream_dna_y` interface: the software interface received an end-of-stream signal prematurely upon reading $80,000 - 110,000^{th}$ score diagonal. Due to lack of time, I was not able to track down the problem after carefully looking into IO constructs on the software side and FIFO synchronization on FPGA. It is most likely that the stream sequence Y writes did not propagate correctly, since the read call for results output would block prematurely upon transferring the whole DNA. Lesson learned: a significant portion of development time must be set aside to take in account every case that the accelerator might be tested against for sound evaluation.

For a performance evaluation of a fully working system, ideally, a comparison of a

```
 -- brca_SR*.txt: short-reads from "WES of homo sapiens - BRCA related breast and ovarian
cancer" (SRR5604276) from the Sequence Read Archive (https://trace.ncbi.nlm.nih.gov/Traces/sra/?
run=SRR5604276) (100 bp length each)

 -- brca_mutation_50.txt: Homo sapiens truncated breast and ovarian cancer susceptibility
protein 1 (BRCA1) gene, exon 16 and partial cds (https://www.ncbi.nlm.nih.gov/nuccore/
DQ075361.1) (50 bp length)

 -- brca_mutation_120.txt: Homo sapiens isolate 11_7_7F breast and ovarian cancer susceptibility
protein (BRCA1) gene, exon 11 and partial cds (https://www.ncbi.nlm.nih.gov/nuccore/DQ299313.1)
(120 bp length)

 -- brca_mutation_151.txt: Homo sapiens partial BRCA1 gene for Breast cancer associated protein
1, isolate SGJH-AUBC4 (https://www.ncbi.nlm.nih.gov/nuccore/HE600033.1) (151 bp length)

 -- brca_mutation_139.txt: Homo sapiens partial BRCA1 gene for Breast cancer associated protein
1, isolate SGJH-AUBC8 (https://www.ncbi.nlm.nih.gov/nuccore/HE600034.1) (139 bp length)

 -- chr17_brca.txt: >NC_000017.11:c43125483-43044295 Homo sapiens chromosome 17, GRCh38.p12
Primary Assembly (https://www.ncbi.nlm.nih.gov/gene/672) (81,188 bp length)

 -- chr17_brca_ext.txt: >NC_000017.11:c43600000-42425000 Homo sapiens chromosome 17, GRCh38.p12
Primary Assembly (extended around BRCA1 gene, 1,175,000 bp length)
```

Figure 5.1: Evaluation DNA sequences. The BRCA1 breast and ovarian cancer gene is a popular target for genetic sequencing.

software versus hardware accelerated solution is preferred, using ranging types of gene sequences and short-reads, tested under different conditions. *Multiple alignment* (mapping multiple short-reads onto reference sequence) is another preferred area of performance testing due to the big amount of short-reads that are being sequenced using NGS technologies and the ability to evaluate the performance gains in a scaled fashion.

I would have chosen to measure the performance of the accelerator by using randomly generated, arbitrary length test sequences[1] as well as mapping *BRCA1* (breast and ovarian cancer) gene mutation exons and short-reads from an NGS sequencing run to a relevant part of human (*Homo sapiens*) chromosome 17. I have obtained the former from the NCBI open-access nucleotide, gene and SRA databases; many more sequences are available to download and use to benchmark sequence alignment methods. The base sequences (X) obtained range from 50 to 160 bp, whereas the stream sequences (Y) range between roughly 80,000 and 1,200,000 bp. The test sequences and supporting information are displayed in Figure 5.1.

The software application in C++ for sequence comparison was run on a Linux Ubuntu machine with intel i5-4210M dual-core processor. Preliminary results indicated that the software approach took 0.2 to 0.6 seconds to process reads of length 100 to 151 bp across a 88,188 bp BRCA1 gene sequence, and 4.5 to 8 seconds to process the same reads across a 1,175,000 bp extended partial chromosome 17 sequence. The hardware-accelerated approach, in addition to being unable to compare sequences longer than roughly 120,000 base pairs, suffered from a data transmission overhead and has achieved lower rates for a subset of the test sequences: it has taken 1 to 1.4 seconds to process the aforementioned reads within the BRCA1 gene sequence.

The tests that I was able to perform and their results are displayed and briefly discussed in Figure 5.2.

[1] Generated in http://www.bioinformatics.org/sms2/random_dna.html

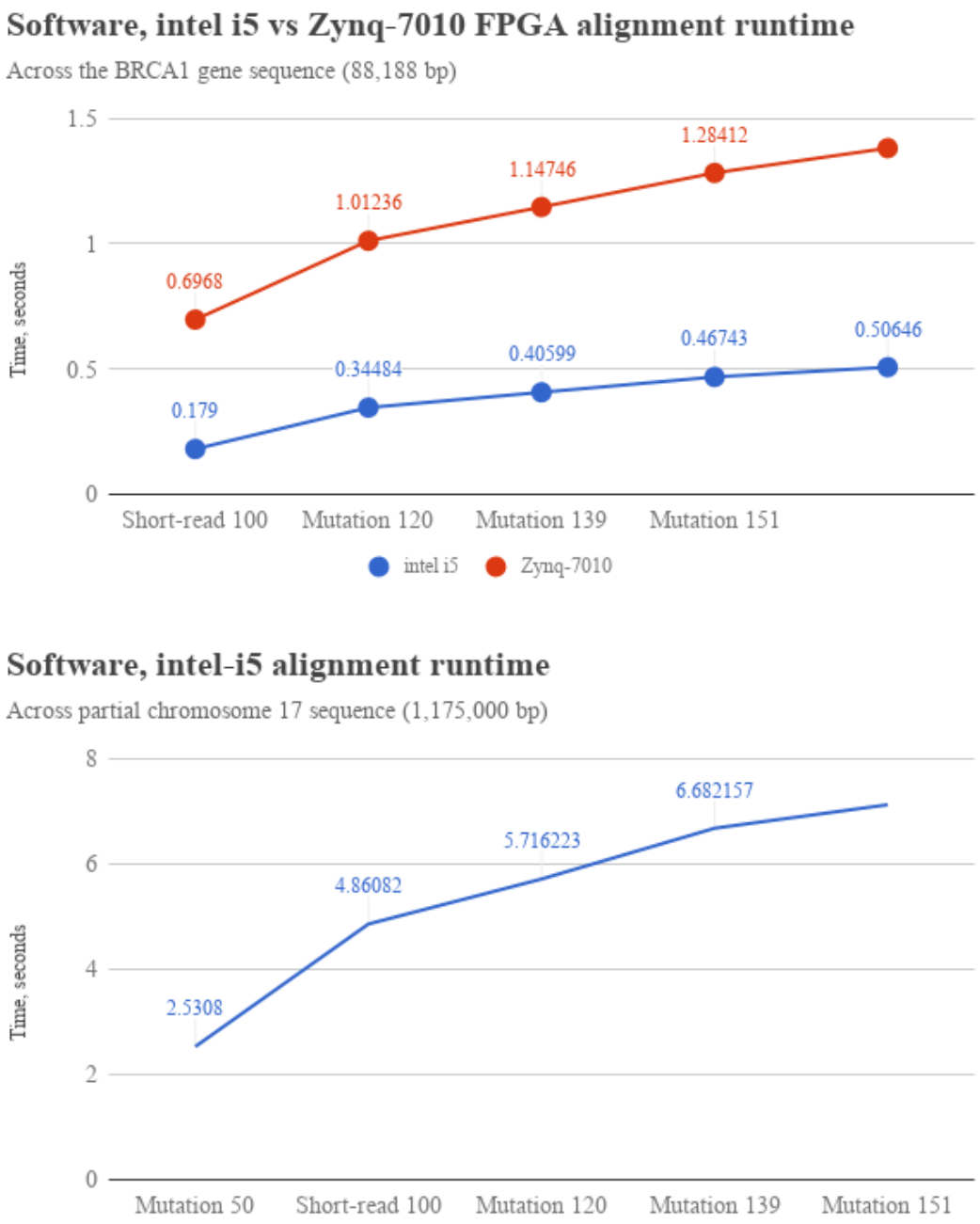

Figure 5.2: Comparison of runtimes observed using the implemented sequence alignment architecture in both C++ on a personal computer with intel i5 processor and and Verilog architecture deployed on the Zynq-7010 FPGA. In the first graph, it can be observed that using FPGA-coupled method yields significantly higher runtime, and that is almost entirely due to the increased communication and score storage overhead. The FPGA is programmed to always send 160 32-bit values, which implies that the hardware processing speed does not depend on the size of short-read that is used. This clearly indicates that a 32-bit streaming architecture might not be ideal for an alignment accelerator.

It should be noted that few conclusions should be drawn from the results. It is extremely important to consider the following factors that influenced the accelerator performance: the architecture spending at least 160 clock cycles idle transferring the score diagonal, communication overhead between host and FPGA core to transfer sequences and obtain results (in addition to the use of predefined IP core such as Xillybus), as well as read and write thread communication overhead on the host side. I believe these were the core factors that influenced the under-performance of my architecture.

5.2 Improvements and alternative approaches

Due to the nature of a linear systolic array, the alignment computation in hardware is inherently parallel, but there is a lot more to consider in terms of scalability and performance. In the following sections I present the areas that I believe could be improved in terms of the proposed implementation.

5.2.1 Packing scores

A possible optimization given the described architecture would be to fit a *few* of the produced scores to a 32-bit line, in particular, stream 3 10-bit values at once using the implemented design. This would, in effect, cut down the time required to stall for transfers in the core logic *three times*. Using FPGAs that support 64- and 128-bit high-speed transfers, it would be possible to reduce that number even further.

5.2.2 Parallel sequence load

Using more complex synchronization constructs, the shift register could operate without a single cycle delay for each load and instead provide characters to the PEs continuously. That can be achieved by checking if the FIFO queue for sequence Y is not empty before the current register elements are exhausted, in conjunction with output processing.

5.2.3 External host via Ethernet

Longer stream sequences will result in more memory taken on the host side to store the score matrix for a full trace-back process. An improvement for the architecture could utilize the 1G Ethernet interface available on the ZYBO (and other similar Systems-on-Chip) to stream alignment scores directly to an external host with a suitable driver to store them in memory. This would help with the limited memory on the ZYBO board by freeing it and allowing it to possibly be used for improvements in the accelerator architecture, and removing constraints

on the memory size supported. An external host with tens of gigabytes of memory would support short-read alignment on billion base pair length sequences (with reference to Equation 4.1).

5.2.4 Reusing processing elements

The current architecture allows a limited 160 base pair sequence to be compared to a longer sequence streamed from the host. A modification of the next-pass strategy proposed by Shah et al. [37] would allow the PE array currently used to align a single sequence to be reused. After a single iteration, scores in the `scores` array would be transferred to a temporary backing store (BRAM), and next part of the base sequence loaded into `sequence_X` array to be used by the PEs. After a fixed number of loading iterations, a single alignment iteration would be complete, allowing it to progress with the next character from stream sequence.

This proposal, on the other hand, requires a lot of other factors to be considered. An architecture that processes millions of bytes of data needs a significant amount of throughput bandwidth available to stream this data, otherwise memory constraints apply and an efficient solution to store the score matrix on the FPGA itself is needed, which I will propose next.

5.3 Solving memory constraints

Due to memory constraints and the time required for the FPGA to spend idle while transferring scores to the host, I have considered a solution that, with certain constraints on the alignment results, is invariant to both of the problems.

5.3.1 Proposal definition

Assume that, in the Smith-Waterman model, there exists a value that determines the maximum gap between parts of the two sequences for the alignment to be significant (similar to affine gap penalty, with reference to Chapter 2, Equation 2.3). Then a technique for a *best alignment so far* could be employed to store a partial score matrix for each best alignment found, and either transferred to the host to manipulate or used with traceback to find an alignment in the FPGA core.

Since the definition of the problem I am exploring is short-read alignment, the user will likely be most interested in finding a match of the whole base sequence across parts of the stream sequence. The problem can be more strictly defined as finding the highest score along the *bottom row* of the score matrix and tracing the best alignment bottom-up, as previously depicted in Chapter 2, Figure 2.4. The task (and the main idea of the proposal) is to then, at each iteration, keep

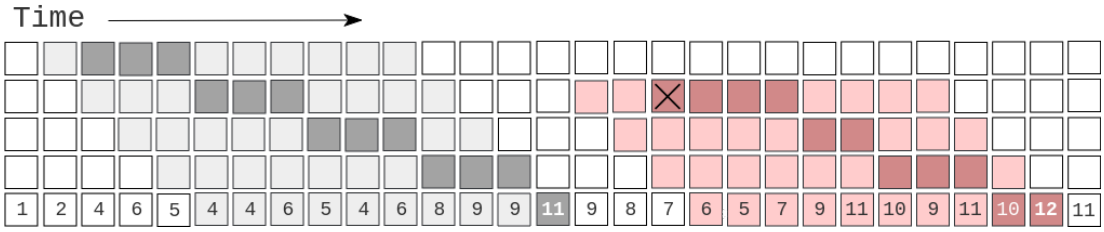

Figure 5.3: A visualization of the proposed *best alignment so far* technique. The light coloured tiles represent space allocated to store partial scores for the alignment that has yielded (one of) the highest score at the bottom row. The darker tiles indicate the traceback path, as done in Smith-Waterman algorithm; in this case the grey tiles are scores included in the final alignment, and the pink ones are discarded upon finding that the gap between subsequences exceeds 2, i.e. two movements left.

track of an arbitrary amount of highest scores seen so far at the bottom level of the matrix and a partial snapshot of the score matrix with each, capturing only the data containing significant alignments. This, in turn, would highly reduce the amount of data that needs to be stored or transferred after each iteration. A visualization of the approach is depicted in Figure 5.3.

5.3.2 Elaboration

Firstly, I shall define the space requirement for any single score in the score matrix using the Smith-Waterman algorithm:

$$S_{score} = \left\lceil \frac{\log_2(X_{len} \times Score_{match})}{8} \right\rceil B \tag{5.1}$$

where: X_{len} = length of the base sequence
$Score_{match}$ = positive match score constant

Then, the space S required to store the partial score matrices is going to be:

$$S = (g_{max} \times {X_{len}}^2 \times S_{score}) \times N \tag{5.2}$$

where: g_{max} = maximum gap size between any two subsequences
${X_{len}}^2$ = length of the base sequence, squared
S_{score} = size of single score, in bytes
N = amount of optimal alignments expected

Let's see the approach in practice, and define the constants: $X_{len} = 160\,bp$, $Score_{match} = 2$, yielding $S_{score} = 2\,B$, $g_{max} = 4$ and $N = 10$. Then, putting the constants into Equation 5.2, the maximum amount of space needed to store the partial alignment scores is approximately $2\,MB$, given absolutely no overlapping score sections. This is more than $150x$ less space consumed than the example presented in Chapter 4, Equation 4.1.

Depending on the size of distributed memory in an FPGA or usable dedicated memory , $g_{max} \times N$ may be constrained such that only a limited amount of partial alignment scores can be stored. Nevertheless, such method would be useful in practice where both constants are sensibly low.

5.3.3 Considerations

My proposed improvement over a traditional approach needs to have a few problems addressed. Firstly, an alignment may need to be discarded due to a gap found to be longer than the maximum defined gap. This might (and in most cases, will) waste FPGA resources, as it will have possibly iterated close to the top of the score matrix for that alignment, before marking the alignment irrelevant and continuing the rest of the process. A suggested solution would therefore employ *parallelizing* the work done for traceback and alignment, such that the processor would continue aligning sequences and looking for highest scores along the bottom row of the score matrix, and only stall due to lack of resources for the traceback process (buffers for temporary traceback score storage, combinational logic, and similar).

Furthermore, a few highest scoring alignments may have overlapping score regions. In the most general case, as defined in Equation 5.2, this is covered by an assumption that each separate alignment would have their data stored separately. In this case, the assumption implies a high degree of repetition of data, which is generally unwanted, but does not have a big impact on the use of memory for the proposed system. Either way, I believe the issue should be addressed as a part of the proposal, in a form of a smart data storage and synchronization solution, but I am not going to cover this in greater detail due to the limited scope of the project.

Chapter 6

Conclusion

I have presented a parallel, FPGA-based sequence alignment design to compute Smith-Waterman algorithm scores on an entry level Zynq-7010 FPGA. Previous work in systolic architecture development and the parallelizable nature of the algorithm has supported the custom design that I have implemented. Despite not achieving results to verify the performance of a hardware-accelerated versus a general purpose hardware-driven solution, I have identified the areas where reconfigurable hardware can improve runtime, as well as issues that can be solved by employing a fully-custom design, thus maximizing utilization of the resources available. On the other hand, it is important to consider the communication overhead, which, especially in the case of my proposed design, can have a negative effect on the overall performance. Finally, better time management on my part would have certainly helped sort out at least the fundamental design issues. This also reflects on the note from previous chapters, that development using hardware design languages is a comparatively resource- and time-intensive process. In retrospect, I could have used C to generate logic design constructs for a prototype design; having said that, writing in pure HDL allows optimization of any arbitrary piece of computation and can ultimately enable the creation of more efficient designs, given enough time and experience.

6.1 Next steps

Going forward, I believe that the research I have done can be interesting for future work. The background, research and findings during the implementation stage helped me visualize and evaluate the scope of the sequence matching problem. Given a reconfigurable processor with enough computational (LUT) resources, wide high-speed data buses and use of the aforementioned proposals to save memory and data transfer bandwidth, the design could be extended to support longer sequences and in-place score traceback (optimal alignment). Ultimately, a design as such could be used to prototype and design an ASIC for a scalable architecture.

Appendix

Zynq-7000 ARM/FPGA SoC Trainer Board platform[1]

Features:

- On-board JTAG programming and UART to USB converter
- On-chip analog-to-digital converter

Key FPGA Specifications:

- Part number: XC7Z010-1CLG400C
- Logic cells: 4,400 logic slices (4 6-input LUTs and 8 flip-flops)
- Block RAM: 240 KB
- DSP slices: 80
- DDR3 RAM: 512 MB w/ 1050Mbps bandwidth
- Internal clock: 450 MHz+

Connectivity and On-board I/O:

- Pmod: 6 Pmod ports (1 processor-dedicated, 1 dual analog/digital, 3 high-speed differential, 1 logic-dedicated)
- General purpose: 6 pushbuttons, 4 slide switches, 5 LEDs
- Audio: Audio codec with headphone out, microphone, and line in jacks
- Networking: Gigabit Ethernet
- USB: USB 2.0
- Storage: MicroSD card slot
- Video: Dual-role (source/sink) HDMI port, 16-bit VGA port

[1]Taken from https://reference.digilentinc.com/reference/programmable-logic/zybo/start?redirect=1id=zybo:zybo

Bibliography

[1] D. B. Thomas, L. Howes, and W. Luk, "A comparison of cpus, gpus, fpgas, and massively parallel processor arrays for random number generation," in *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '09, (New York, NY, USA), pp. 63–72, ACM, 2009.

[2] C. Cullinan, C. Wyant, T. Frattesi, and X. Huang, "Computing performance benchmarks among cpu, gpu, and fpga," 2012.

[3] D. Burger and J. R. Goodman, "Billion-transistor architectures - guest editors' introduction," *IEEE Computer*, vol. 30, no. 9, pp. 46–49, 1997.

[4] E. Waingold, M. B. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. I. Frank, P. Finch, R. Barua, J. Babb, S. P. Amarasinghe, and A. Agarwal, "Baring it all to software: Raw machines," *IEEE Computer*, vol. 30, no. 9, pp. 86–93, 1997.

[5] J. K Pace and C. Feschotte, "The evolutionary history of human dna transposons: Evidence for intense activity in the primate lineage," *Genome research*, vol. 17, pp. 422–32, 05 2007.

[6] K. Ranjan, P. Minakshi, and G. Prasad, "Application of molecular and serological diagnostics in veterinary parasitology," *J. Adv. Parasitol.*, vol. 2, no. 4, pp. 80–99, 2015.

[7] F. Sanger, S. Nicklen, and A. R. Coulson, "Dna sequencing with chain-terminating inhibitors," *Proc Natl Acad Sci U S A*, vol. 74, pp. 5463–5467, Dec 1977.

[8] S. C. Schuster, "Next-generation sequencing transforms today's biology," *Nature Methods*, vol. 5, no. 16, 2007.

[9] N. Hall, "Advanced sequencing technologies and their wider impact in microbiology," *Journal of Experimental Biology*, vol. 210, no. 9, pp. 1518–1525, 2007.

[10] Y. Kodama, M. Shumway, and R. Leinonen, "The sequence read archive: explosive growth of sequencing data," *Nucleic Acids Res*, vol. 40, pp. D54–D56, Jan 2012.

[11] Bethesda (MD), *Sequence Read Archive Handbook*. National Center for Biotechnology Information (US), 2010. `https://www.ncbi.nlm.nih.gov/books/NBK47528/`, accessed 31 March, 2018.

[12] 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*, vol. 48, no. 3, pp. 443 – 453, 1970.

[13] S. F. Altschul, "Global and local sequence alignment," 2011. `https://web.archive.org/web/20170829134200/http://www.cs.umd.edu/class/fall2011/cmsc858s/Alignment.pdf`, accessed 31 March 2018.

[14] C. Charras and T. Lecroq, "Knuth-morris-pratt algorithm." `http://www-igm.univ-mlv.fr/~lecroq/string/node8.html`, accessed 31 March 2018.

[15] N. B. Nsira, M. Elloumi, and T. Lecroq, "On-line string matching in highly similar dna sequences," *Mathematics in Computer Science*, vol. 11, pp. 113–126, Jun 2017.

[16] N. Singla and D. Garg, "String matching algorithms and their applicability in various applications," *International Journal of Soft Computing and Engineering (IJSCE)*, vol. 1, Jan 2012.

[17] S. R. Eddy, "What is dynamic programming?," *Nature Biotechnology*, vol. 22, pp. 909–910, Jul 2004.

[18] D. Sankoff, "Matching sequences under deletion/insertion constraints," *Proceedings of the National Academy of Sciences*, vol. 69, no. 1, pp. 4–6, 1972.

[19] D. S. Hirschberg, "A linear space algorithm for computing maximal common subsequences," *Commun. ACM*, vol. 18, pp. 341–343, June 1975.

[20] R. A. Wagner and M. J. Fischer, "The string-to-string correction problem," *J. ACM*, vol. 21, pp. 168–173, Jan. 1974.

[21] National Center for Biotechnology Information, U.S. National Library of Medicine, "Basic local alignment search tool." `https://blast.ncbi.nlm.nih.gov/Blast.cgi`, accessed 31 March 2018.

[22] T. Smith and M. Waterman, "Identification of common molecular subsequences," *Journal of Molecular Biology*, vol. 147, no. 1, pp. 195 – 197, 1981.

[23] J. R. White, M. Matalka, W. F. Fricke, and S. V. Angiuoli, "Cunningham: a blast runtime estimator," *Nature Precedings*, 2011.

[24] E. Sotiriades, C. Kozanitis, and A. Dollas, "Fpga based architecture for dna sequence comparison and database search," in *Proceedings 20th IEEE International Parallel Distributed Processing Symposium*, pp. 8 pp.–, April 2006.

[25] L. Liu, Y. Li, S. Li, N. Hu, Y. He, R. Pong, D. Lin, L. Lu, and M. Law, "Comparison of next-generation sequencing systems," *Journal of Biomedicine and Biotechnology*, p. 11, 2012.

[26] T. P. Niedringhaus, D. Milanova, M. B. Kerby, M. P. Snyder, and A. E. Barron, "Landscape of next-generation sequencing technologies," *Anal Chem*, vol. 83, pp. 4327–4341, Jun 2011.

[27] Xilinx Inc., "A xilinx zynq-7000 all programmable system on a chip," 2012. `https://en.wikipedia.org/wiki/Field-programmable_gate_array#/media/File:Xilinx_Zynq-7000_AP_SoC.jpg`, accessed 29 December 2017.

[28] T. T. Hieu and T. N. Thinh, "mdfa: A memory efficient dfa-based pattern matching engine on fpga," *Wirel. Pers. Commun.*, vol. 78, pp. 1833–1847, Oct. 2014.

[29] T. N. Thinh, T. T. Hieu, V. Q. Dung, and S. Kittitornkun, "A fpga-based deep packet inspection engine for network intrusion detection system," in *2012 9th International Conference on Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology*, pp. 1–4, May 2012.

[30] Y. H. Cho and W. H. Mangione-Smith, "Deep network packet filter design for reconfigurable devices," *ACM Trans. Embed. Comput. Syst.*, vol. 7, pp. 21:1–21:26, Jan. 2008.

[31] V. Pradeep, "Ethereum's memory hardness explained, and the road to mining it with custom hardware," 2017. `https://www.vijaypradeep.com/blog/2017-04-28-ethereums-memory-hardness-explained`, accessed 30 November 2017.

[32] Cadence Design Systems Inc., "Asic prototyping simplified," 2011. `https://www.cadence.com/content/dam/cadence-www/global/en_US/documents/tools/pcb-design-analysis/asic-prototyping-tp.pdf`, accessed 30 December 2017.

[33] Xilinx Inc., "Emulation & prototyping," 2017. `https://www.xilinx.com/applications/asic-prototyping.html`. accessed 30 December 2017.

[34] M. C. Herbordt, T. V. Court, Y. Gu, B. Sukhwani, A. Conti, J. Model, and D. DiSabello, "Achieving high performance with fpga-based computing," *IEEE Computer*, vol. 40, no. 3, pp. 50–57, 2007.

[35] A. Boukerche, J. M. Correa, A. C. M. A. de Melo, and R. P. Jacobi, "A hardware accelerator for the fast retrieval of DIALIGN biological sequence alignments in linear space," *IEEE Trans. Computers*, vol. 59, no. 6, pp. 808–821, 2010.

[36] O. Knodel, T. B. Preuer, and R. G. Spallek, "Next-generation massively parallel short-read mapping on fpgas," in *ASAP 2011 - 22nd IEEE Inter-*

national Conference on Application-specific Systems, Architectures and Processors, pp. 195–201, Sept 2011.

[37] H. A. Shah, L. Hasan, and N. Ahmad, "An optimized and low-cost fpga-based DNA sequence alignment - a step towards personal genomics," in *35th Annual International Conference of the IEEE Engineering in Medicine and Biology Society, EMBC 2013, Osaka, Japan, July 3-7, 2013*, pp. 2696–2699, 2013.

[38] C. W. Yu, K. H. Kwong, K. Lee, and P. H. W. Leong, "A smith-waterman systolic cell," in *Field Programmable Logic and Application, 13th International Conference, FPL 2003, Lisbon, Portugal, September 1-3, 2003, Proceedings*, pp. 375–384, 2003.

[39] F. A. Khan, Aurangzeb, and Z. A. Khan, "Dna sequence matching system based on hardware accelerators utilized efficiently in a multithreaded environment," in *2009 Third International Conference on Electrical Engineering*, pp. 1–6, April 2009.

[40] R. J. Lipton and D. Lopresti, "A systolic array for rapid string comparison," 01 1985.

[41] Y. Hu and P. Georgiou, "A study of the partitioned dynamic programming algorithm for genome comparison in fpga," in *2013 IEEE International Symposium on Circuits and Systems (ISCAS2013)*, pp. 1897–1900, May 2013.

[42] K. Benkrid, Y. Liu, and A. Benkrid, "A highly parameterized and efficient fpga-based skeleton for pairwise biological sequence alignment," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 17, pp. 561–570, April 2009.

[43] M. Lizana, *Linaro for the Zybo board - an extended guide to run the Linaro OS on the Zybo Board*, 2016. `http://mariolizanac.com/posts/linaro-for-the-zybo-board`, accessed 29 November 2017.

[44] T. Skibo, "Freebsd on zynq-7000 / zybo / zedboard." `http://www.skibo.net/zedbsd/`, accessed 29 November 2017.

[45] S. Takamaeda-Yamazaki, "Setup flow of debian linux on zynq." `https://github.com/PyHDI/zynq-linux`, accessed 29 November 2017.

[46] Xillybus Limited, "An fpga ip core for easy dma over pcie with windows and linux." `http://www.xillybus.com/doc/xilinx-pcie-principle-of-operation`, accessed 8 March 2017.

[47] Xillybus Limited, "Product brief." `http://www.xillybus.com/downloads/xillybus_product_brief.pdf`, accessed 31 March 2018.

[48] P. Faes, B. Minnaert, M. Christiaens, E. Bonnet, Y. Saeys, D. Stroobandt, and Y. Van de Peer, "Scalable hardware accelerator for comparing dna and protein sequences," in *Proceedings of the 1st International Conference on*

Scalable Information Systems, InfoScale '06, (New York, NY, USA), ACM, 2006.

[49] P. Zhang, G. Tan, and G. R. Gao, "Implementation of the smith-waterman algorithm on a reconfigurable supercomputing platform," in *Proceedings of the 1st international workshop on High-performance reconfigurable computing technology and applications, HPRCTA 2007, held in conjunction with SC07, Reno, Nevada, USA, November 11, 2007*, pp. 39–48, 2007.

[50] N. Topham, "Inf3 computer design, practical 2, ssd driver module." `http://www.inf.ed.ac.uk/teaching/courses/cd/Practicals.html`, accessed 31 March 2018.