TECHNICAL UNIVERSITY OF CRETE

DIPLOMA THESIS

Hardware Acceleration of Genome Assembly Algorithms

Author:
Georgios GALANOS

Committee:
Professor Apostolos DOLLAS (Supervisor)
Professor Michail ZERVAKIS
Dr. Georgios KOTOULAS (HCMR)

A thesis submitted in fulfillment of the requirements for the diploma of Electrical and Computer Engineering

in the

School of Electrical and Computer Engineering
Microprocessor and Hardware Laboratory

September 13, 2021

Abstract

Georgios GALANOS

Hardware Acceleration of Genome Assembly Algorithms

Η συστοιχία γονιδιωμάτων (genome assembly) είναι ένα πεδίο της βιοπληροφορικής που αναφέρεται στη διαδικασία λήψης μικρών μερών γενετικού υλικού και επανασύνδεσής τους, με διαφορετικές μεθόδους, προκειμένου να αναδημιουργηθεί η αρχική αλληλουχία από την οποία προήλθε το DNA. Δεδομένου ότι τα σύνολα δεδομένων εισόδου των DNA έχουν πολυάριθμο μέγεθος και στις περισσότερες περιπτώσεις αποτελούν πολύ μεγάλο όγκο δεδομένων, είναι σημαντικό να εφαρμοστούν λειτουργίες και αλγόριθμοι προκειμένου να επιτευχθούν σημαντικές μειώσεις χρόνου και χώρου όσον αφορά την πολυπλοκότητά τους. Το φίλτρο ανάγνωσης (Read Matching Filter - RMF), το οποίο υλοποίησα και παρουσιάζω σε αυτή τη διπλωματική εργασία, είναι ένα είδος αυτών των διαδικασιών και έχει τον ρόλο της προεπεξεργασίας (φιλτράρισμα) των δεδομένων εισόδου στην διαδικασία του genome assembly.

Το RMF παίρνει το σύνολο δεδομένων εισόδου που περιέχει το γενετικό υλικό διαχωρισμένο σε μέρη που ονομάζονται reads, ένα ανά γραμμή και εφαρμόζει μια διαδικασία αντιστοίχισης μεταξύ τους προκειμένου να βρεθεί αχρησιμοποίητος πλεονασμός. Όταν η διαδικασία εκτελεσθεί επιτυχώς, ο αχρησιμοποίητος πλεονασμός εξαλείφεται από το σύνολο δεδομένων και στην έξοδο παράγονται τα τελικά reads τα οποία ονομάζονται ενδιάμεσα (intermediate) contigs. Το τελικό αρχείο εξόδου έχει λιγότερα σε αριθμό και μεγαλύτερα ή ίσα σε μήκος reads σε σχέση με αυτά του συνόλου δεδομένων εισόδου, αλλά χωρίς τον αχρησιμοποίητο πλεονασμό και με αυτόν τον τρόπο το συνολικό μέγεθος του συνόλου δεδομένων γίνεται μικρότερο. Αξιοποιώντας αυτό το αποτέλεσμα, η διαδικασία του genome assembly λαμβάνει ένα μικρότερο σύνολο δεδομένων ως είσοδο και ως αποτέλεσμα κερδίζει ένα όφελος χρόνου στην διαδικασία εκτέλεσης.

Ο παραπάνω αλγόριθμος εφαρμόστηκε τόσο σε λογισμικό όσο και σε σχεδιασμό λογισμικού-υλικού σε Field Programmable Gate Array (FPGA) προκειμένου να επιταχυνθεί ο χρόνος εκτέλεσης. Οι έξοδοι του RMF και το αρχικό σύνολο δεδομένων εισόδου δίνονται ως είσοδος στο Velvet genome assembler το οποίο βασίζεται στον χειρισμό των γραφημάτων de Bruijn, μέσω της αφαίρεσης σφαλμάτων και της απλοποίησης επαναλαμβανόμενων περιοχών, προκειμένου να επεξεργαστεί τη συναρμολόγηση και να δώσει τις ακολουθίες εξόδου. Συμπεριλαμβανομένου του RMF η διαδικασία του genome assembly κέρδισε μια ταχύτητα εκτέλεσης της τάξης του 2x-6x, με καλή ποιότητα στα αποτελέσματα μεταξύ των δύο μεθόδων.

Abstract

Georgios GALANOS

Hardware Acceleration of Genome Assembly Algorithms

Genome assembly is a field of bioinformatics that refers to the process of taking small fragments of genetic material and putting them back together by different methods in order to reconstruct the original sequence from which the DNA originated. As the DNA input datasets has numerous data size and in most cases has a very large amount of data, it is important to implement functions and algorithms in order to speedup these processes and gain significant time and space reductions in complexity. The Reads Matching Filter (RMF), which i implemented and present in this diploma thesis, is a kind of these processes and it has a preprocessing role in the whole genome assembly process.

The RMF takes the input dataset which contains the genetic material separated in reads, one per line and implement a matching process between each other in order to find unused redundancy. As the matching process executed successfully, the unused redundancy thrown out of the dataset and remain the output reads from the algorithm which they called intermediate contigs. The final output file that contains these intermediate contigs has less reads in number and bigger or equal than the input dataset's reads in length but without the unused redundancy and in this way the overall dataset size gets smaller. Exploited this result, the genome assembly process take a smaller dataset as input and as a result gain a time benefit in execution procedure.

The above algorithm implemented both in a software only and in a software-hardware design in Field Programmable Gate Array (FPGA) in order to gain an acceleration in execution time. The outputs of my design and the original input dataset are given as input in Velvet genome assembler which based on the manipulation of de Bruijn graphs, via the removal of errors and the simplication of repeated regions, in order to process the assembly and give the output sequences. The overall design included the genome assembly processing gained a speedup of the order of 2x-6x ratio, with good quality in the results between the two methods.

Acknowledgements

Throughout the writing of this thesis i have received a great deal of support and assistance. All the way from the first appointment in order to decide the diploma thesis theme until the presentation i had a big support from colleagues and family members.

I would first like to thank my supervisor, Professor Apostolos Dollas, whose expertise was invaluable in formulating the whole outcome planning of this thesis in order to finish the work and the whole presentation of my diploma thesis, the writing and visualizing of the results and the tips in presenting things with a more specialized look, the look of an engineer. His insightful feedback pushed me to sharpen my thinking and brought my work to a higher level. Also, i gratefully acknowledge the occupation of my committee's members, who, including Professor Apostolos Dollas, are Professor Michail Zervakis and Dr. Georgios Kotoulas (HCMR).

I would also like to thank my colleague to a great extent, a special partner on my work Mr. Pavlo Malakonaki. Pavlo, I want to thank you for your patient support and for all of the opportunities i was given to further my research, implement the job and produce the results. You provided me with the tools that i needed to choose the right direction and successfully complete my thesis. I was glad to work with you and learn from you.

In addition, I would like to thank my parents for their cooperation and assistance to join this amazing faculty and sympathetic ear. You are always there for me. Finally, I could not have completed this dissertation without the support of my sisters, close friends and acquaintances, who provided stimulating discussions as well as happy distractions to rest my mind outside of my research.

Georgios Galanos,

Heraklion 2021

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Bioinformatics is an interdisciplinary field that develops methods and algorithms in order to compute and execute biological data and tasks. As an interdisciplinary field of science, bioinformatics combines biology, computer science, information engineering, mathematics and statistics to analyze and interpret the biological data. Biological computation combines bioengineering and biology to build biological computers and algorithms, whereas bioinformatics uses to better understand biology. It is a fact that nowadays we have large amount of data to deal with for research propose, studies and innovations and we want useful tools to implement our job faster and better. In bioinformatics we have jobs such as genome processing, genome assembly and many others biological tasks, presuppose a very good space exploitation and a reduction in execution time as much as possible, to have a functional operational level and a good execution time. The primary goal of bioinformatics is to increase the understanding of biological processes. We have machine learning algorithms, pattern recognition, data mining, sequence alignment, protein structure prediction and clustering in families, visualization and so on, tasks which can be implemented with the contribution of the science of maths, software and hardware engineering and IT (Information technology) in general [18].

Biologists and more especially molecular ones, in the wide range of their work, use DNA, RNA, protein and other sequences, which in many cases are very large quantitative data. They have to deal with many bytes of sequencing data which need processing on computing clusters. Even if we assume the unusable redundancy this sequences may have, we can understand the increment of the time and space complexity of algorithms could have. Here is where genome sequencing take place and help by reconstructing the large sequences, by assembling the contigs into new one without any redundancy, by keeping the quality of the genome unanalyzed, as much as possible (e.g. error may occurred while reconstruction, repeats may exists). In this work we implemented a genome preprocessing filter which the main approach is presented in [39] and we make some changes to make it functional in a new FPGA to obtain a better time benefit, which is analyzed below. After that, a genome assembler can take the output contigs of our filtering stage and can construct new individual genomes. The processing time of the tasks of

a genome assembler can be considered exponentially increasing with the input file size and thus have been implemented methods which reduce the data size without missing valuable information.

1.1 My Thesis Contribution

In this section i will present the contribution of my diploma thesis theme among the above needs of increasing the efficiency and the effectiveness of the bioinformatics algorithms and programs. The design i implemented is a preprocessing technique and i present the implementation and the differences that we done among the various similar implementations. There are many techniques that can be implement in order to speedup the genome assembly processing and the genome sequencing in general. As DNA assembly technology cannot read whole genomes in one go, but rather reads smaller pieces, genome sequencing is a necessary task. The processing of genome assembly is a very time consuming, and as a result, expensive job. So, it is important to implement tools that assist the processing of genome reconstruction by filtering the data, which reduce the space complexity (RAM usage, hard drive space) as well as the processing time. In this paper we propose the use of reconfigurable hardware (FPGAs) in order to accelerate the execution of a pre-filtering process that removes the redundancy in a genome dataset. The Reads Matching Filter (RMF) executes a matching process between the reads of a DNA dataset in order to combine them and remove the redundant reads. As a result we have reducing on the assembly input dataset's size and the complexity of the assembly at all.

The implemented filter which acts as preprocessor generates an output that can be as input in Velvet genome assembler [34]. The basic logic about this RMF preprocessor is to takes the input reads, find for matches between the reads and generating the output contigs in order to give it as input in Velvet assembler to generate the final contigs. In contrast we pass the input reads as it is in Velvet assembler and taking the output contigs to check the similarity between the two outputs to examine the efficiency of the implementation. We found numerous speedups between the two methods and the general processing time of the assembly by Velvet reduced. The quality of the outputs seems to be satisfactorily in terms of targeting a state of the art FPGA platform that lead to a different architectural implementation in order to take advantage of the characteristics of the new platform, with respect to both I/O capabilities and hardware resources.

1.2 Thesis Outline

In this section we outline the organization of this thesis.

- **Chapter 2:** We describe in detail the theoretical background of the Genome Sequencing and Genome Assembly and we present the related work that exists in this field.

- **Chapter 3:** We intend to describe our implementation of the preprocessing genome sequencing program, both in software and hardware implementation and describe the differences between our's and paper's approach.
- **Chapter 4:** We present the results of our implementation.
- **Chapter 5:** We conclude this thesis and we provide directions for future work and possible extensions to our work.

Chapter 2

Theoretical Background and Related Work

In this Chapter, we will describe in detail the theoretical background of Genome Assembly and Genome Sequencing and we present the related work which has already been done both in research and the already existing bioinformating field.

As DNA sequencing technology works with big amount of data using bigger genomes and sequences, it needs algorithmic ways to implement aligning and merging techniques to the sequences in order to reduce the size of the read used or reconstruct genomes without loosing the quality of the output. This techniques follow algorithms of specific steps which are based on the match between consecutively reads in order to cover the unused redundancy between them. This can be done by many different ways which are listed below. The final result is a DNA sequence that it is in most cases unique or needing further processing in order to be unique (became a whole genome sequence). The result may contains faults or unwanted redundancy and in that case we can use other algorithms to prevent or fix these issues or we further processing it in order to reconstructed. These techniques cover the overall genome sequencing processing and all these approaches can help the genome assembly and many other biology tasks.

2.1 Genome

In the fields of molecular biology and genetics, a genome is the genetic material of an organism that present in the cells or in atoms. It provides all of the information the organism requires to function. It consist of **Deoxyribonucleic acid (DNA)**, a chain that is a molecule composed of two poly nucleotide chains that coil around each other to form a double helix. This two DNA strands are known as polynucleotides as they are composed of simpler monomeric units called nucleotides. Each nucleotide is composed of one of four nitrogen-containing **nucleobases (cytosine [C], guanine [G], adenine [A] or thymine [T])**.

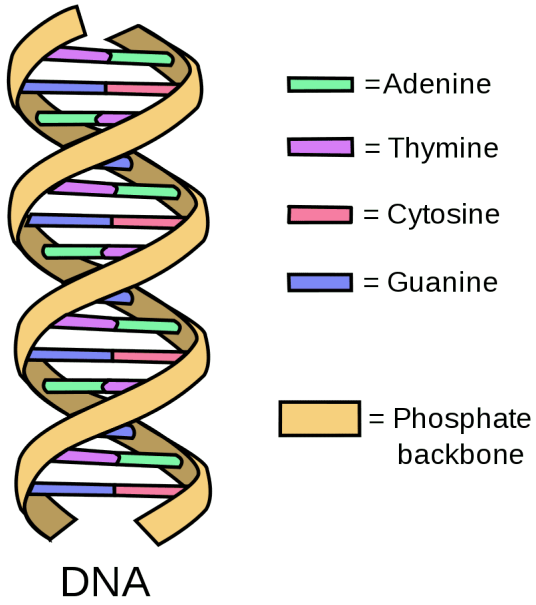

FIGURE 2.1: The DNA double helix with the nucleobases
https://www.yourgenome.org/sites/default/files/illustrations/diagram/dna_double_helix_yourgenome.png

The genomes can be extended, reconstructed and concatenate between each other in biological processes and create a very large sequence that needs assembly to reworked and deciphering the sequence composition of the genetic material (DNA), in order to exported new genome material for a new organism or a different version of the already existed organism.

2.2 Genome Processing

As we mention above, genome processing combined with genome assembly are the computational job of reconstruct genomes and assemble reads in order to generate new or make a deeper research about genome sequences of organism, bacteria, chromosomes and all the other genome stages that detailed described below. Genome processing started many years ago and here we present the general chronology.

2.2.1 Genome Processing Chronology

Genome Processing is a very old research field for the need of humanity to learn more about his health and for survival purpose. Besides of this, humanity want to learn more about the life, the creatures, the material and the whole world around. For that humanity's thirst of learning, the chronology of genome processing begins before 1900.

The very early 1871 Friedrich Miescher first publishes his paper where found the appearance of 'nuclein' (now known as DNA) and associated proteins, in

the cell nucleus [11]. In 1910, the first big step in genome processing happened, when Albrecht Kossel is awarded the first Nobel Prize in Physiology or Medicine for his discovery of the five nucleotide bases, adenine, cytosine, guanine, thymine and uracil.

Computers became essential in molecular biology when protein sequences first became available in the early 1950s. Then was the time when Frederick Sanger determined the sequence of insulin. Starting of this, comparing multiple sequences manually turned out to be impractical, so scientist try to figure out how to work on computer. After that, in 1953, James Watson and Francis Crick, with contributions from Rosalind Franklin and Maurice Wilkins, discover the double helix structure of DNA. In 1977, Frederick Sanger develops a DNA sequencing technique which he and his team use to sequence full genomes and invented the very first genome - that of a virus called phiX174 [14].

The need of software implementation, sequencing and assembling on computer, became more imperative decade-by-decade as in 1980 Frederic Sanger shares the Nobel Prize for Chemistry with Wally Gilbert and Paul Berg, for pioneering DNA sequencing methods which help on determination of the amino acid sequence of insulin, RNA and DNA sequencing [25]. In 1990, Human Genome Project is launched [31] (figure 2.2). The project aims to sequence all 3 billion letters of a human genome in 15 years. In 1999, Chromosome 22 is the first human chromosome to be sequenced as part of this project. The first try of determination of the human genome sequence released in 2001 and 2003 Human Genome Project is completed and confirms humans have approximately 20,000–25,000 genes. The human genome is sequenced to 99.99 per cent accuracy, 2 years ahead of schedule, in a very historic moment for biology. Additionally a recent update in this impressive project, on May 27 2021, the complete human genome sequence is close. Scientists from University of California, report that they have sequenced the remainder, in the process discovering about 115 new genes that code for proteins, for a total of 19,969. Researchers added 200 million DNA base pairs and 115 protein-coding genes — but they have yet to entirely sequence the Y chromosome, mentions Nature Portfolio [24].

The chronological development and the requirements of genome sequencing is constantly increasing and nowadays we have big amount of data for bioinformatics researches. As DNA assembly technology cannot read whole genomes in one go, but rather reads smaller pieces, genome sequencing is a necessary task. The DNA chains may consist of many bases and the processing in computers may be very difficult and expensive job. So, it is important to implement algorithms and tools that reconstruct the genomes (by throws the unused redundancy or errors) and reduce the space complexity (RAM usage, hard drive space) of biological processes.

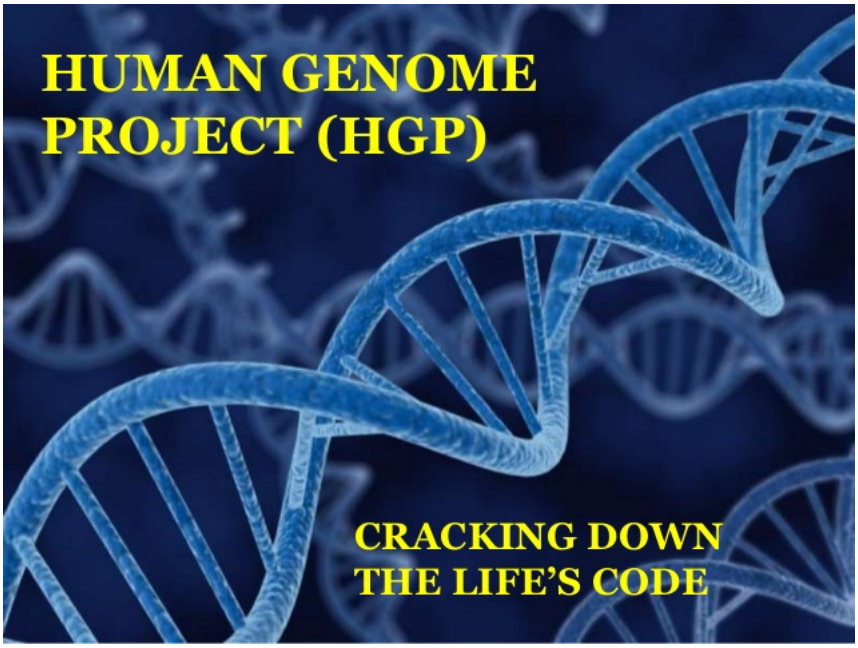

FIGURE 2.2: The Human Genome Project launched in 1990
https://www.slideshare.net/vinithasekar/human-genome-project-72272927

2.3 Genome Processing Programs and Tools

Many databases, dataset files, techniques and algorithm are used by scientists in genome processing processes. Many file types describes DNA sequences with different details and information level between them. These file types form datasets that include DNA material and can be used as input in genome assembly programs or output from them. Depending the read's length, the number of reads or the information that each dataset consist of, there are many assembly programs in order to reconstruct genomes. After the assembly stage further genome processing might happen. The output of the genome assemblers can be further processed and analyzed by sequence analyzers such as BLAST [3], tools that can calculate the similarity between multiple output sequences to define the measure of correctness of the result.

2.3.1 The shotgun sequencing technique

Most DNA sequencing techniques produce short fragments of sequence that need to be assembled to obtain complete genome sequences. The so-called shotgun sequencing technique and it is named by analogy with the rapidly expanding, quasi-random firing pattern of a shotgun [23]. The ends of these fragments overlap and, when aligned properly by a genome assembly program (such as Velvet), can be used to construct the complete genome. Shotgun sequencing yields sequence data quickly, but the task of assembling the fragments can be quite complicated for larger genomes. For a genome as large as the human genome, it may take large time frames of CPU time on large-memory, multiprocessor computers to assemble the fragments, and the resulting assembly usually contains numerous gaps that must be filled in

later. All the assembling methods that exist have big temporal and spatial complexity because the size of input data. The task must take all the reads, one by one and compare between each other for possible matches. Here is the hardware based acceleration that can be useful. Shotgun sequencing is the method of choice for virtually all genomes sequenced today, and genome assembly algorithms are a critical area of bioinformatics research to take shotgun sequences as input in order to implement the assembly. The majority of the input files which we used is a result of this shotgun technique.

2.3.2 Datasets and File Types

Especially in dataset and file formats we have different formats concerning the details we want to include in the specific file. There are many file types that contains and describes DNA sequences [29] [13] and the most widespread one is the FASTA (and FASTQ) file format in case we do not want to include additional information about the genome. This type of file is the simplest one and has a first line begin with the symbol '>' and can contains the title of the genome and some of the description, such as the type of data this file contains (e.g. shotgun generated reads or complete genomes). After that details line we have the genome reads, one per line with a various line length per dataset, most commonly of 60 or 70 bases per line and in some cases bigger line length (e.g. 80,90), but it is a fact that all the reads contained in a FASTA file has the same length per line. FASTQ is a similar file type to FASTA, but is has other information, nucleotide base calls, a second definition line, and per-base quality scores, all in text form. In our approach we use FASTA file format both as input and output of our algorithm.

2.4 Genome Assembly

One of the main task in genome processing is the genome assembly. In bioinformatics, genome assembly refers to aligning and merging fragments from a longer DNA sequence in order to reconstruct the original sequence. Is the computational step that follows sequencing with the objective of reconstructing the genome from its reads. There are two major categories of genome assembly, the reference-based mapping assembly and the simple de novo assembly.

2.4.1 Reference-Based Mapping Assembly

The first category assembling reads against an existing "template" sequence, build-in a sequence that is similar but not necessarily identical to the "template" sequence. If the genome has been sequenced before and a reference genome sequence already exists, then the newly obtained re-sequence reads are first mapped to the reference genome through alignment and then assembled in proper order. A revolutionary technique is the one mentioned above of Polymerase Chain Reaction (PCR)[22] which developed by Kary Mullis in

the 1980s. It is about a reference-based mapping sequence technique which is based on using the ability of DNA polymerase to synthesize new strand of DNA complementary to the offered template strand. In 2007 a new DNA sequencing technology is introduced [22] that increases DNA sequencing effectiveness. The new DNA sequencing process is simpler, more accurate and efficient than the multiplex PCR that was previously used. A microarray-based technique of genome sequencing machines which quickly determine the exact genetic code of the material, come to replace the previous PCR technique [28]. Another one aligner is Bowtie, an ultra fast memory-efficient short read aligner which aligns short DNA sequences to a reference genome at a rate of over 25 million 35-bp reads per hour [4].

2.4.2 De Novo Genome Assembly

The second category of genome assembly is the De novo sequence assemblers. De novo sequence assemblers are a type of program that assembles short nucleotide sequences into longer ones without the use of a reference genome. We got involved with a de novo genome assembly which has two common types of assemble programs; greedy algorithm assemblers and assemblers that construct a De Bruijn graph to represent their intermediate contigs. This assembler is Velvet [34] that we used it assembly our intermediate output contigs of our design. The input of the assembler is the intermediate contigs that our kernel generates. The main idea of our algorithm is that the pairwise alignment of all reads is done and the reads with the largest overlap is merged. This process is repeated till a single lengthy sequence is obtained. In this way we can give, as input in assembler, a smaller input file without loosing any significant information and so speeds up the processing time of the tool.

2.4.3 Genome assembler's implementations

Genome assembly from sequence reads is an algorithm-driven automated process. It is a computational expensive problem and for that reason there are many different techniques and methods in order to implement it. To date genome assembly can be done using one of the below three approaches:(1) greedy, (2) overlap-layout-consensus (OLC) and Hamiltonian path, and (3) de Bruijn graph and Eulerian path [17].

Greedy approach is the simplest, most intuitive, solution to the assembly problem. Individual reads are joined together into contigs in an iterative fashion, starting with the reads that overlap best, and ending once no more reads or contigs can be joined. This technique may failed under specific conditions that the contigs do not have any significant base coherence in common. In this situation the output contigs includes gaps. Paired-end sequencing is used to close these gaps. This technique allows the sequencing both of the ends of a fragment and generate high-quality, alignable data.

The second approach is the overlap-layout-consensus approach, which has 3 basic steps. The first step is using the greedy algorithm that described above, generating an output with the intermediate contigs that had been joined. In the second step it uses this output to construct an overlap graph; a graph containing each read as a node and an edge connects two nodes if an overlap was identified between the corresponding reads. The third and final step is the solution of the Hamiltonian path of this graph. Assembler try to find a single path that traverses each node in the overlap graph exactly once and generate and output which is a complete genome.

The third approach is that which uses the reads to construct a de Bruijn graph [6]. It can be recommended used in cases that the reads are short (<100bp). The main idea is that reads broken down to smaller sequences called k-mers. These k-mers are aligned using (k-1) sequence overlaps. The actual size of k depends on sequence coverage, read length, etc., but usually is not less than half of the actual read length. The final genome constructed by the Eulerian cycle method that visits every edge exactly once (allowing for revisiting vertices).

2.4.4 The hierarchical stages of the genome assembly and the N50 metric

Generally speaking the genome assembly is a multiprocessing job that includes different stages. Therefore, genome assembly is a hierarchical process; it is performed in steps beginning from the assembly of the sequence reads into contigs, assembly of the contigs into scaffolds (supercontigs) and assembly of the scaffolds into chromosomes. The most difficult assembly process is the assembly from the scaffolds into chromosomes because the gaps can not be easily sequenced. For that reason we have many assemblies remain restricted to scaffold level.

The quality review of the assembly is not so clear predefined. On the other hand, one very usual and useful metric in order to quantify the quality of the assembly is the N50 value. The N50 contig value can be determined by first sorting all contigs in decreasing order of size, then adding the contigs until the total added size reaches at least half of the total size of all assembled contigs. The size of the smallest contig used in this addition process represents the N50. The assembly processing can be executed multiple times until we reach a maximum N50 value. The larger the N50 value, the better is the assembly. Using the same concept, higher values of N are also used, such as N60 (until reach the 60% of the assembled contigs) and N80 (until reach the 80% of the assembled contigs). If the N50 scaffold length is too short, additional rounds of shotgun sequencing are recommended.

2.4.5 Genome assembler programs

Occasionally there were many software programs and tools that implement genome assembly processes and nowadays there is a very big occupation from the researchers and programmers to make faster and better programs and tools for this tasks. We have many assemblers that can make the reconstruction of the genome from its fragments. Our approach takes either the shotgun sequencing dataset or any other datasets, with the scattered parts of the genome (input reads), make the match between similar reads and throws the presented redundancy. The output intermediate contigs of our filtering stage, given as input in a genome assembler program in order to match the contigs to construct the whole genome (or new assembled contigs). We had a wide variety of these programs to use as well as most of them are open to use.

As we mention above, there are three main ways to implement a genome assembler and in that three ways we have the grouping of them, having as criterion and the intermediate contig's length. According to article [42] we have the figure 2.3 of genome assembly programs and its features. We have in column 'Algorithm' the way each assembler implemented and some others information such as the programming language they are implemented, the required read length and if they works with single-end, paired-end or both, reads. In our case we had single-end reads with numerous read length in FASTA input formats and so we try to use Velvet, SOAPdenovo [33] or SPAdes[2] (which is not included in the figure) and finally we keep the Velvet which compiled faster, it was easiest to use and easiest to pass the arguments we want. Apart from this, we wanted to use the same assembler with the motivated paper's one in order to compare the results between them.

2.4.6 Software Tools

After the genome assembler gives an output of complete genome from our intermediate contigs as inputs, we have a big variety of sequence alignment tools, such as BLAST [3] to farther analyze the results. These tools takes multiple sequences and try to find the biggest matching rate between them by shift the sequences. This task can be very useful if we take the output from genome assembler (as they generating by genome assembler) and put it in sequence analyzer, alongside with the output of the genome assembler by using the intermediate contigs of our design as input. This match rate can be a satisfying criteria of similarity, in order to decide if our intermediate contigs where right. The match rate is calculated in a reward/penalty ratio and depending on this we can assume a percent of similarity. An average percent of similarity above 95% is a very good result.

Program	Algorithm	Programming Language	Running Platform	Required read length	For Single-end reads?	For Paired-end reads?	Exploit Quality-value?	Input file format	Download Website
SSAKE (V3.5)	Greedy-extension	perl	*	25-36nt	Y	Y	N	Fasta/Raw	http://www.bcgsc.ca/platform/bioinfo/software/ssake
VCAKE (V1.0)	Greedy-extension	perl	*	<40nt	Y	N	N	Fasta/Raw	http://sourceforge.net/projects/vcake/
QSRA	Greedy-extension	C++	Unix/Linux	<40nt	Y	N	Y	Fasta/Raw	http://qsra.cgrb.oregonstate.edu/
SHARCGS (19-Nov-07)	Greedy-extension	perl	*	25-40nt	Y	N	Y	Fasta/Raw	http://sharcgs.molgen.mpg.de/download.shtml
Edena (V2.1.1)	OLC	C++	Win/Linux	N/A	Y	N	N	Fasta/Fastq	http://www.genomic.ch/edena.php
Velvet (V0.7.59)	De Bruijn	C	Linux/Mac OS X/Cygwin	N/A	Y	Y	N	Fasta/Fastq	http://www.ebi.ac.uk/~zerbino/velvet/
SOAPdenovo (V1.04)	De Bruijn	C	Linux/Mac OS	N/A	Y	Y	N	Fasta/Fastq	http://soap.genomics.org.cn/soapdenovo.html
Taipan (V1.0)	Hybrid algorithm	C	Linux	N/A	Y	N	N	Raw	http://sourceforge.net/projects/taipan/

FIGURE 2.3: Features of selected short reads assembly programs.
https://doi.org/10.1371/journal.pone.0017915.g011

2.5 Related Work

In related work we can find numerous bibliography about the innovations and the additional study that has been done in bioinformatics field. These works helped and continue to help the ever-increasing technology in biology and medical science, revealing and improving techniques and algorithms. In these many researches, our institution, the Technical University of Crete, has involved with many interesting works.

First of all there is the work in [8] where Grigorios Chrysos et. al. presents an in-depth look of how FPGA computing can offer substantial speedups in the execution of bioinformatics algorithms, with specific results achieved to date for a broad range of algorithms. The main conclusion is that FPGAs with the programmable logic they have, can be a significant tool to make bioinformatics algorithms works faster. Examples and case studies are presented for sequence comparison (BLAST, CAST), multiple sequence alignment (MAFFT, T-Coffee), RNA and protein secondary structure prediction (Zuker, Predator), gene prediction (Glimmer/GlimmerHMM) and phylogenetic tree computation (RAxML), running on mainstream FPGA technologies as well as high-end FPGA-based systems (Convey HC1, BeeCube).

In [16], Matina Lakka et al., presents the implementation on FPGA of two of the best known Multiple Sequence Alignment (MSA) algorithms, which offer high accuracy and great performance, T-Coffee and MAFFT. This paper presents the implementation of these algorithms on present-day FPGAs.

They conclude that for “large” datasets their design is 4 to 5 times faster. For ”small” datasets they did not take good results but they assumed that for a large modern FPGA device they can have up to 15 parallel their designs, thus achievings peedup from 10 to 55 times faster.

In advance we can mention the PhD dissertation on Reconfigurable Architecture Structures for the BLAST DNA Sequencing Algorithm, which presented in 2011 from Dr. E. Sotiriades in [27]. In a few words as he mentions Dr. Sotiriades, presents a system based on reconfigurable logic to implement the BLAST algorithm, regardless of data size or algorithm variation [3]. His design consists of software and hardware parts and achieves a speedup of several times up to thousands of times vs general purpose computers. We use the Blast sequence alignment in our study to check the efficiency of our algorithm.

In May of 2014 in [5], Chuming Chen et. al. developed ngsShoRT (next-generation sequencing (NGS) Short Reads Trimmer), a flexible and comprehensive open-source software package written in Perl that provides a set of algorithms commonly used for pre-processing NGS short read sequences. They compared the features and performance of ngsShoRT with existing tools: CutAdapt, NGS QC Toolkit and Trimmomatic. They also compared the effects of using pre-processed short read sequences generated by different algorithms on De-novo and reference-based assembly. Their results show that across three organisms and three sequencing platforms, trimming improved the mean quality scores of trimmed sequences. Using trimmed sequences for De-novo and reference-based assembly improved assembly quality as well as assembler performance.

Nathaniel McVicar et al., in [20], present a flexible and fast FPGA-based short read alignment tool. Their aligner provides a speedup of 5.6x over BWA-SW with energy savings of 21%,while also reducing incorrect short read classification by 29%.They also offer optimizations with which the speedup can be increased to 71.3x, while still enjoying a 28% incorrect classification improvement and 52% improvement in unaligned reads.

Another one task that we may take into account is the error correction that must done after the sequence’s generation. In April of 2010 we have a Parallel Algorithm for Error Correction in High-Throughput Short-Read Data on CUDA-Enabled Graphics Hardware where they present a scalable parallel algorithm for correcting sequencing errors in high-throughput short-read data with numerous speedups in their results [26]. They proesent that by using a CUDA-enabled mass-produced GPU, their results are in speedups of 12-84 times for the parallelized error correction, and speedups of 3-63 times for both sequential preprocessing and parallelized error correction compared to the publicly available Euler-SR program.

Finally, we can mention the growing involving of Microprocessors & Hardware Laboratory of TUC [21] which has present a numerous work on Genome Assembly and on Genome processing with many diploma thesis implemented of its students.

2.6 Scientific Contributions

During the 1950s, when the birth of modern molecular genetics happened, started the need of exploration of genes, heredity and the structure of DNA, the genetic information of all matter. In 1952, Alfred Hershey and Martha Chase proved that DNA was the molecule of heredity and James Watson, Francis Crick, Maurice Wilkins and Rosalind Franklin solved the three-dimensional structure of DNA with the double helix shape. From that point and till today, many scientific contributions have solve a big amount of the life questions, about what the genetic information of matter is and from what specific terms the life is configured and dependent.

Passing the years, one big problem was the inability to read genome sequences, because the large amount of the bases involved. Then, during the late 1960s and early 1970s, the combined work of several groups of researchers (Meselson & Yuan, 1968; Jackson et al., 1972; Cohen et al., 1973), helped in this problem by using DNA cut techniques at specific sites and spliced with DNA from other species. In that way started the mapping of genes where Mr. Francis Collins was the leader one, in 1980s, discovering the location of three important disease genes. After that, by the late 1980s, multiple approaches for sequencing DNA were in use and this all began to change with the work of National Institutes of Health (NIH) scientist J. Craig Venter. He started in his laboratory the genome sequencing by combine multiple sequencing techniques and he managed to sequence a big amount of genomes, with some missmatches, about 2000 whole genomes, which was as many as had been sequenced in the entire world to that point. Combine techniques seemed to work and was the start of what if follows [1]. Over the years, scientific discoveries that resulted from the application of next-generation DNA sequencing technologies, had their impact on the genome processing. Parallel platforms and new methods appear which works in a genome-wide scale with base precision and in this way these technologies brought enormous change in genetic and biological research. We have sequencing of RNA, serial analysis of gene expression (SAGE) and sequencing of ancient DNA samples as remarkable points in this point [19].

2.6.1 Acceleration of Algorithms

Many researchers has try to implement and accelerate algorithms and tasks of bioinformatic field. These implementations have much of them a hardware implementation and more specifically an FPGA implementation. A Field-programmable gate array board (FPGA) is an integrated circuit designed to be configured by a customer or a designer after manufacturing and give the benefit to the programmer to make his own spatial allocation and try to accelerate an algorithm with much of optimizations in the design. Many of the design techniques and more information about these works can be found on the "Architecture Exploration of FPGA Based Accelerators for BioInformatics Applications" book from Springer Singapore [38].

An FPGA based acceleration work done from Steven Derrien and Patrice Quinton [10]. They implemented a new parallelization scheme for the hmmsearch function of the HMMER software, which is used for searching sequence databases for sequence homologs and for making sequence alignments [15]. In advance, we have the publish of Parallel accelerators for GlimmerHMM bioinformatics algorithm from Nafsika Chrysanthou et.al. researchers of our institute, the Technical University of Crete [7]. They managed to take speedups up to 200x for the FPGA-based system and up to 34x for the GPU-based system for the most compute intensive part of the algorithm. In conclusion we can present the paper of Reconfiguring the Bioinformatics Computational Spectrum: Challenges and Opportunities of FPGA-Based Bioinformatics Acceleration Platforms from a group of authors, including my diploma thesis supervisor Pr. A. Dollas, where they conducts a detailed survey on the use of FPGA-based reconfigurable computing platforms for a wide range of sequence and structural bioinformatics applications, with emphasis on performance and energy savings of the underlying architectures [9].

2.7 Motivation and our approach

In my diploma thesis we studied many different techniques and ended up to the paper of FAssem : FPGA based Acceleration of De Novo Genome Assembly [39]. This general study refers to a FPGA based Acceleration of De Novo Genome Assembly and in this particular part refers to achieved speedups over software implementations using FPGA-based accelerators. They implemented an application that use a parallel hardware implementation to make a redundancy job in the reads in the input data and they build the consensus sequences from the outputs of this design by using the de Bruijn graph-based Velvet software. So they implemented a Redundancy Remover Unit (RRU) in FPGA which acts as preprocessor and generating the input of Velvet genome assembler. The basic logic about this RRU preprocessor is to takes the input reads, find for matches between the reads and generating the output in order to give it as input in Velvet software to generate the final contigs. In contrast they pass the input reads as it is in Velvet software and taking the output contigs to check the similarity between these two outputs to examine the efficiency of the implementation. This processes appear in figure 2.4 with the two data-paths generating the two outputs.

They implemented and ran their design in an Alpha-Data board having Xilinx Virtex-6 (XC6VSX475T) FPGA with speed-grade 1 [40]. They implemented a kernel of this accelerator and named it Process Element (PE). In order to take speedup they implemented a multi-instance of PEs. As they present they managed to fit a multi-instace of 15 PEs in this specific FPGA board and they present an estimation of the speedups up to 13x faster in terms of 300 PEs. They present generally estimated speedups with different multi-PEs instances with 30,300,1000 and 3000 PEs.

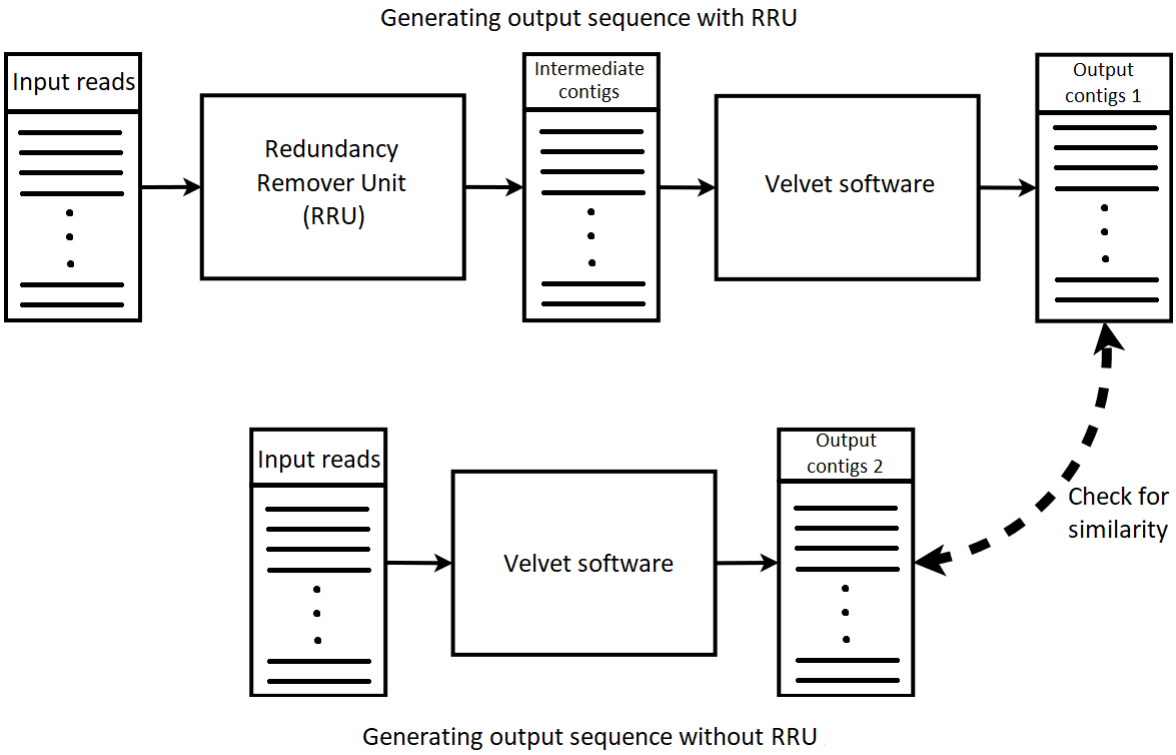

FIGURE 2.4: Generation of two outputs sequences.

The main differences with their approach has to do with the hardware implementation. The first main difference is that we implemented a single kernel design, on the contrary of their multi-PEs design, so lower level of parallelism but in conclusion we saw average similar speedups and output's similarity rates thanks to next generation FPGA board. The second difference has to do with the internal hardware logic where we eliminated a stage that we ended up that it does not give any advantage in the process. In order to evaluate the results, with respect to performance and results' quality, of our work we utilized Velvet as well. About these differences we speak more specific in Chapter 3 and 4.

2.7.1 Amdahl's law and theoretical parallel speedup

The theory of doing computational work in parallel has some fundamental laws that place limits on the benefits one can derive from parallelizing a computation work. To understand these laws, we have to first define the objective. In general, the goal in large scale computation is to get as much work done as possible in the shortest possible time within our budget. According to the Amdahl's law there is a upper limit about the gained speedup for each computational work. This is in almost all cases the best speedup one can achieve by doing work in parallel.

In our results that will be detailed presented in Chapter 4, we have the software execution times for our RMF design. For the processing stage of the RMF, which is the one that take the most execution time, we can compute

an upper limit of possible speedup. This stage is an independent stage from the I/O operations, throw the RMF datapath, does not include dependencies between variables and we can have more than one (1) instances of this for parallel processing. In this point of view we can assume that this processing stage is a parallelizable stage.

Assuming the execution time of the processing stage as Tp and the overall RMF execution time (including I/O operations) as To we can calculate the percent of the overall RMF's execution time, which the processing stage occupies and we have the following table 2.1 for our datasets:

Dataset	Size in Kb	Execution time in sec		Percent of processing stage
		Tp	To	
Pyruvatibacter	3291	86,71	87,27	99,3%
Pseudomonas	6689	174,8	175,96	99,3%
Aythya	11749	312,72	314,78	99,3%
Melopsittacus	22887	625,41	629,65	99,3%
Photinus Piralis	70234	1985,71	1997,87	99,4%

TABLE 2.1: The percentage of the overall RMF's execution time, which the processing stage occupies.

As we observe the percentage of the execution time of the most computational expensive part, the processing stage of the RMF, is about 99,3-99,4%. This means that there is no I/O bottleneck in RMF, as the processing stage is an independent stage from read and write stages. According to Amdahl's law if we assume the percentage of the processing stage as f the possible upper speedup that we can gain is:

$$1 \div (1 - f)$$

So, we have the below maximum theoretical possible speedups concerning these percentages:

Dataset	Size in Kb	Percent of processing stage	Maximum possible speedup
Pyruvatibacter	3291	99,3%	142,8x
Pseudomonas	6689	99,3%	142,8x
Aythya	11749	99,3%	142,8x
Melopsittacus	22887	99,3%	142,8x
Photinus Piralis	70234	99,4%	166,6x

TABLE 2.2: Maximum theoretical possible speedups.

This implementation can achieve significant speedups according to the previous profiling and as it can easily be configured to run in a modern FPGA, we concluded to implement this design in order to get the results using a next generation FPGA. As we see in Chapter 4, we finally achieve speedups

up to 45x for our initial implementation. The differences between the maximum theoretical speedup and the gained speedup lead to the conclusion that the RMF has room for improvement. On the other hand we gain significant speedups and both for RMF design and for the whole assembly processing.

After that first theoretical speedup calculation and as we take our first software only execution time measurements from the whole assembly processing (including Velvet), we calculate, in the same way with above, the overall maximum theoretical speedup that the process can achieve. First, assuming again as Tp' the execution time of the processing stage of the RMF and as To' the overall execution time of the whole process including RMF and Velvet, we have the following table 2.3:

Dataset	Size in Kb	Execution time in sec		Percentage of processing stage
		Tp'	To'	
Pyruvatibacter	3291	86,71	125,94	68,8%
Pseudomonas	6689	174,8	232,81	75%
Aythya	11749	312,72	516,96	60%
Melopsittacus	22887	625,41	1065,13	58%
Photinus Piralis	70234	1985,71	6354,03	31,2%

TABLE 2.3: The percentage of the entire assembly process' execution time if we do not account for the RMF.

As we observe the percent of the execution time of the processing stage throw the whole assembly process (RMF+Velvet), is different depending the file size. Like the previous calculations if we assume as f' these percentages and by taking the Amdahl's law we can calculate the maximum theoretical speedup that we can gain in the entire assembly process if we manage to implement the processing stage of the RMF in parallel implementation. These speedups are tabulated in the table, below 2.4:

Dataset	Size in Kb	Percentage of processing stage	Maximum possible speedup
Pyruvatibacter	3291	68,8%	3,2x
Pseudomonas	6689	75%	4x
Aythya	11749	60%	2,5x
Melopsittacus	22887	58%	2,38x
Photinus Piralis	70234	31,2%	1,45x

TABLE 2.4: Maximum theoretical possible speedups of the entire process.

So we can gain a possible maximum speedup up to 4x if we manage to parallelize the execution of the processing stage of the RMF. In our initial implementation, we finally managed to gain a maximum speedup up to 3,61x and by making further improvements in the design we manage to increase this speedup as we will see in Chapter 4. In these speedups we have include

all the I/O operations and the results are very encouraging in order to make sense to implement the RMF.

This speedup is a significant speedup in the whole assembly process and this drove us to implement RMF. Apart from this, the software implementations of these preprocessing stages like RMF, are very slow and not useful in the whole assembly process (and are not used from genome assemblers). For that reason the hardware implementation of them becomes necessary in order to executed in parallelism and by that way to reduce the execution time at all.

We are targeting a state of the art FPGA platform that lead to a different architectural implementation in order to take advantage of the characteristics of the new platform, with respect to both I/O capabilities and hardware resources. Apart of these, this is a design that can easy get optimizations and parallelism and we continue in the logical design implementation of this. The hardware implementation, which consist of subsystems easy to implemented in hardware, can easily synthesized and designed.

Chapter 3

Reads Matching Filter

In this chapter we intend to describe the implementation of the Reads Matching Filter (RMF) of our approach and the differences it have with the paper's one which we motivated. In section 3.1 we have the software implementation with the algorithm and the processes we made and in section 3.2 we have the hardware implementation of the most computing expensive stage of our filtering and all the optimizations that we made in order to speedup the process. In hardware implementation we used the Vitis unified software platform [36] from Xilinx [41], where we build our project, with the functions that reads the input file and writes in output file runs in software mode in the host and all the matching job between the reads done in hardware kernel on the FPGA. The general filtering divided into three main stages which they presented in detail below.

3.1 Software initial implementation

The Reads Matching Filter (RMF) is divided into three main stages. The first one is the preprocessing stage, where we read and store the input dataset in RAM, the second one is the process of the matching between all the reads and the last stage where the intermediate contigs generated. Here we can mention that because the length of the reads (60,70 bases per read), to have variables to store and process them, we can not use typical variables to store our data. We use Arbitrary Precision Data Types (ap_int.h) library and in this way we can manage the bit-width of the integer numbers within the boundaries of the specified width [37].

In the first stage we have to read the input file (FASTA format), line by line as reads, converting the DNA bases into binary form and store them in RAM in 128bits entries (ap_uint<128> type, where an 60-bases read stored in 120 MSBs (Most Significant Bits)), to take advantage of the access speed. It is more useful and much more faster to work on the RAM despite the hard drive and this helps us in this stage, on execution time. The conversion in binary form done according to the conversion formula of the bellow table 3.1.

This conversion is useful because considering of a DNA base (A,T,C,G), which is a 8 bit character variable, after conversion in 2 bits we reduce the space that

DNA base	Binary form
A	00
T	11
C	01
G	10

TABLE 3.1: The conversion formula of DNA bases

dataset occupies in RAM to 25 percent of the original space, because in this 8 bits variable can be stored 4 different bases. In this way an 60 bases read can be stored in 120 bits by this conversion against of $60 \times 8 = 480$ bits in character variables. Each read stored in RAM in 120 bits variables until no read remained in dataset.

The time this process done, we continued to the next stage, the matching stage. We take all the dataset's reads that had been stored in RAM and try to find matches between them in order to remove the presented redundancy. This stage returns 3 vectors of data for each dataset, 1 for left extensions, 1 for right extensions and 1 for the starters that took part in each comparison, included the number of bases that done the extensions. This process and the returned values of it will be further described below.

This part is the most expensive part in terms of CPU and RAM usage and for that reason it takes the most time of execution. If we suppose a dataset of N lines, we take N reads as starters and each of this starters matched with other N-1 ($\approx N$) reads in order to throw the redundancy process of the reads. This iterations phase, which consist of N^2 matching combinations in worst case ($O(N^2)$), can be accelerating from a FPGA implementation in order to redeem significant space and time reductions in complexity.

In the last stage we have the concatenation of the intermediate contigs from the returned values of the second stage above. From the second stage we take 3 banks of data, as we describe below, 1 for left extensions parts, 1 for right extensions parts and 1 for the starters that took place in the matching. The concatenation process take first the left extensions, afterward takes the starter of the matching and finally the right extensions to create the final intermediate contig that is written in output file. The left and right extensions of the each index of this output banks correspond a specific read-starter and comes from the comparison with this specific starter. The logic is that in left extension's bank of data, we have all the extensions of the comparisons of each time's starter and before we take the next available starter for the next iteration's matches, we write a '0' value in the last available index of the output in order to separate the current's starter left extensions from the next's one. The same logic follow the right extensions data bank. In this way we clarify which starter's is which extensions. The generated intermediate contigs written in output FASTA file. The starters that did not extended written as it is in the output file.

The 'contigs.fasta' file is the file that contains the created output intermediate contigs from our design. This file has contigs with similar material information with the input reads with the difference that they are less in quantity and greater or equal in (bases) length, but definitely less or equal overall file size from the input file. Reads that contains redundancy between them will have been eliminated and remained unique. This output file can be sent to assembler (like Velvet) in order to create new unique sequences considering error corrections and mismatches from input reads.

3.1.1 Using different HBM banks to store the data

One big problem was the fact that if the input dataset was big (e.g. more than 100MB - 100million bases), our design does not work and these sizes of input files is very common situation in genomes. This problem exists because the limitations of the high bandwidth memory (HBM) interface the FPGA has. The acceleration card that we used (and detailed described in Chapter 4) has 32 HBM banks of 256MB each, in order to store our data and in our first implementation, all the data (in/out) goes in the same bank (HBM[0]) by default. This bank after a number of iterations filled up, so we implement a different bank data save technique. As we will mention below we use 2 read-sets buffers in order to make a double buffering dataset technique (to manually reduce the number of iterations), 1 output buffer for left extensions, 1 output buffer for right extensions and 1 output buffer for the starters that extended (or not). This 5 ports assigned in 5 different HBM banks (HBM[0] to HBM[4], 256MB each) and so we overtook the space limitation problem of big datasets winning a small increase in speedup because of the different port's accesses. All these connections shows below in figure 3.1.

3.2 Hardware initial implementation

After the first stage of reading the input dataset in the host, follows the hardware implementation, the main design of our approach. We implement it in Vitis and synthesize it with the Vitis HLS to take information about run time, number of iterations per loop and space allocation. The top level function is the krnl_iterativeStage which consist of three different parts. The first one is a prefilter design which was implemented and removed finally for the reasons that we will describe them below. The second part is an extender design to search for coverages between starters and reads. The third stage is the write back stage where we store the information of left and right extensions and the number of bases that took part in the extensions, which they are the outputs of the hardware kernel, in order to construct the output intermediate contigs in host. The concatenation of the contigs done in host in software because the dynamic allocation of memory that is necessary.

As we mention above, the top level of our kernel is the krnl_iterativeStage. The main design, as showed in figure 3.1, combines retrieving data from HBM[0] which is the reads_input data stream, a prefilter stage where a read

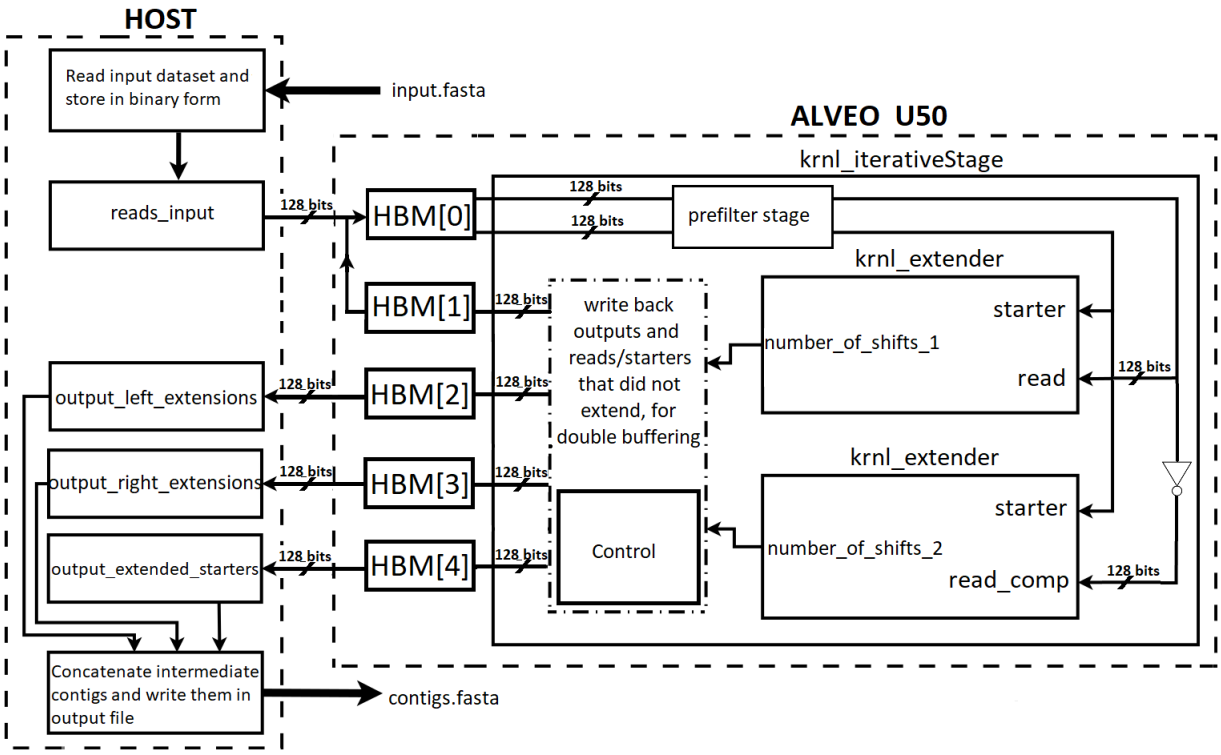

FIGURE 3.1: Our first design in general (host and kernel).

vector generation technique is implemented, a bitwise NOT operation of the read in order to take the read complementary strand (read_comp), the extender module (krnl_extender) in order to find the matches and the write back stage in the end, where the possible extension's data written in RAM via HBM[3]-5. The remaining reads that did not extend the each time starter, written in HBM[2] which is the reads_output stream for a double buffering technique that implemented for the dataset. All the ports of FPGA that communicates with the DDR are configured as m_axi interface which implements an AXi4-Lite interface. This interface defines a point to point communication between master and slave ends. The reads_input stream, in HBM[0] bank, has DDR as master and FPGA as slave in order to read the dataset's reads. The other 4 HBM banks used as outputs from FPGA so the top level of the kernel is the master and DDR is the slave.

3.2.1 The top level implementation

The steps we implemented in order to reach the functionality of the algorithm in the top level function of the kernel, described in the follow **Algorithm 1**

and further details presented below:

Algorithm 1: Iterations stage algorithm

```
Procedure krnl_iterativeStage(input read set R in binary format)
 coverage_factor = 0.1;
for each read s ϵ R as starter do
    number_of_shifts_1=0;
    number_of_shifts_2=0;
    for each read r ϵ R as read do
        prefilter_design(starter_left,read_left);
        prefilter_design(starter_right,read_right);
        if prefilter pass then
            number_of_shifts_1=extender(read, starter_left, starter_right,
             coverage_factor);
            number_of_shifts_2=extender(read_comp, starter_left,
             starter_right, coverage_factor);
        end
        if number_of_shifts_1!=0 or number_of_shifts_2!=0 then
            if left_extension then
                starter_left = read;
                update output_L stream with extension parts of read;
            else if right_extension then
                starter_right = read;
                update output_R stream with extension parts of read;
            end
        else
            write read in output read set for next iterations;
        end
    end
    if number_of_shifts_1 != 0 or number_of_shifts_2 != 0 then
        write s in output_ST stream as intermediate contig;
    end
    double buffering output read set in input read set R;
end
End Procedure
```

First of all we take every single line-read as a starter from the HBM[0] bank of data and the next available as a read. For each starter and read we implemented a vector construction in prefiltering stage. As we mention above, by represent each read in binary form with 2 bits for each base, in one byte we can store 4 bases which is 8 bits. This 8 bits represent a number between 2^0 and $2^8 - 1$ (0 and 255). We can keep information about all of the octaves (4 bases) that read or starter consist of by set an '1' in a 256-bits vector (ap_uint<256>) of each read. For example, as we can see in figure 3.2, if a read has the four bases 'ATAG' which in binary form (according with the conversion formula of table 3.1) is the '00110010', this octave of bits represent the decimal number 50, so we can store an '1' at index 50 of this read's vector to keep the information that this read has this octave of bits, so the

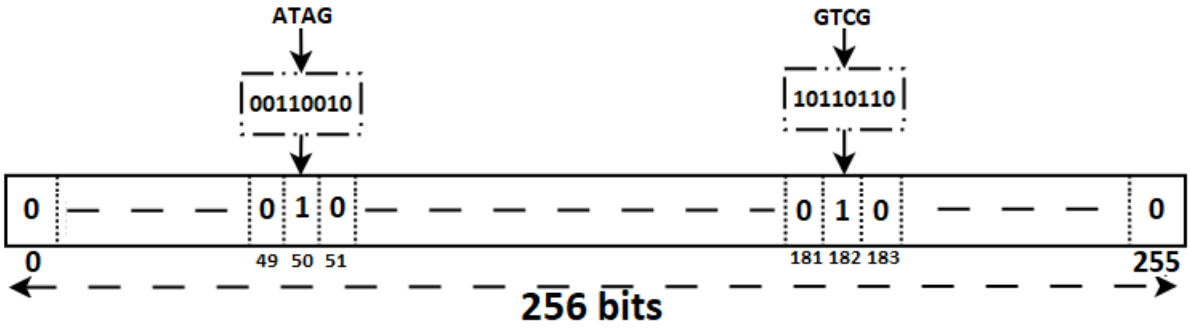

FIGURE 3.2: Read vector construction.

read has the four bases 'ATAG'. In the same way the four of bases 'GTCG' which represented by bits '10110110', is the decimal 182, so we mark as '1' in the corresponding position of the read_vector. After each 8 bits (4 bases) we shift the read by 2 bits and we take the next 8 bits and by taking the decimal number of the octave of bits we represent them as '1' in the vector of this read or starter. This vectors used in the prefilter stage which described below.

Before calling the extender we have the prefilter design where we define if the starter and the read have at least some group of bases in common, hence there is not any chance to find any coverage. This is the main idea of this prefilter module. More specifically, a logic bitwise AND between starter's vector and each read's vector defines the common '1', so the common octaves of bits, so the common four of bases, that starter and reads can have. If this result has not any '1' in common means that the starter and read do not have bases in succession so they can not have any important coverage and they are not sent to the extender. In this way we theoretically can overtake a starter-read matching process that it will not offer any significant matching and we can reduce the processing time.

3.2.2 Removing read vector construction and prefilter stage

This prefilter stage of creating the read's and starter's vectors and the comparison between them in order to find the possible common fours of bases, we came up after a lot of experimentation that we can skip it. Removing it, we measured 2x-3x speedup on hardware execution time and the same outputs compared with the design which includes this prefilter stage. The technique that we follow to decide whether keep this stage or skip it, has to do with the possible fours of bases threshold we had set, which is the number of bases that the starter and the read must have in common in order to send them in extender. We noticed that changing this threshold in multiple values, the outputs did not change even if we set it 0 (0 threshold in this stage means that prefiltering stage is disabled). Taking into account all of the above we skip the vector prefiltering stage and exploiting this we took better processing time in FPGA as it seems in below table 3.2.

Datasheet	Size (in bytes)	Kernel execution time with prefiltering stage (in sec)	Kernel execution time without prefiltering stage (in sec)
Staphylococcus	4751KB	6,72 sec	2,18 sec
Streptosporangium	19305KB	27,68 sec	9,02 sec
Drosophila melanogaster	40021KB	57,52 sec	18,67 sec

TABLE 3.2: Changes in kernel execution time in FPGA with or without prefiltering stage

We used many of datasets in order to measure execution time both included and not the prefilter stage and some of these datasets are a Stapfylococcus strain, a Streptosporangium strain and a Drosophila melanogaster strain. We can draw the conclusion that the execution time takes an average 3x speedup between the first implementation including the prefiltering stage and the next implementation which this stage eliminated. The outputs of the two designs has a 100% of similarity and we decided to skip this prefiltering process at all.

3.2.3 The complementary DNA strand

Alongside with the read we calculate the twin DNA strand (read_comp) of this read to see if this can extend the starter. The twin DNA strand is the abreast nucleotide of the DNA chain and this is the complementary read with the first one. The base at a given position in one strand is related to the base at the corresponding position in the other strand of the double helix of the DNA by the following base-pairing rule (referred to as "base complementary"): A⇔T, C⇔G as it shows in figure 3.3. Any of the two nucleotide can extends the starter in the same way.

We generated the read_comp with a reverting bitwise NOT of the read, the time before calling the extenders and we call the two extenders pipelined. The first extender instance take the read and the starter and the second extender instance take the complementary read with the starter and they try to find matches. This execution implemented in this point by declare different registers for the arguments of the two extenders to retrieve the data of each one from different location, in order to work the two functions calls pipelined. Starter controlled for matches with the read and the read_comp in the same time (pipelined) and at the end of this two extender function runs, we take two return values about the coverage of each extender, the number_of_shifts_1 and the number_of_shifts_2.

The read_comp generated by inverting the bits of the read. We done the assignment of bits statements by keeping into account this specific conversion. As we mention the base A (00) converted in base T (11) and the base C (01) converted in base G (10) and vice versa. So a bitwise NOT of the read is the read_comp as it seems in figure 3.1. Because of the read length of 60 bases, the bases are in the 120 MSBs of the read (128 bits variable), so the 8

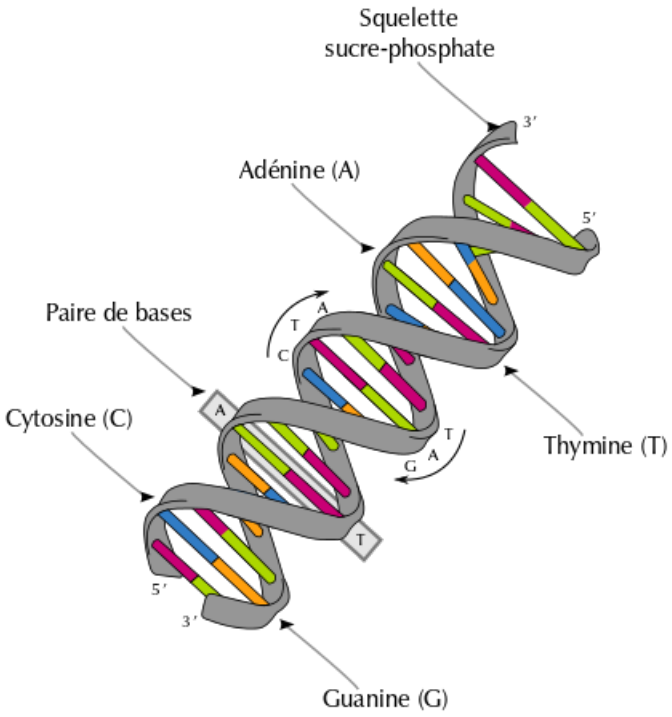

FIGURE 3.3: The double helix of the DNA: the two twin DNA strands.
https://commons.wikimedia.org/wiki/File:DNA_structure_and_bases_color_FR.svg

LSBs that was '0' converted in '1' from the NOT operation and they do not represent any valuable information. As we will see in the write back stage below we want these 8 bits to be '0'. To overtake this problem we make a logic bitwise AND between the read_comp and a mask of '1's in the 120 MSBs and '0' in the 8 LSBs.

3.2.4 Extender implementation

The next step is the extender stage where the starter and each read (or read_comp as we described above) checked for possible matches between them. Before describe the functionality of this module we must describe the registers that we used and the data that they contain. First of all the starter can be extended by the read from both of his sides, left or right. If a read extends the starter from a side, this read must be the next starter in this side. For example as we can see in figure (**edw**) if the starter left extended by the read from the left, the read that done the extension must be the next round's left starter as this is the most left part of the intermediate contig. For that reason we have to declare two registers for each side so we have a starter_left and a starter_right, which in the first round they both take, the value of the starter. After an extension occurred on some side, this side's starter register will be replaced by the read that done the extension.

As we mention above, the read can extend the starter from both of its sides. For that reason we have the shifted read_left and the shifted read_right registers. The starter_left checked with the shifted read_left for matches in order to define if we have an extension from the left side of the starter. The same operation done between the starter_right and the shifted read_right for the right side respectively. We named the read_left and the read_right as shifted registers because in case of no extension, we shift these register left or right by 2 bits respectively. By that way we shift the read by one base from each side and in the next iteration we check if the read can be extends the starter.

All the datapath of the extender can be visually separated in left and right side and further details described below. The extender described in **Algorithm 2** and the hardware block diagram is showed in figure 3.4.

Algorithm 2: Extender algorithm

```
Procedure extender(read,starter_left,starter_right, threshold)
  max_shifts= read_length - threshold x read_length;
for shifts=0; shifts<max_shifts; shifts+=2 do
    match_L=read_left XNOR (starter_left AND mask_L);
    match_R=read_right XNOR (starter_right AND mask_R);
    scoreL=krnl_modifiedCounter(match_L);
    scoreR=krnl_modifiedCounter(match_R);
    if scoreL > (threshold x read_length) then
        // left extend
        return (0-shifts);
    else if scoreR > (threshold x read_length) then
        // right extend
        return shifts;
    else
        shift read_left « 2;
        shift mask_L « 2;
        shift read_right » 2;
        shift mask_R » 2;
    end
end
// no extensions
return 0;
end Procedure
```

At the beginning we take the starters left and right and the read (or read complementary strand) and assign to local registers starter_left, starter_right and the read to local shift registers shifted read_left and shifted read_right that will take place in the comparison. Starter left will be compared with the read left and the starter right with the read right respectively. In the end of the iteration read left shifts left by 2 bits and read right shifts right by 2 bits if the comparison failed. By these shifts we want 2 masks in order to correspond the bits of the starters that take place in the comparison in each iteration. These masks initialized, by the starter's length, with ones ('1') and

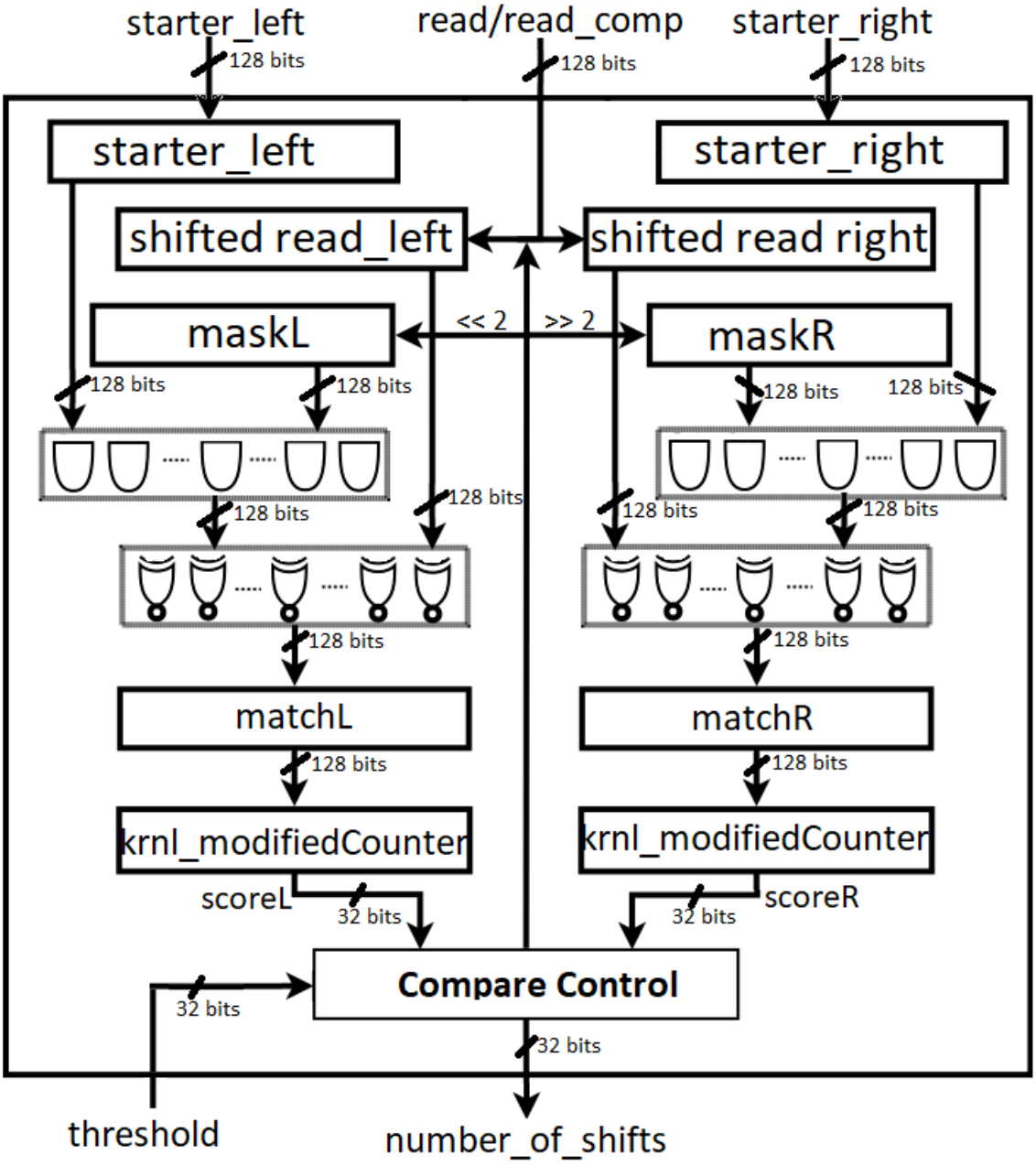

FIGURE 3.4: The extender design.

shift as the reads shifts. A bitwise logic XNOR between read and the output of the bitwise AND of the starter and the mask, gives a result of common '0' and '1' starter and read have in each round. XNOR gives '1' in each index where the 2 bits are the same. The results of these comparisons stored in 2 registers matchL and matchR for the next step. So the logic equations that take place in order to find the matches are the following:

$$matchL = read\_leftXNOR(starter\_leftANDmaskL)$$

$$matchR = read\_rightXNOR(starter\_rightANDmaskR)$$

After that a counter (krnl_modifiedCounter) counts the recurring '1's starting from the left side of the matchL and from the right side of the matchR and returns in variables score_L and score_R the decimal number of recurring '1's. This counter counts up by one if a double '1' occurs as this means that starter and read have one base (2-bits) in common. In order to have a coverage between starter and read we must have a recurring matching of bases in the matchL and matchR from each side.

Subsequently, we check the scoreL and scoreR values that the previous counter returns. The scoreL correspond the number of bases that matched from the comparison between starter_left and read_left and the scoreR from the comparison between starter_right and read_right. We check either scoreL or the scoreR if is greater than possible bases extension threshold which is has been set and we describe it below. These comparisons done in the Compare Control module. If scoreL is greater that this threshold and greater than scoreR, we have a left extension and respectively if scoreR is greater that this threshold and greater than scoreL, we have a right extension. The match is accepted only with a 10% of coverage factor between starter and read. This threshold means that if we have a 60 bases starter, we accept a coverage bigger than 6 bases.

In case we do not have any coverage, so scores equal to 0 or less than threshold, we shift the reads, right or left, according to each side. We shift left the read_left and shift right the read_right by two because we have a binary form of the bases that means 2bit representation. These shifts and matches are done simultaneously and the loop ends after max_shifts iterations. Max_shifts is the upper bound of the loop and it is the max possible coverage that we can have between starter and read. It is equal with the max starter's length minus the threshold of possible extension bases. For example, if we suppose starter_length=120 bits (60 bases) and a threshold of 10% bases coverage, we have $max\_shifts = 120 - (120 \times 0.1) = 108$. So as we shift by 2 we have maximum of 108 iterations(shifts). The equation is presented below:

$$max\_shifts = starter\_length - (starter\_length \times base\_threshold)$$

After max_shifts if we do not have any extends, the extender returns 0. If any of the extension is accepted (either left or right), extender returns the number_of_shifts that is calculated by the current shifts counter and gives as output $0 - shifts$ for left extension or $shifts$ for right extension to have the information that we have left extension -negative return- or right extension -positive return- and how many bases done the extension, which is the number of shifts.

3.2.5 Write back stage

Finally in the last stage of our hardware kernel we have the decoding of the outputs of the extenders and the write back process of writing the outputs in output HBM banks as we can see in figure 3.5. First of all we take the two

returned values from extenders, one from the comparison between the starter and the read (number_of_shifts_1) and one from the comparison between the starter and the twin strand of the read, the read_comp (number_of_shifts_2). As we mention, these values can be negative integers for left extensions, positive integer for right extensions or zero in case of no extension and they correspond the number of shifts that extender did, until it finds a math. If these two values are both 0 means that neither read or read_comp extends the starter, because extender reach the maximum of shifts bound (which described above) without coverage found and returns 0. In this case the output register written with the read that did not extend the starter, via MUX (select_1 to '00' by the control) and the control select to send the output via 00 of the DEMUX to HBM[1] (reads_output) to be a part of double buffering of the dataset for further processing in the next rounds.

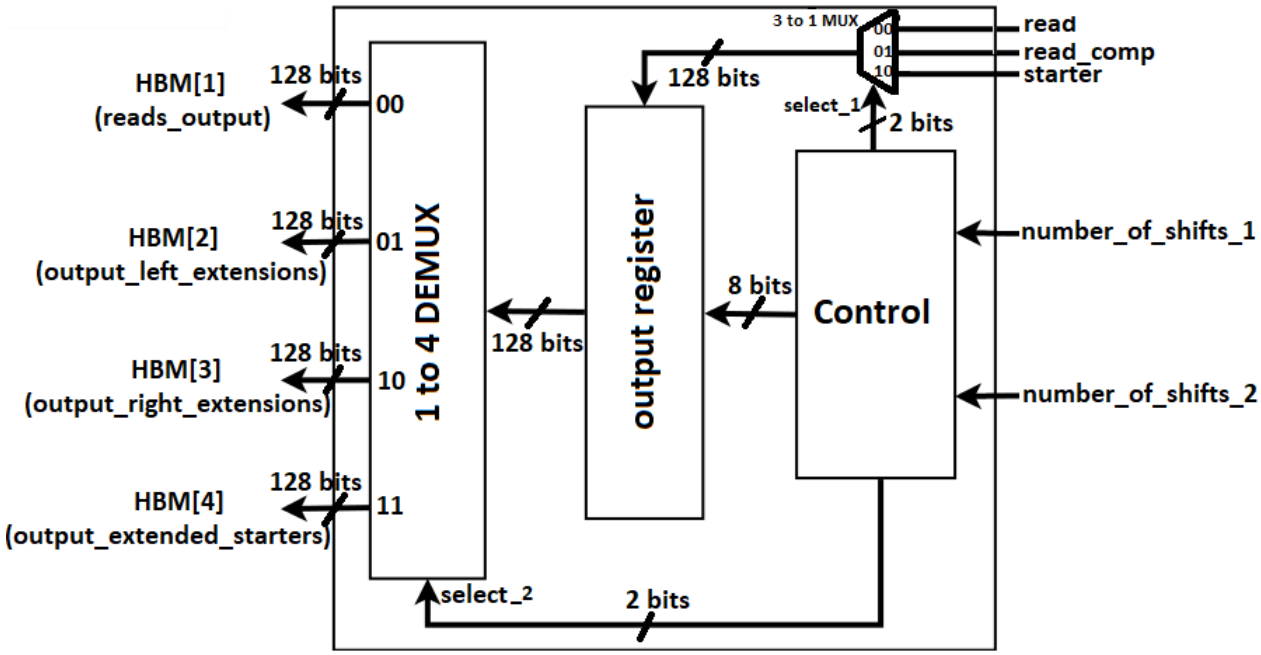

FIGURE 3.5: The write back stage of our kernel.

If either read or read_comp extend the starter we have to write back to DDR the data of the extension via HBM[2]-4 ports. With the data ready to be written, the control decide if there was a left extension or a right extension by checking the positive or the negative sign of the shifts corresponding values. First, the control find the (absolute) smallest, but non zero, number of shifts from the two extenders, as the smallest value means the smallest number of shifts, so bigger coverage between starter and read or read_comp respectively. If the smallest return value is the return value from extender with the read control keep the read to write in output register, via MUX and select_1 to '00' or if the smallest return value is from the extender with the read_comp, control keep the read_comp to write in the output register, via MUX and select_1 to '01'.

The return value of the extender as we mention above is the number of shifts that done, so is an information that we must to know in the host in order to

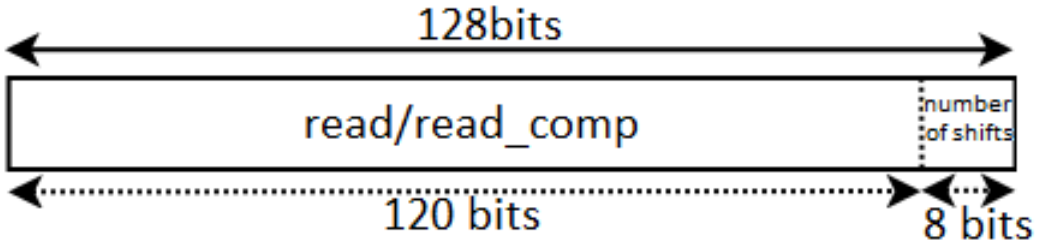

FIGURE 3.6: The output register of the write back stage. Read or read_comp concatenated with the number of shifts.

keep the right bases from the read, the bases that are the extended part of the match. For that reason a concatenation take place between the output register and the number of shifts and written in the output register as it showed in figure 3.6, in order to know in the host, how many bases to keep on the intermediate contig's concatenation stage. The output register is an 128bit register whose the 120 MSBs is the read (for a 60 bases read) and the 8 LSBs (Last Significant Bits) is unexploited and we can store a number between 0 and 255 (maximum number representing in 8 bits). We store in this 8 bits the (absolute) return value (the corresponding number of shifts) of the extender and all this 128bits output sent to the host via HBM[2] if it is a left extension or via HBM[3] if it is a right extension. The selection done from the control and by the DEMUX where the select_2 signal is "01" for left extension or "10" for right extension and decided from the control. All the starters that did not extended by any read written as it is in the output stream of starters in HBM[4] bank of data by choosing the '10' select_1 of the MUX (to pass the starter into output register) and the '11' select_2 of the DEMUX by the control to send it to the wright HBM bank.

Then we must to replace the current starter with the most left of right edge of the current's round intermediate contig in order to continue the matching process. If a left extend done, we replace the left starter with the read (or read_comp) that done the extension and in right extend we replace the right starter with this read respectively. The new starter now is the read that done the extend of each side and checked for matching with the new reads for further extensions. After this read done the extension we can ensure that this read is the new most left or right part of intermediate contig and by keeping this read as the new starter, we continue the matching process normally.

Every read that extends the starter and every starter that extended by reads, after written in the outputs HBM banks, are not written in reads_output bank (HBM[1]) since they do not need further processing. At the end of the matching process of the starter has been implemented a double buffering technique where the reads_output (HBM[1]) bank of data become the input dataset for the next round and the reads that will be not extend the next

round's starter, will be written in the HBM[0] bank. In this way if in an iteration the HBM[0] is the input dataset's reads and the HBM[1] is the bank that we store the reads that not extend the starter, in the next iteration the HBM[1] will be the input dataset read's and the HBM[0] will be the bank to store the reads that not extend the starter. Only the reads that does not extends the starters written in the output HBM and in this way we implement a dynamic reduce in dataset entries and as a result in the number of iterations. This double buffering process done after the write back stage where the reads_input and the reads_output pointers exchanged between the HBM[0] and HBM[1] banks of data.

3.3 Final design implementation

The previous overall design, which is the initial implementation that we created, presupposes datasets with 60 bases read length, as the conversion of the reads in binary form generating 120 bits values and with the maximum input/output interface implementation of 128 bits width values on the board, the design cannot transfer bigger length's reads. Searching the NCBI database and general datasets that we found, the most used read length is 70 bases, so we must design a different implementation in order to work with these datasets. The differences of the implementation are not so many as is showed in figure 3.7 and presented below in details.

3.3.1 Host reconfiguration

First of all the in/out interface of the board can transfer or receive values of 128 bits, 256 bits and 512 bits max width. Assuming a read length of 70 bases, in binary form we have 140 bits to transfer so we must to configure the in/out interface of the board to 256 bits width. In software part of the design had to be done some changes in storing the input dataset in DDR for the first stage and in generating and writing the output contigs in output file in the final stage. In the first stage we had to store the input reads in 256 bits (ap_uint<256>) entries in RAM, in order to agree with the board's interface. So for 70 bases reads we commit the 140 most left bits of a 256 bits value and storing the binary representation of the read, according to the conversion formula we have presented above. The overall space allocation is bigger than the first 128 bits implementation but we can work with datasets with bigger read length.

In the final stage of generating the intermediate contigs from the processing stage outputs, we had to done the changes in order to decode 256 bit values which has retrieved from the processing stage. As we mention above, as output of the processing stage we encode the read that took part in the extension concatenated in the 8 LSB with the number of bits that matched with the starter in order to keep the appropriate bits of the read in the output contig. So we decode the output 256 bits values with the 140 MSB as the read and the 8 LSB as the number of bases and concatenate the reads to construct the

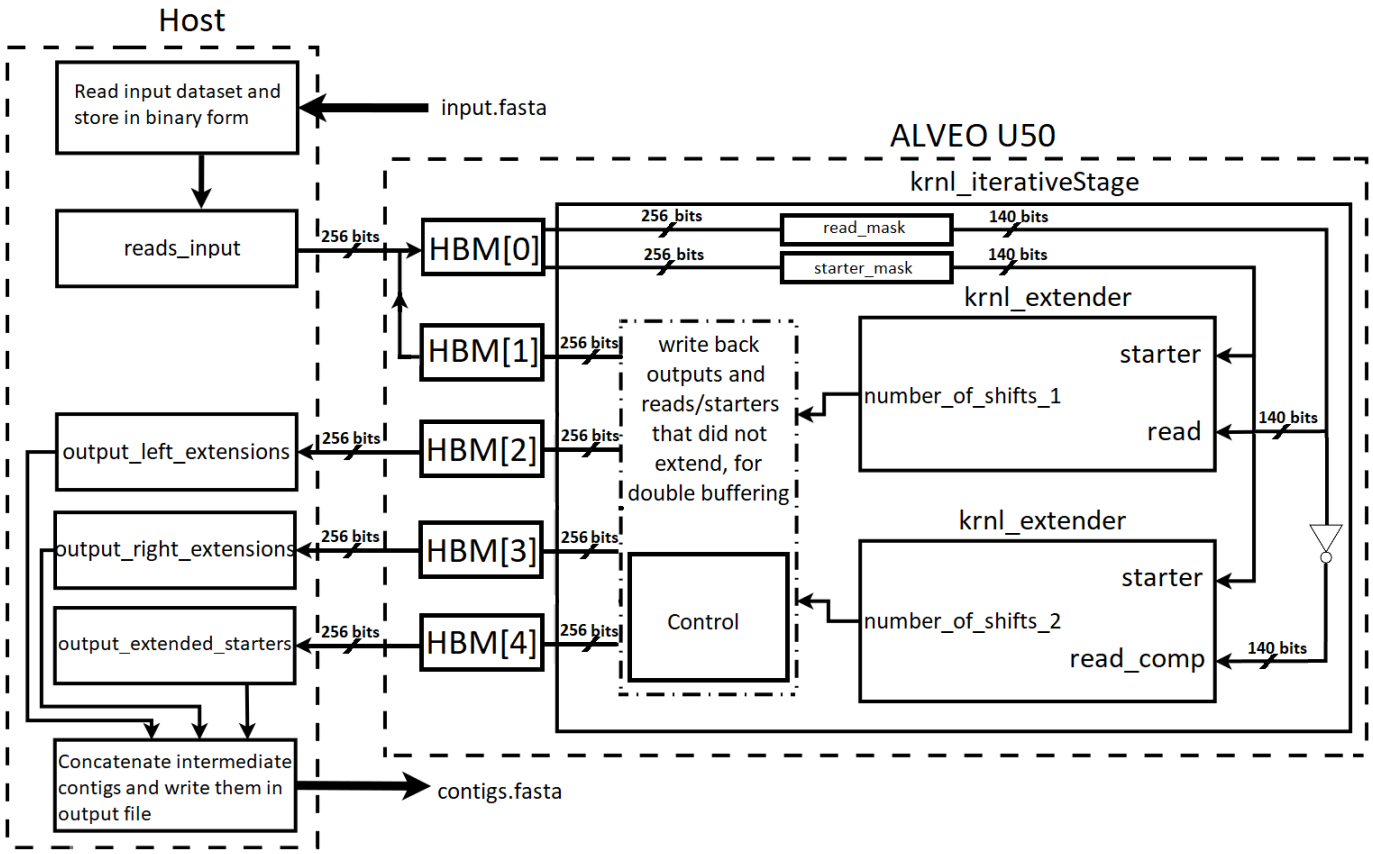

FIGURE 3.7: Our final design in general (host and kernel). We have the 256-bits width board interface and the removal of the prefilter stage of read vector's construction.

intermediate contigs. These intermediate contigs written in the output file of our design.

3.3.2 Kernel reconfiguration

In the second stage of our design, the processing stage of matching the reads between them in order to find redundancy, we had to do some changes to work for 70 bases reads. As we mentioned above the interface of the board now transfer 256 bit values and for that reason we must to change our kernel top level's arguments. We used 256 bits (ap_uint<256>) for each of the 5 HBM and assign each in different bank to store the data as we did in the previous implementation.

In the top level function (krnl_iterativeStage) we start by taking each read from HBM 1 from the DDR. Each read is a 256 bits variable where the read is the 140 MSB of the variable, so we initialize a 256 bits mask with '1' in 140 MSB of it and by taking a bitwise AND between the input variable and the mask, we taking the read. After testing , to make a better spatial allocation in the FPGA, we decide to store the reads in 140 bits registers instead of 256 bits that we take from DDR so the result of the above AND operation shifting right by $256 - 140 = 116$ bits in order to stored in a 140 bits register. In this

way we keep only the beneficial information of the read and occupy exactly the needed space in FPGA.

The next operations of generating the read complementary strand remain as it is with the difference of storing the data in 140 bits registers instead of 120 bits of the previous first implementation. After that in the extender stage, we change the arguments of the extender kernel function, using 140 bits and the overall design of this module remained the same. In the final step of the kernel design, the write back process of generating the output of the kernel, we mentioned that in case of extension, the output is a vector where we store the read in the MSBs and the number of shifts this read done until the extension pass stored in the 8 LSBs of this vector. In this implementation the output must be a 256 bits value, so we store the read in the output vector, shifting it left by 116 bits to saved in the 140 MSBs of the vector and with a bitwise OR we store the number of shifts that extender returns, in the 8 LSBs of the output. This output returned from the kernel to host by HBM[2] bank if we have a left extension or by HBM[3] bank if we have a right extension. The starter that took part in the last matching process stored in the output vector, shifting it left by 116 bits to saved in the 140 MSBs of the vector and returned from the kernel to host by the HBM[4] bank.

3.3.3 The reconfiguration in the software only implementation

This implementation works for 70 bases reads length and we do some configurations to make it an overall design to work for any kind of read length. As we read the input dataset in the host from the input file, we keep the information of the line length and passing it to the kernel to give a variable dimension in the kernel operations in order to work for numerous read lengths. The masks, the shifts and the loop bounds of all the operations of the kernel became a function of line length to work for every dataset. The design became an overall design for every read length's dataset and we keep this in final measures that we will see in chapter 4.

In order to check the efficiency of the design's results and to measure the execution time of the whole process both in software only and software-hardware implementation, we make some changes in the software only implementation in order to work for numerous read's length dataset. The main problem is that for a software only implementation we have a data type's limitation that we cannot work with bigger than 128 bits variables. In order to store in DDR the reads, if the dataset contain bigger than 60 bases per read, we want double entries of 128 bits each to store the read in RAM. So in the first stage of reading and storing the reads in the DDR, for bigger than 60 bases reads, we store the first 64 bases of the read, which is 128 bits in the first entry and the remaining 6 bases in the second entry. Note that by that implementation we can work with bigger than 70 bases reads.

All the next operations of matching the reads between them, generating and writing the final intermediate contigs in the output file, works in terms of double entry of each read in DDR and follow the same configurations. This software only implementation generated the same output intermediate contigs with the software-hardware implementation one, but because the mentioned above limitation was more slow in execution time and this difference of the variable's width can be considered as a hardware implementation's advantage. The time measures and the speedup between these two executions presented in chapter 4.

3.4 Time and space reports for both of the implementations

Afterwards we want to reveal the libraries, functions and programs we use in order to measure function's execution time, space complexity and time reports in hardware. As for the execution time measures, we used many libraries and functions in order to calculate the right measures, but we concluded to the below one. As for the frequency that our design ran and the spatial complexity of our design we used the measures of vivado log after the synthesize and place and route jobs done.

In order to calculate our speedups we must to calculate function's execution time and for that task we use the library of c++, <chrono>, which is a flexible collection of types that track time with varying degrees of precision. Using the function "high_resolution_time" one before call function and one after it returns, we calculate the execution time. This function was the best of what we try to measure time. The time measures with all the results presented below in chapter 4.

As for the hardware part of our design we run the top level function "krnl_iterativeStage" in Vitis HLS [35] to synthesize our kernel and generate reports about clock cycles each data want to be created, latency and intervals and other information about our kernel functions such as pipeline status for each routine or loop and the overall space complexity of our design in this specific ALVEO U50 acceleration card. The kernel took all of the available optimizations, both in time and space, such us pipelining methods which reduces the initiation interval for a function or loop by allowing the concurrent execution of operations and unrolling loops that create multiple independent operations rather than a single collection of operations. In general terms Vitis HLS report calculates the preliminary results of the design based on RTL description of the kernel. The design in phase of synthesize and routing by Vivado can take multiple optimizations both is space and time complexity, such as mapped LUTs logic in Block RAMs for better space occupation or merge logic modules for time speedup, so the Vivado log includes the final report values of space and time complexity. We informed by the Vivado log that contains the real values after synthesize.

Because of the variable upper bound of loops we limit the upper bound with the directive #pragma TRIP COUNT, in order the HLS report give us measures. The program want to know the maximum number of iterations each loop will perform to give time and space measures. Loops and routines is pipelined by using the system #pragma HLS PIPELINE directive. In addition, in order to make the in-kernel functions to work pipelined, we had to store their arguments in different registers for each function call. For example in case of extender (which detailed described in chapter 3), if we want to run the extender with the read and the read complementary strand pipelined, we must store the arguments of each function in different registers in FPGA in order the two function calls retrieve each data from different registers.

The extender design which described above included the modified counter that it contains, pipelined with a pipeline level II=1 which means that extender takes new data each clock cycle and with an interval of 1 cycle, it generate output the next cycle. The latency of the extender found to be 121 clock cycles. The outer loop that takes each read and checking it with the starter, did not manage to take any pipeline level because of the variables dependencies. It has an iteration latency of 272 clock cycles which means that we take every read from the input dataset each 272 cycles, number of loop execution's time in clock cycles. From place and route tools, the maximum clock frequency for the accelerator was found to be 300,3 MHz for our initial implementation and 298,8 MHz for our final implementation. These measures appeared in below table:

Kernel top function	krnl_iterativeStage
Scaled Frequency (MHz)	300,3 MHz
BRAMs (%)	14,29%
DSPs (%)	0,12%
Registers (%)	14,26%
LUTs as Logic (%)	18,1%
LUTs as Distributed RAM (%)	1,72%
LUTs as Shift Registers (%)	1,14%

TABLE 3.3: The Vivado log measures for the first implementation pf the RMF (128 bits data transferring).

Kernel top function	krnl_iterativeStage
Scaled Frequency (MHz)	298,8 MHz
BRAMs (%)	15,18%
DSPs (%)	0,12%
Registers (%)	16,24%
LUTs as Logic (%)	19,87%
LUTs as Distributed RAM (%)	1,7%
LUTs as Shift Registers (%)	1,22%

TABLE 3.4: The Vivado log measures for the final implementation of RMF (256 bits data transferring).

From the measures of the table 3.3, the overall space occupation of our initial implementation's kernel is sufficiently small of the order of 14,29% LUTs as Logic and as presented in the table 3.4, for the final implementation of 256 bits interface, we have similar values of the occupation of the design in the FPGA (LUTs as logic = 15,18%). The resources utilization reported above indicates that multiple accelerators can be placed inside the specific FPGA and so we can try to run our kernel with two or three instances in the same FPGA fabric, in parallel. In that way, theoretically speaking, the input dataset can divided in two or three parts and the processing can be done faster. In our final implementation as we observe in the tables 3.3 and 3.4, the measures of the registers, shift registers, LUTs, BRAMs raised a bit as we have to work with bigger read length.

3.5 Extended implementation

As we saw above, the kernel occupies in FPGA about $\approx$ 20% maximum of the overall space. That theoretically means that we can implement a bigger parallel design in order to improve the efficiency and the speed of the algorithm. Theoretically speaking we can combine 2 or 3 extenders in the kernel (in the same FPGA) and try for matching between starter and multiple reads at the same time. An implementation that utilizes two such RMF accelerators is already implemented but the bitstream creation was not yet possible (at the time that this diploma thesis is written) due to routing congestion, as reported by Vitis.

The ALVEO U50 FPGA that we used has 512-bits wide memory-mapped AXI4 interface and that means that for each iteration we can retrieve a maximum of 512-bits per port. In this way we can take more that one read from the reads input HBM, in the same time and make a parallel matching process. If we assume a read length of 70 bases, we have 140-bits read's representation and in 512 bits we can concatenate 3 reads per iteration to retrieve. In this way we can implement a kernel that checks for matching the starter with these 3 reads in parallel. We implemented this kernel, in terms of 70 bases read length and with some of reconfigurations the design implemented with 60 bases read length as well. The overall design follows the same HBM banks connections with the first basic design and we have HBM[0] and HBM[1] for the dataset's double buffering, the HBM[2] for the left extensions output, the HBM[3] for the right extensions output and the HBM[4] for the starters output of the kernel. The main differences are in the bits width in the board's interface and we have 512-bits for double buffering HBMs and 256-bits each of other 3 output HBMs.

3.5.1 Host reconfigurations

We start from the host and the first stage of reading the input dataset and storing it in RAM. The basic design that implemented first, used 128-bits values and works fine for 60-bases reads. The storing of the reads in RAM

done in 128-bits (ap_uint<128>) values and we have one index-address for each read. In this extended implementation, we have to store 3 reads in the same index-address, using 512-bits length value (ap_uint<512>). In this way, we make a realloc of 512-bits for each triad of reads that we read from input file and storing them concatenated in the same memory address. As we mention above in this implementation, we want to works for 70-bases reads as well with the 60-bases reads. Concatenation and storing of each triad of reads done sequentially starting from index 511 (MSB of ap_uint<512> value - most left bit), until index $511 - 3 \times 140 = 91$ for 70-bases reads and until $511 - 3 \times 120 = 151$ for 60-bases reads. By that way each triad of reads saved in the same memory address and HBM[0] and HBM[1] of double buffering of the dataset now receive and transfer 512-bits values as it seems in figure 3.8.

In the kernel of the design we needed to do multiple changes and different reconfigurations in order to work for trinities of reads. First of all we reconfigured the input/outputs of the kernel as we mention above. The HBM[0] and HBM[1] of the double buffering of the dataset became 512-bits width from 128-bits width in order to transfer the triads of the reads. The HBM[2]-4 that they are for the output data, now transfer 256-bits of data from 128-bits that they transferred before in order to match with the 70 bases read length. Further detailed reconfigurations about the kernel presented below.

In the third stage of our design, the concatenation and writing of the intermediate contigs in the output file, the differences from the previous first design were fewer. The output HBM[2]-4 (as we see in figure 3.8) of the kernel of left and right extensions and for starters received, now contain 256-bits values of data, as we will see in the next kernel configurations section and the main reconfiguration of this third stage of concatenation is to works with 256-bits values in contrast of the previous design of 128-bits returned values. So we change the ap_uint<128> to ap_uint<256> values and my shifting methods and masking in order to retrieve the 8 LSBs of the returned values of the kernel, which is the number of matched bases, we concatenated and wrote the intermediate contigs in the output file.

3.5.2 Kernel reconfigurations

As we retrieve 3 reads from the input HBM in each clock cycle, we must implement a trinity of extender processing modules with different arguments values for each one, in order to implement a pipelined execution. Each one of the three reads checked for extension with the starter in the same time with the other reads. As we mention above, alongside with the read we check for extension the complementary strand of the read with the starter, pipelined. So if we assume three reads, by taking the three complementary strands of them, we must implement six extender executions pipelined, as it seems in the figure 3.8. In order to implement this design we must define all the extender's arguments separated to have individual access of each argument per extender, essential feature for pipelining. From HBM[0] we

take the first 512 bits from reads_input and with mask_input we take the 140 MSBs (511 to 371) as the starter and assign this value to six different 140 bits registers starter_x_y where x represent the comparison per read (1 to 3) and y represent either read or read complementary comparison (1 to 2). Afterward from HBM[0] we take the next entry of the reads_input and by the mask_reads we split this 512 bits to three 140 bits registers that represent the three reads that will take part in the comparisons. From the 512 bits, the 140 MSBs (511 to 371) assigned to read_1 register, the next 140 bits (370 to 231) assigned to read_2 register and the next 140 bits (230 to 91) assigned to read_3 register. From these three registers by a bitwise NOT for each, we take the three complementary strands in registers read_comp_1, read_comp_2 and read_comp_3. These registers passed as arguments in the extenders and the six comparisons done pipelined so in the next cycle we take the six outputs of number_of_shifts where in the write back stage the control decide which comparison of six gave the best match between the starter and the reads and the biggest matching read (or read_comp) returned to host via HBM[2] if we had a left extension or HBM[3] if we had a right extension and via HBM[4] the starter that extended. If no one of the comparisons give a result, the starter did not extend from any read so it written in reads_output stream via HBM[1] for double buffering.

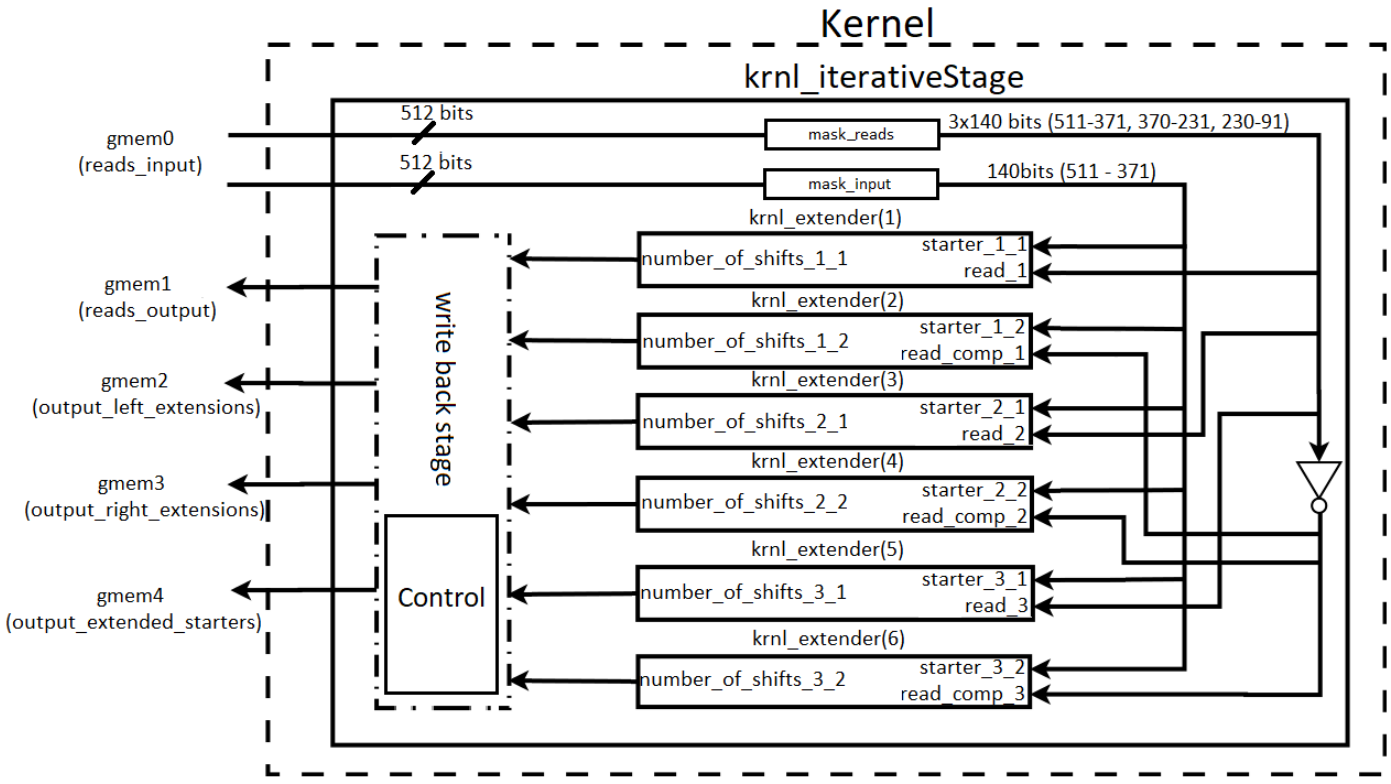

FIGURE 3.8: The extended Kernel with the triad of reads.

Each iteration we retrieve three reads from DDR. That means that for the first iteration, the first read of the triad is going to be the starter and the other two reads will be the reads to checked for extensions with the starter. Another specific situation is when the overall number of the dataset's reads is not a

multiple of 3 so the last index has less than three reads. In this particular cases, we have less than three reads and complementary strands of them to checked with the starter so we can limit the extenders runs less than six times that we mention above. To satisfy this functionality we assign to each extender a enable signal. If we have three different reads, we have all the extender's enable to '1' else if we have two different reads we have the four of the six enabled to '1' and if we have one read only we have the two of the six enabled to '1'. In that way we save unused runtime. Apart from this we manage to save execution time by decide if we want to calculate or not the read complementary. Obviously for read that has zero value (that means that we do not have a read), we did not calculate the read complementary strand and the particular extender is disabled. After that stage we continue to the extender executions (pipelined) as they mentioned in the corresponding subsection of extender stage above. The returned values of the extenders now are six different values (return value of extender that did not ran is 0 -enable='0'-) and we moved on the write back stage.

In the write back stage, we have 3 HBM[2]-4 of 256-bits width in order to write the outputs in DDR for further processing by the host (concatenate and write the intermediate contigs in output file). The first implementation that described above, used 128-bits values to return the outputs to host. In 120 from 128 bits, stored the read and in the most right 8 bits stored the number of matched bases by the extender, in order to know in the host the number of bases from the read to keep. Now we return 256 bits values because this design works for 70 bases as well and we want at least 140 bits to store the read. The LSB 8 bits of the 256, used in the same way with the previous implementation, by storing the number of the bases that matched in the extender's matching process. This return type used to return the left and write extended reads via HBM[2] and HBM[3] respectively. The starters that took part in the matching process each round, returned as it is in 140 most left bits of the 256 bits variable via HBM[4] port. As a final step here we implement the double buffering from HBM[1] to HBM[0] to ready the input dataset stream for the next iteration. All the triads of reads that remained unused by the extender process, were written in output stream via HBM[1] and now they will be the next iteration's new triads of reads.

3.5.3 Extended design exceeded the limit of FPGA resources

After all the previous extended implementation, we start with the synthesize phase of our design in order to make a assessment of the kernel space occupation. The synthesize in the Vitis HLS report, give us very large measures concerning the number of LUTs that the design will occupy in the board. This is expected as a point because as we take three different reads in each iteration, we must define triples of registers to store each read, triples of registers to store the read complementary strands, six instances of the extender stage in order to execute each read or read complementary matching with the starter and a write back stage that execute three times more operations and comparisons in order to define the correct returned value. Except of these, all

the variables and the arguments of the in-kernel functions, declared six different times in order to make the six extenders that reported above to work pipelined.

As a result of the above, the Vitis HLS report calculated a maximum LUTs usage over the 100% of the available resources of the ALVEO U50 card. As a following step we try to build the hardware on Vitis to rule out any possibility of successful design build, due to optimizations or design merges from the Vivado. The build of the hardware, first passed the block synthesis stage, executed a first logic optimization but in phase of logic placement of the LUTs in the FPGA space the build failed inform us that LUT as Logic over-utilized in Top Level Design and as a result this design requires more LUT as Logic cells than are available in the target device ALVEO U50. After that, we tried to merge some of the logic, processes or routines in the code but we did not manage to reduce the LUTs usage of the design as much as to fit the target device's available resources.

3.5.4 A different extended implementation design

The above implementation did not fit the FPGA resources and failed in placing the logic phase of the build. The following step that which failed again for the same reason, is to implement a two reads matching process per iteration. The previous design take three reads and with the three complementary strands we wanted six different extenders in order to execute the pipelined matching process. Now the idea is to take two reads, find the two complementary strands and design a four of extenders in the same way with the previous extended implementation. The declaration of the variables and the arguments of the in-kernel functions and the operations and comparisons of the write back stage now made for two reads.

Despite the reduction of the LUTs usage of this implementation, the design neither now fits in the target device. The HLS report calculated above 100% LUTs usage and the build of the hardware failed in the same phase with the above design. The placing of the logic in LUTs failed to implemented and the merging of the logic in the code did not help the outcome.

3.6 The software implementation with score table

Another one implementation that has theoretically fine results is the score table implementation. The main idea is to generate a score table that keeps the matching score for each read from the comparison with the starter and start the extender after find the read with the best matching score with starter. Starting by taking the first read as starter and after that, passing all the dataset read by read, we make the comparison between this starter and all the other reads as we described above using the extender design. If a successful extend done, we do not return the results immediately but we store this matching score in the corresponding index of each read in the score table. After passing

all the dataset, we have the information on how many bases has in common the starter with all the other reads. By searching this score table we find the biggest score, which is the read with the best matching rate with the starter. So we keep this read as the best read and continue the basic algorithm that we described in our main design implementation to find all the the remaining extensions to generate the intermediate contig.

In this way we starting the matching process with guaranteed the read that has the best matching rate with the starter and by this way we can improve the quality of the results. We implemented this design in software only and we realize that it could have very good output results but it was very slow process in execution time. This makes sense if we observe that before starting the matching process between the starters and the reads by passing all the dataset multiple times, the algorithm passing all the dataset in order to calculate all the score rates. The speedup that we took from our final design above was the order of 2x-4x faster so we decide not to implement the hardware of this design because the spatial and time complexity is already big and the benefits of the design of better output results are not bigger than the implementation procedure.

Chapter 4

Results and Discussion

In this chapter we will present the results both in software and hardware and we will submit the partial and the overall speedups that we measured both in subsystems and in the whole design (included Velvet processing). We used a total of 5 dataset in order to run the design and the Velvet software. All the datasets are downloaded from NCBI database [32] and they are whole genome shotgun sequences. This method involves breaking the genome into a collection of small DNA fragments that are sequenced individually. Each dataset has a coverage factor that refers to how many times we have the genome of the organism in the sequence. The datasets referring to a Pyruvatibacter mobilis bacterium (3291 Kb) with a 100x coverage, a Pseudomonas bacterium (6689 Kb) with 114x coverage factor, an Aythya fuligula-tufted duck (11749 Kb) with a 64x coverage factor, a Melopsittacus undulatus-parrot bird (23291 Kb) with 61x coverage factor and a Photinus Piralis-common eastern firefly (70234 Kb) with 40x coverage factor. We choose different type of datasets and numerous size in order to quantify our results in depth.

These datasets consist of 70 bases length per read and used for the final experiments while we ran and checked our final implementation of 256-bits width interface on the board. In the initial implementation of 128-bits width interface, as we mention above, we can transfer 120-bits reads maximum which means a maximum of 60-bases reads. The most usual read-length of genome datasets is the 70 bases per read and for that reason we used the original datasets in the final implementation and in the initial implementation we used synthesized datasets based on the original one in order to have 60 bases length reads.

The FPGA we used to run our kernel is an ALVEO U50 Data Center Accelerator Card [30] which provide optimized acceleration for workloads in financial computing, machine learning, computational storage, data search and analytics, which is presented in figure 4.1. It is a single slot, built on Xilinx UltraScale+ architecture, low profile form factor passively-cooled card. It supports PCI Express (PCIe) Gen3 x16 or dual Gen4 x8, is equipped with 8 GB of high-bandwidth memory (HBM2-316 GB/s) and Ethernet networking capability. The overall board specifications showed in table 4.1. Our results are compared with an optimized single threaded software, executed on

FIGURE 4.1: The ALVEO U50 Acceleration Card
www.xilinx.com/content/dam/xilinx/imgs/kits/U50_Hero_1_Bracket.png

a workstation which contains an Intel® Xeon® Processor E5-2630 v4 (25M Cache, 2.20 GHz) with 256 GB of RAM.

Board Specifications	**ALVEO U50 Accelerator Card**
Look-up Tables (LUTs)	872K
Registers	1,743K
DSP Slices	5,952
HBM Memory Capacity	8GB
HBM Total Bandwidth	316 GB/s
Internal SRAM Capacity	28 MB
Internal SRAM Total Bandwidth	24 TB/s
Clock Precision	IEEE 1588
Vitis Platform	Gen3x16 XDMA, Gen3x4 XDMA3
Maximum Total Power	75W

TABLE 4.1: The ALVEO U50 acceleration card specifications.
www.xilinx.com/products/boards-and-kits/alveo/u50.html#specifications

4.1 Quality results

The main job of our design is to reduce the dataset's size without losing any valuable information. In order to have a similarity measurement to ensure that the filtering process does not corrupt the output data, we used a sequence alignment tool to check the similarity of the two outputs, the output of the velvet run with the original dataset and the output of the velvet run

with our Hardware's intermediate contigs. Manually comparing the outputs was not feasible. BLAST is a Basic Local Alignment Search Tool [3] which it has two main features. First you can attach a dataset and the tool can search the biological databases to find the biggest match and inform you about the genome. The second feature is the sequence aligner. We can attach two or more files with genomes and BLAST can check them, by shifting them in specific position, to calculate a best matching rate between them by giving the comparison a reward/penalty ratio. It calculate a similarity rate which can inform us about the quality of output of our design. As we mention in chapter 2, a similarity bigger than 95% between the two outputs is a very good result. We downloaded the NCBI BLAST software [12] and we compared the 2 outputs of the Velvet assemblies (Velvet only/Velvet+Hardware) from each dataset to check for similarity.

4.1.1 The N50 values of the contigs

Generally speaking there are many factors that affect the quality of assembly. It depending to the assembler, the parameters that it takes to execute the assembly and other factors that can differentiate the outputs. In this particular situation for the velvet genome assembler, the input parameters like k-mer length, which is the length of fragments of the reads in order to build the de Bruijn graph and number of mismatches allowed can significantly affect the quality of the output and after many experiments we set them in auto. Only the k-mer length took numerous values during the runs and we keep the values where the assembly generates the biggest contigs and less in number. The most popular metric to measure quality are the maximum length of the contigs and the "N50". As we mention in Chapter 2, N50 is the minimum length of the contig such that summing up the length of only those contigs whose length is more than N50 cover 50% of the genome. In the below table 4.2 we present the quality of the assembly included these two basic factors in each of 5 datasets of 70 bases per read, which we used them in order to calculate the final implementation's results that will be presented below.

Dataset	N-50 in bases	Max Contig in bases
Pyruvatibacter mobilis	429918	429918
Pseudomonas	897500	897500
Aythya fuligula	5261723	5261723
Melopsittacus undulatus	6794489	6794489
Photinus pyralis	6789577	6789577

TABLE 4.2: The N50 metrics of the assembly from the outputs of the velvet genome assembler.

From multiple experiments and runs of velvet between our intermediate contigs as input and the original dataset as input, the number of N50 and max contigs found to be in similar lengths and the maximum assembled contig's

length found to be the same with the N50 value. As we mention above, the assembly depends on the factors of the velvet make and run stages such as the max k-mer length, the hash length, the coverage cut-off and the expected coverage. We general used the default values of these factors and the outputs appeared to be similar included the BLAST's outputs. In evaluating our results beyond the N50 metric, using BLAST, we have better than 95% similarity with Velvet results without RMF, the minor difference being an occasional base difference (e.g. A or C instead of T).

4.2 Initial implementation results

As we mention above, we used synthesized datasets based on the original one in order to have 60 bases length reads in datasets. This synthesize make the dataset size bigger than the original one because we want to keep the DNA information in smaller reads length, so we have more reads in quantity. In this point of view the datasets size almost doubled as we present in the following results.

4.2.1 Speedup without I/O operations

The time measures and the speedup of the kernel can be quantified both included I/O operations of read the input dataset and write the final contigs in hard drive and without these operations. In table 4.3 we will present the time measures of running the processing stage both in software function and hardware kernel, without these I/O operations, to quantify first the processing stage of the RMF only.

Dataset	Size in Kb	Execution time in sec		Speedup
		Software	Hardware	
Pyruvatibacter	5655	86,71	1,93	44,92x
Pseudomonas	11494	174,8	3,9	44,82x
Aythya	20188	312,72	6,96	44,93x
Melopsittacus	39326	625,41	13,51	46,29x
Photinus Piralis	120683	1985,71	44,29	44,83x

TABLE 4.3: Processing module's measured execution times both in software and hardware and the speedups (without I/O operations from host/software).

As we can see, the optimizations of the hardware speeds up the processing time up to 44x-46x faster. The pipelining, unrolling and other hardware optimization options, help the most expensive in terms of calculation process of our design. Below we will present the general speedups for our design in total, included files I/O operations which implemented in software and about the overall speedup included the velvet genome assembly processing.

4.2.2 Speedup including I/O operations

Generally, in the host, we have two main tasks. In the first stage we have the reading of the input dataset file and the storing of the reads in RAM in binary form and in the third stage a second task which is the concatenation of the intermediate contigs and the writing of them into the output file 'contigs.fasta'. We implement a basic time calculation of this tasks to measure the efficiency of our software part. Obviously the bigger the input dataset file, the most time it takes to read it, store it in DDR and at the end to write in output file the intermediate contigs, because there are more.

We measured the overall time execution of our both implementations (Software only and Software+Hardware) included the I/O operations of the first and the third stage. The software compiled with -O3 optimizer and has exactly the same calculate operations with the hardware. We ran all of the 5 datasets and we take 100% match between the outputs of the two methods, as expected, but we take numerous speedups as we present in the table 4.4.

Dataset	Size in Kb	Execution time in sec		Speedup
		Software	Hardware	
Pyruvatibacter	5655	87,27	3,28	26,6x
Pseudomonas	11494	175,96	6,67	26,38x
Aythya	20188	314,78	11,6	27,13x
Melopsittacus	39326	629,65	21,77	28,92x
Photinus Piralis	120683	1997,87	71,47	27,95x

TABLE 4.4: My design runs both in software and hardware and the speedups (included I/O operations).

As we can see the overall speedup fails a little from above only processing stage speedup and this is explained by the I/O file operations. We have an average 26x-28x speedup in hardware for our filter. As we see, a general conclusion, observing the dataset's speedup measures is that the speedup first rises as the input dataset's size rises, reach a peak point and after that start to fails. This downhill course is not so important as the speedup fails a little as we will see in the following results. Next step is to measure the overall execution times included Velvet operations and examine the overall speedup that the genome assembler can take by included our design as an preprocessing add-on.

4.2.3 Overall speedups included Velvet

As we mention in Chapter 2, we generate the output contigs via two main data paths. One path is where the input dataset pass through the Velvet genome assembler and the second one is where the input dataset pass through our hardware design and the intermediate contigs that our design generates, pass into Velvet genome assembler. In order to measure possible speedups between the two ways, we measure Hardware and Velvet execution times each and compare them.

The measured values of the execution time with our hardware kernel involved seems in the below table 4.5 and the measure times of the Velvet run without our filtering processes seems in table 4.6.

Dataset	Execution time of hardware in sec	Execution time of the assembly in sec	Total execution time in sec
Pyruvatibacter	3,75	38,67	42,42
Pseudomonas	7,57	56,85	64,42
Aythya	13,39	202,18	215,57
Melopsittacus	26,1	435,48	461,58
Photinus Piralis	82,98	4356,16	4439,14

TABLE 4.5: The measured execution times of the assembly with our preprocessing stage involved.

Dataset	Execution time of the assembly in sec
Pyruvatibacter	90,53
Pseudomonas	284,19
Aythya	862,99
Melopsittacus	3337,83
Photinus Piralis	32580,5

TABLE 4.6: The measured execution times of data path without our hardware design involved (only Velvet run).

We can notice that proportional the input dataset size, we have different execution times. As the dataset's size is growing, the time measures is growing, reach a peak point and then it shrinks. After that measures we can calculate the overall speedup of our design. In general we noticed a numerous speedup between the two ways and the Velvet genome assembler runs faster with our intermediate contigs as input. The overall speedups of the task is presented in the following table 4.7.

Dataset	Overall assembly time without our RMF in sec	Overall assembly time with our RMF in sec	Overall speedup
Pyruvatibacter	90,53	42,42	2,13x
Pseudomonas	284,19	64,42	4,41x
Aythya	862,99	215,57	4,01x
Melopsittacus	3337,83	461,58	7,23x
Photinus Piralis	32580,5	4439,14	7,33x

TABLE 4.7: Overall speedup between the two methods.

The overall speedup has numerous values where the most significant is for bigger datasets such as Melopsittacus and Photinus Piralis. The biggest the dataset's size, the higher the overall speedup of the process including filtering from RMF and assembly from Velvet. In that point of view the Velvet took a significant overall speedup with our filter's outputs as input.

4.3 Final design results

As we mention above the previous initial design results was in testing implementation of the 128-bits input/output interface of the board and referred to 60 bases reads datasets. The final design implemented a 256-bits width interface in order to work with 70 bases (140 bits) reads length. The optimizations of the kernel was the same as the initial's implementation, included pipelining, loop unrolling, data flow optimization and multiple register's stores for the function's arguments.

In this implementation, as the initial one, the interval of the extender held in factor II=1 which means that in each clock cycle, the extender take a new read and a starter and in the next cycle, produce the results of the matching process. The latency of the extender found to be 141 clock cycles. An interval II=1 calculated from the write back stage and the double buffering process which is the best pipeline level. The iteration latency of the loop which take every read from the dataset in order to matching it with the starter found to be 292 clock cycles and that means that every read from the dataset retrieved every 292 clock cycles (from 272 that was in the initial implementation).

We did the same experiments and we took the same time measures as the initial design. The datasets we used are the original ones, as we downloaded them from NCBI database.

4.3.1 Speedup without I/O operations

As a first measurement, we calculated the execution times both for the software and the hardware part of the main processing stage of the filter. In these values we did not take the I/O operations of reading and writing in files. The software built by using the -O3 optimizer of gcc compiler. The below table 4.8 present these measures.

Dataset	Size in Kb	Execution time in sec		Speedup
		Software	Hardware	
Pyruvatibacter	3291	365,37	3,28	111,39x
Pseudomonas	6689	720,66	6,48	111,21x
Aythya	11749	1263,66	11,37	111,13x
Melopsittacus	22887	2437,59	21,92	111,2x
Photinus Piralis	70234	7595,37	68,39	111,05x

TABLE 4.8: The measured execution times for the processing module of the RMF in final implementation.

The speedups between the two options are enormous and reach a 111x faster in hardware implementation. This can be explained by the method that the dataset's reads has been saved in the memory in the software implementation. As we mention in Chapter 3, in order to store a read of 140 bits (70 bases) in software we used two entries of 128 bits integers, because the limitation of the data types in software. That means that for each read we occupy

double memory to save it and in order to work the processing stage, for each read we retrieve and save double entries of data from the RAM and the operations done double. As a result the execution time increased drastically, in contrast with the hardware implementation which using arbitrary precision data types and can handle the 140 bits reads better in space allocation.

4.3.2 Speedup including I/O operations

As a second step, we included the I/O operations in our measurements and as expected the results remained fine in execution time between the two methods (Software/Hardware). In this point we mention that the output files with the intermediate contigs which generated by the two methods, had a 100% percent of match. The execution times between the methods showed in the following table 4.9.

Dataset	Size in Kb	Execution time in sec		Speedup
		Software	Hardware	
Pyruvatibacter	3291	365,79	3,96	92,37x
Pseudomonas	6689	721,52	7,82	92,26x
Aythya	11749	1265,18	13,7	92,34x
Melopsittacus	22887	2440,58	26,4	92,44x
Photinus Piralis	70234	7604,38	82	92,73x

TABLE 4.9: The measured execution times for the RMF included the I/O operations in final implementation.

As we can observe, the speedup decreased slightly included the I/O operations but it still has advantageous values. The optimizations and the overall pipeline level of the design in the hardware, speedup the filtering by an overall ratio of 92x faster.

4.3.3 Overall speedups including Velvet

The above speedups referring to the reduction of the execution time from the software to the hardware of our final design. The main idea is to speedup the genome assembler's execution time. We took the same measurements as the initial implementaton's for both assembly data paths, included Velvet (Velvet, RMF+Velvet). In all of the measurements we used the default parameter values for the assembly as we mentioned above.

We present in table 4.10 the execution time of the velvet genome assembly processing by taking as input the original datasets (70 bases read length).

As we can observe the execution time of the assembly increasing depending the file size. The assembler first reads the input dataset and by a hash factor, stores in RAM the k-mers, little fragments of the input reads in order to build a de Bruijn graph, which used in the second part of the process, the assembly stage. The larger the file size, the larger the number of reads and therefore

Dataset	Execution time of the assembly from Velvet in sec
Pyruvatibacter	65,11
Pseudomonas	220,67
Aythya	604,78
Melopsittacus	1932,39
Photinus Piralis	12922,43

TABLE 4.10: The measured execution times of Velvet with the original datasets.

the build time of the graph. By throw away the unused repeats of the reads these times reduced, as it presented in table 4.11.

Dataset	Execution time of RMF in sec	Execution time of Velvet run in sec	Total execution time in sec
Pyruvatibacter	3,96	31,14	35,1
Pseudomonas	7,82	28,57	36,39
Aythya	13,7	162,05	175,75
Melopsittacus	26,4	345,66	372,06
Photinus Piralis	82	3234,37	3316,37

TABLE 4.11: The measured execution times of the assembly with our RMF involved.

As we see the overall execution time of the assembly reduced as the velvet has to deal with smaller datasets. The reduction of the dataset size, presented in table 4.12 .

Dataset	File size in Kb	
	Original	Final
Pyruvatibacter	3291	1454
Pseudomonas	6689	2957
Aythya	11749	5190
Melopsittacus	22887	10095
Photinus Piralis	70234	30937

TABLE 4.12: The reduction of the dataset's size from our RMF.

Observing all of the measurements we conclude in the following table 4.13, where we present the overall speedup of the process included the velvet genome assembler and our final design.

As we can see in the results, the overall speedup of the final design increased as the dataset file's size increased. The speedups in each case first increases, reaches a peak and then tapers off. This is a common conclusion with the work that done in [39], which the approach we followed in order to build our kernel. We have an overall speedup between 2x to 6x faster in the whole

Dataset	Velvet execution time in sec	Velvet+RMF execution time in sec	Overall speedup
Pyruvatibacter	65,11	35,1	1,85x
Pseudomonas	220,67	36,39	6,06x
Aythya	604,78	175,75	3,44x
Melopsittacus	1932,32	372,06	5,18x
Photinus Piralis	12922,43	3316,37	3,89x

TABLE 4.13: The measured execution times and the overall speedup with our final implementation filter involved.

process with the final outputs of the assembly between the initial and the final datasets, having 95% similarity as the BLAST informed us in the quality section above. In [39] they concluded in an overall estimated speedup that reached the 13x ratio, but they estimate it with multiprocessing elements in the FPGA, in the same design, while our implementation consist of a single kernel design.

Chapter 5

Conclusion and Future Work

5.1 Conclusion

According to the results, we managed to speedup the Velvet assembly process by design a pre-assembly filtering implementation. This implementation consist of checking the input dataset for unused redundancy between the reads and by that way we reduce the size of the input file that Velvet use to generate the output assembled sequences. Our design managed to gain significant speedups. The differences between the execution times between the software and the hardware implementation of our both designs (initial and final), found to be very large. We measure speedups up to 27x-28x ratio for our initial design and up to 92x ratio for our final implementation.

As we can see in table 4.13, the overall speedup of the whole process, for the smaller datasets such as Pyruvatibacter remains in $< 2x$ ratio. This means that for small datasets (sizes smaller than 5mb), we concluded, after several experimentation, that the whole process did not take any significant speedup and the execution time of the assembly processes remained the same. We managed to gain an $\approx 2x$ speedup using our filter but it refers in differences of the order of seconds. After that with a bigger size of the input dataset and as this size increased, we have bigger speedups. This speedup ratio increased until it finds a peak value which found to be approximately in 7x speedup for the initial implementation and in 5x-6x speedup for our final implementation.

These final results differ a little in relation to the work of [39]. As we mention above, this can be explained by the fact that they used multiple instances of the same kernel into different Processing Elements (PEs). This design can reduce even more the execution time of the preprocessing stage and consequently the whole assembly's execution time because the dataset's reads divided in different PEs. In our implementation we tried to implement a similar extended implementation (as we described in Chapter 3), but it failed to fit in the FPGA due to the resources limitation.

Our results presented in Chapter 4 and at this point following the visualizing of the results. We executed our implementation many times on Alveo U50 and further visualization with data graphs presented below.

5.1.1 Visualize our results

As we mentioned in Chapter 4 we used 5 synthesized datasets of numerous size in order to fit for our initial implementation's limitations of 60 bases read length and the original datasets for our final implementation that works for bigger read's length (70 bases per read), in order to check the functionality and the efficiency of our designs.

First of all in tables 4.3 and 4.4, we saw the exponential increase of the execution time in terms of dataset's file size in initial implementation and we calculated the speedup that the design conquered between the software only implementation and the hardware one which tabulated in table 4.7. We present these results in below 2 graphs (5.1, 5.2) in order to visualize the exponential increase of time execution compared with the input file's size.

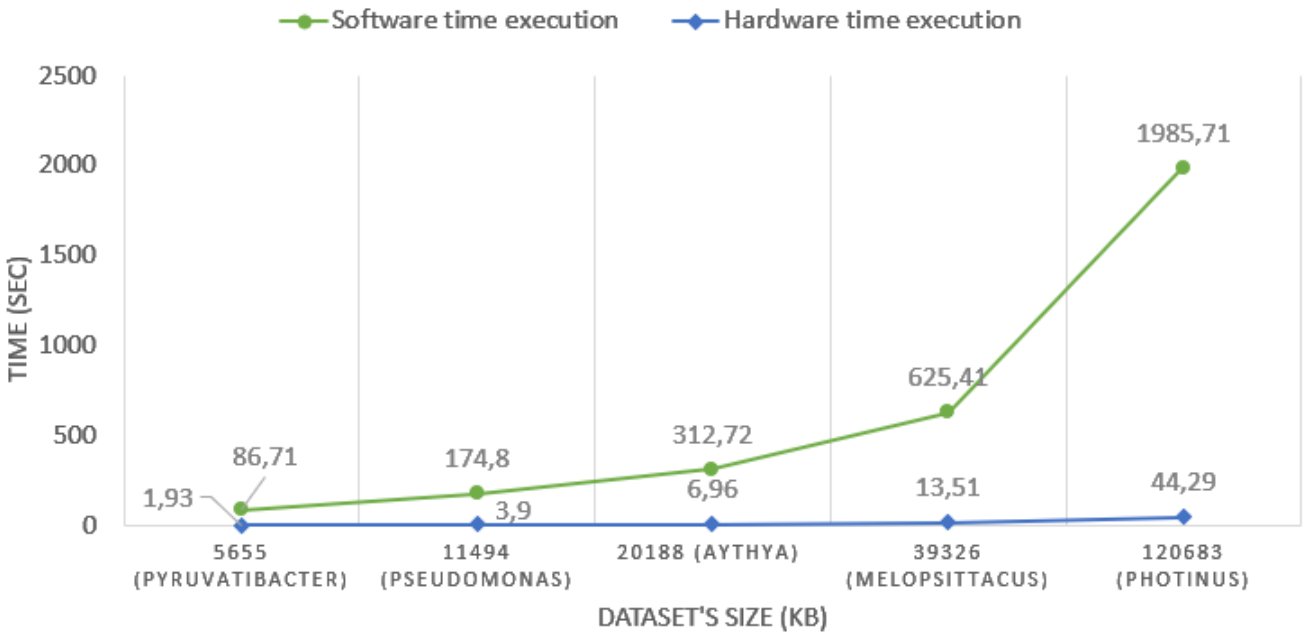

FIGURE 5.1: RMF measured execution times both in software and hardware (without I/O operations from host/software)

As we see in graphs 5.1 and 5.2 the overall execution time increased as the file size increased. The reduction of the execution time in hardware is large for each dataset's size and as the file size increased, this reduction increased. This is because the O(N^2) time complexity of the algorithm that compare each read with all the other reads of the dataset to find the matches. As the dataset's size increased we have more reads to compared between them and the hardware pipelining execution method help the process to run more faster than software.

The corresponding time measurements about the final implementation showed in below graphs 5.3 and 5.4. The first graph 5.3 presents the execution times about our final implementation's processing module, without I/O operations and the second graph 5.4 presents the final measures about the whole final implementation's execution. We can observe similar results both in initial

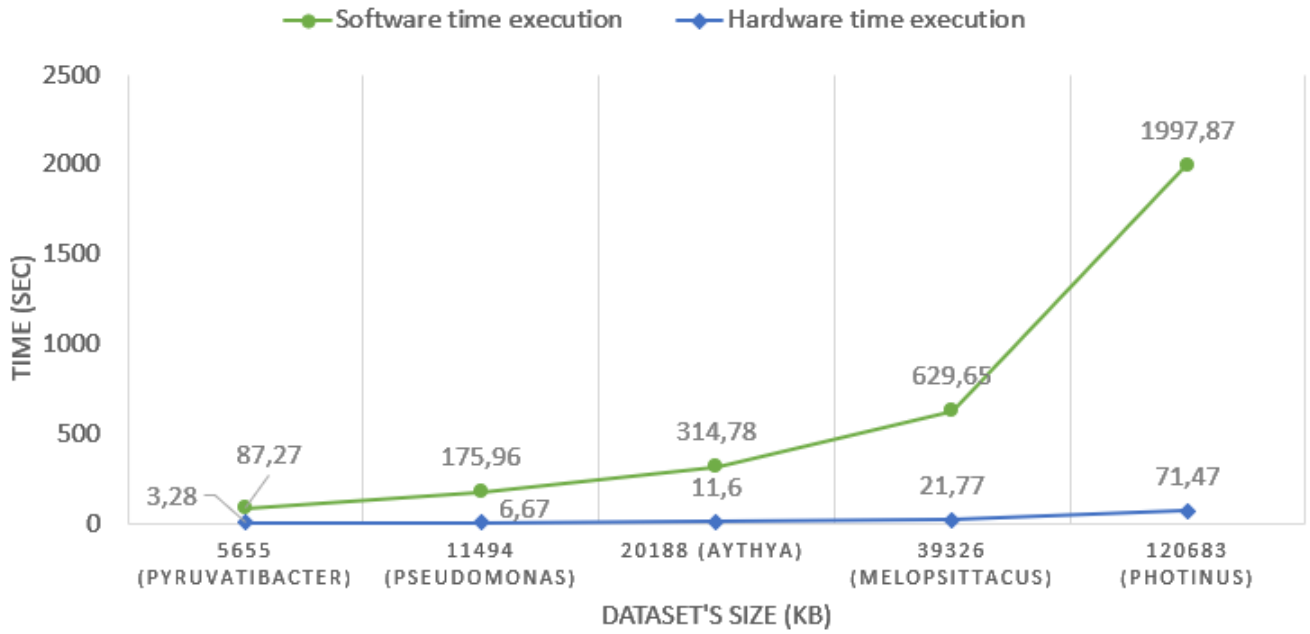

FIGURE 5.2: RMF's measured execution times both in software and hardware (with I/O operations from host/software)

and final implementation of the design. The execution times increased exponentially as the file size increased and the overall difference increased the same for larger dataset's size because the O(N^2) time complexity of the algorithm.

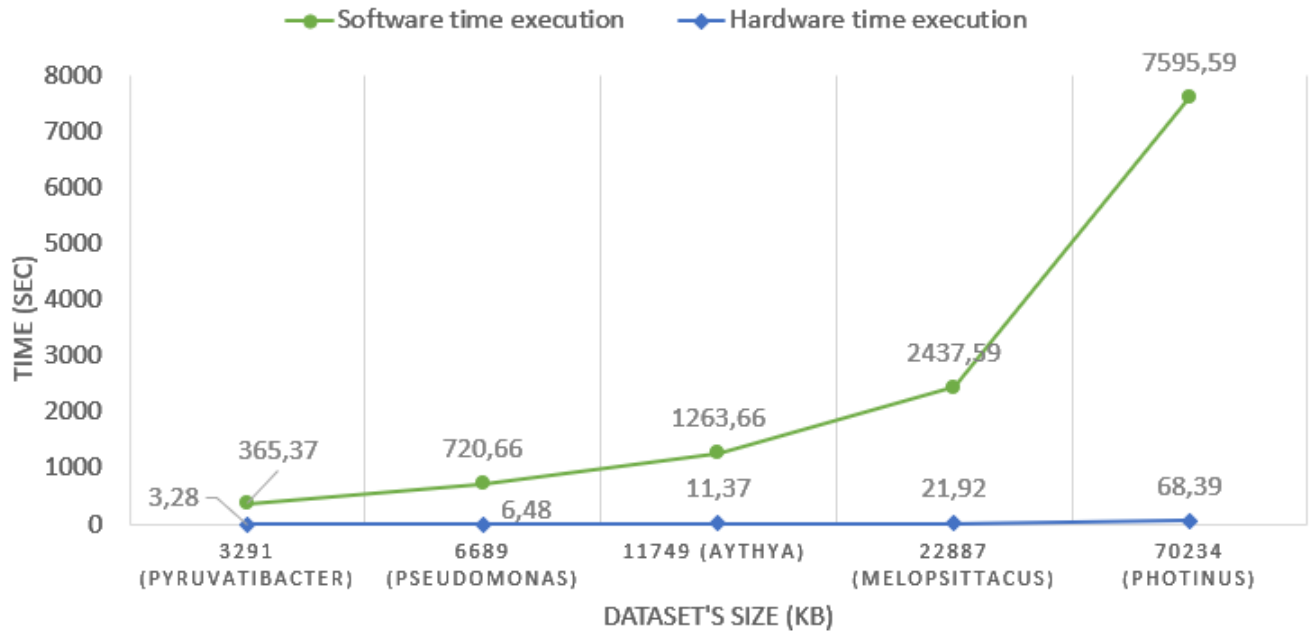

FIGURE 5.3: RMF's measured execution times both in software and hardware (with I/O operations from host/software)

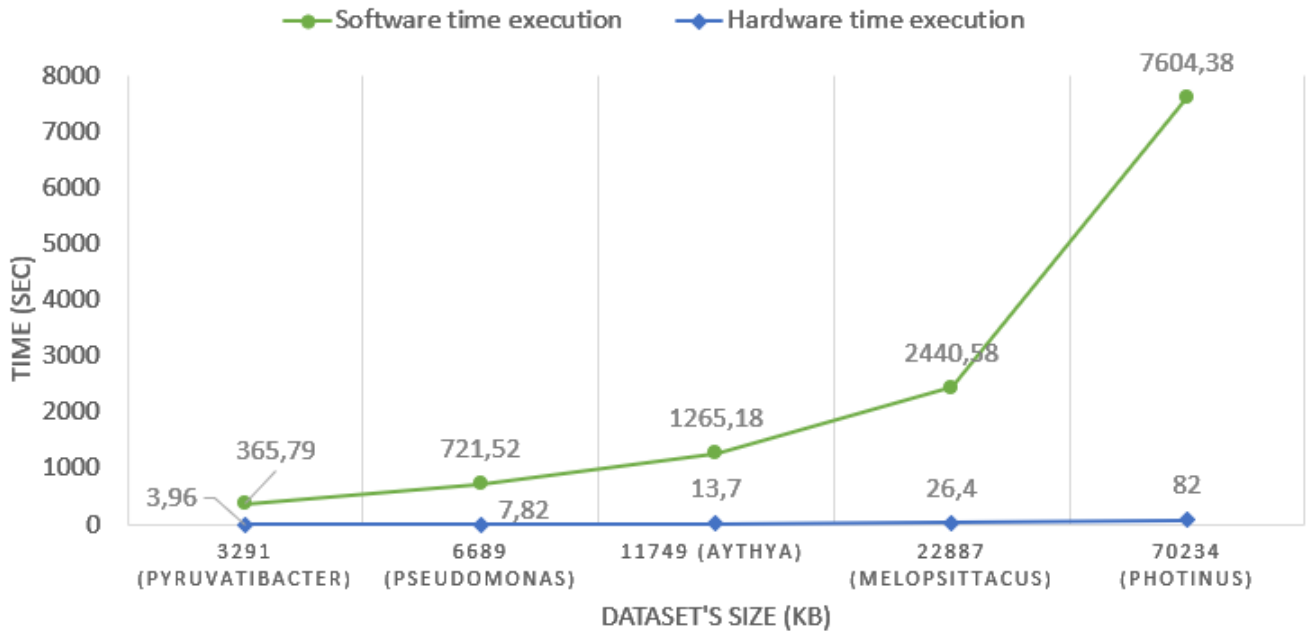

FIGURE 5.4: RMF's measured execution times both in software and hardware (with I/O operations from host/software)

In bar charts 5.5 and 5.6 we present the overall speedup that the whole process gain, included the velvet genome assembler processing that do the assembly, both in initial and final implementation of our RMF.

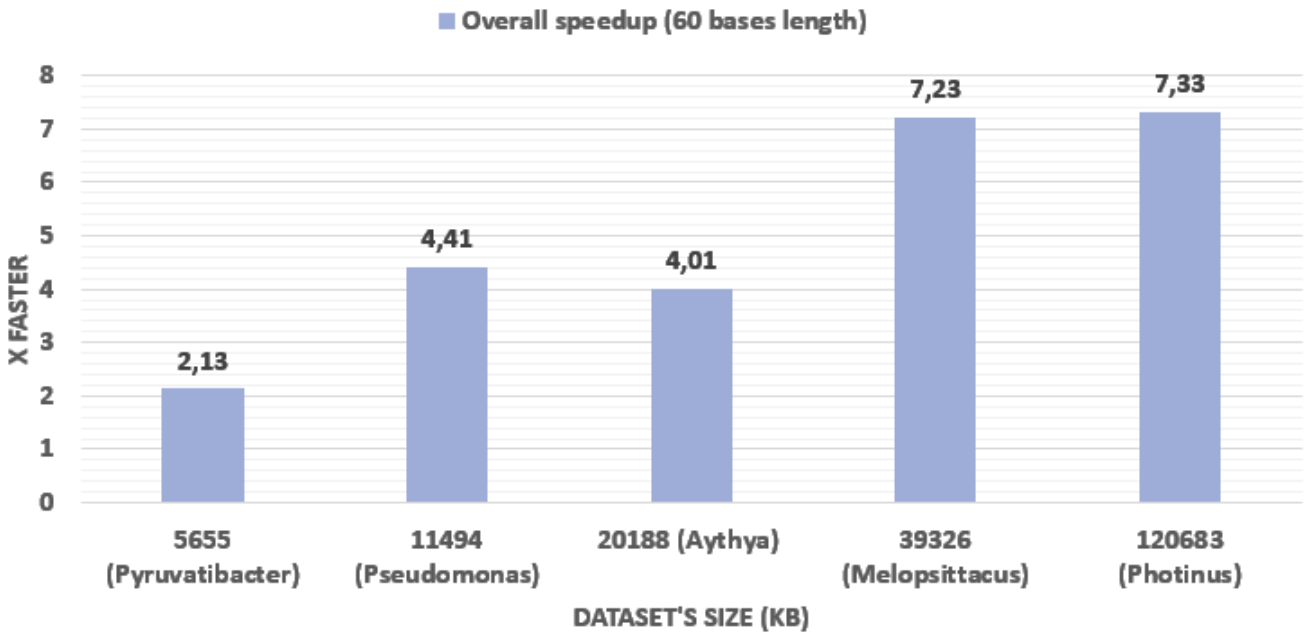

FIGURE 5.5: The overall speedup of the whole process included the Velvet assembly stage and our initial filtering implementation.

As we can see the overall speedup increased as the input dataset's size increased until it reached a peak point and then tapers down, except some

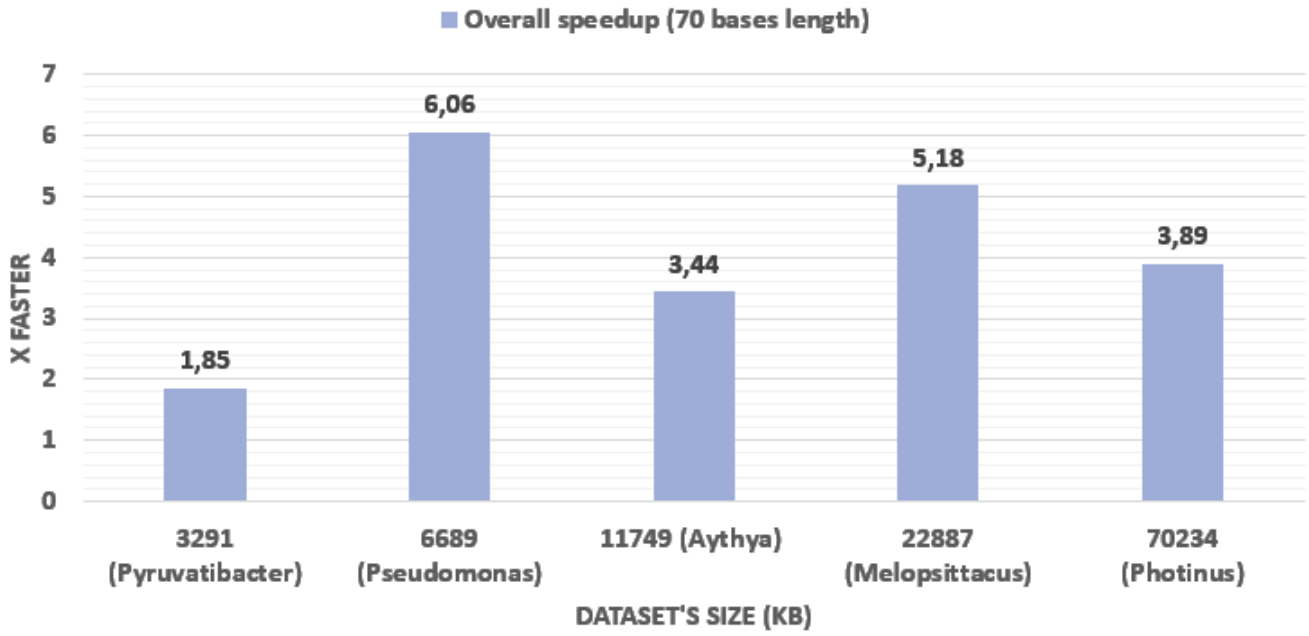

FIGURE 5.6: The overall speedups of the whole process included the Velvet assembly stage and our final filtering implementation.

of cases such as Pseudomonas original dataset in our final implementation which gain a significant x6,06 speedup in assembly with our filtering involved. The speedup ratio of big datasets found from experiments that it is not a very bad result as we saw that for a big dataset such as the Photinus Piralis (Chapter 4), the assembly process gain a speedup of the order of hours ($\approx 4x$ faster), which is very useful by reducing the processing time of the Velvet run.

5.2 Future Work

Further development about our work may include other optimizations or different hardware design. The main functionality of our algorithm is to make a preprocessing in the input dataset to improve the execution time of the assembly stage. In our case we used the Velvet genome assembler and a future work may can deal with the implementation of an add-on feature to the assembler. The reads matching filtering of our design may implemented as an add-on to attached in the input phase of the Velvet and make the redundancy job of the input dataset (cover and throw the repeats). This design has the advantage that it is independent of the kind of assembly or the assembler. So another use of our design may includes another genome assemblers such as Spades [2] or others.

Another approach, may include a work with a multi-instantiation of our kernel into the FPGA (or in multi-FPGAs system). As we saw in the Chapter 3 our kernel occupies approximately $\approx$ 17% of total space of the Alveo U50

for the initial implementation and approximately $\approx$ 17% for our final implementation. A nice approach is to implement a double or triple instance of the same kernel in order to share the processing job of the one kernel. Every kernel could run the process pipelined with the others (in the same datapath connecting the instances each other) and we could have increased speedups. This design implemented in this work but it did not manage to fit in the Alveo U50 in terms of limitations in card's resources. The further implementation on this could be a bigger FPGA included more resources or a multi-FPGAs system in order to combine multitudinous kernel instantiation.

Concerning the scores table's implementation that we managed to implement, there are many different approaches and implementation to follow in order to make it functionally and quality. We made a software only implementation with bad time execution results and following the plan we had in order to implement it, we saw that it was not worth the design. On the other hand a more detailed implementation plan can give nice results and can under certain conditions give a better quality concerning the outputs.

References

[1] Jill Adams. "Sequencing human genome: the contributions of Francis Collins and Craig Venter". In: *Nature Education* 1.1 (2008), p. 133.

[2] Anton Bankevich et al. "SPAdes: a new genome assembly algorithm and its applications to single-cell sequencing". In: *Journal of computational biology* 19.5 (2012), pp. 455–477.

[3] *BLAST: Basic Local Alignment Search Tool*. URL: https://blast.ncbi.nlm.nih.gov/Blast.cgi.

[4] *Bowtie, An ultrafast memory-efficient short read aligner*. URL: http://bowtie-bio.sourceforge.net/index.shtml.

[5] Chuming Chen et al. "Software for pre-processing Illumina next-generation sequencing short read sequences". In: *Source code for biology and medicine* 9.1 (2014), pp. 1–11.

[6] Rayan Chikhi et al. "On the representation of de Bruijn graphs". In: *Journal of Computational Biology* 22.5 (2015), pp. 336–352.

[7] Nafsika Chrysanthou et al. "Parallel accelerators for GlimmerHMM bioinformatics algorithm". In: *2011 Design, Automation & Test in Europe*. IEEE. 2011, pp. 1–6.

[8] Grigorios Chrysos et al. "Opportunities from the use of FPGAs as platforms for bioinformatics algorithms". In: *2012 IEEE 12th International Conference on Bioinformatics & Bioengineering (BIBE)*. IEEE. 2012, pp. 559–565.

[9] Grigorios Chrysos et al. "Reconfiguring the bioinformatics computational spectrum: Challenges and opportunities of fpga-based bioinformatics acceleration platforms". In: *IEEE Design & Test* 31.1 (2013), pp. 62–73.

[10] Steven Derrien and Patrice Quinton. "Hardware acceleration of HMMER on FPGAs". In: *Journal of Signal Processing Systems* 58.1 (2010), pp. 53–67.

[11] *DNA and proteins are key molecules of the cell nucleus from Friedrich Miescher (1844-1895)*. URL: http://www.dnaftb.org/15/bio.html.

[12] *Download BLAST Software and Databases*. URL: https://blast.ncbi.nlm.nih.gov/Blast.cgi?CMD=Web&PAGE_TYPE=BlastDocs&DOC_TYPE=Download.

[13] *File Format Guide from NCBI*. URL: https://www.ncbi.nlm.nih.gov/sra/docs/submitformats.

[14] David J Hall. "Phi X 174, the first DNA-based genome to be sequenced in 1977." In: (2013).

[15] *HMMER: biosequence analysis using profile hidden Markov models*. URL: http://hmmer.org/.

[16] Matina Lakka et al. “Reconfigurable Computing IP Cores for Multiple Sequence Alignment.” In: *BIOINFORMATICS*. 2011, pp. 216–221.

[17] Zhenyu Li et al. “Comparison of the two major classes of assembly algorithms: overlap–layout–consensus and de-bruijn-graph”. In: *Briefings in functional genomics* 11.1 (2012), pp. 25–37.

[18] Nicholas M Luscombe, Dov Greenbaum, and Mark Gerstein. “What is bioinformatics? A proposed definition and overview of the field”. In: *Methods of information in medicine* 40.04 (2001), pp. 346–358.

[19] Elaine R Mardis. “Next-generation DNA sequencing methods”. In: *Annu. Rev. Genomics Hum. Genet.* 9 (2008), pp. 387–402.

[20] Nathaniel McVicar et al. “FPGA acceleration of short read alignment”. In: *arXiv preprint arXiv:1805.00106* (2018).

[21] *Microprocessors & Hardware Laboratory of TUC*. URL: https://www.mhl.tuc.gr/en/home.

[22] *Polymerase Chain Reaction (PCR)*. URL: https://www.ncbi.nlm.nih.gov/probe/docs/techpcr/.

[23] Mihai Pop. “Genome assembly reborn: recent computational challenges”. In: *Briefings in bioinformatics* 10.4 (2009), pp. 354–366.

[24] Sara Reardon. “A complete human genome sequence is close: how scientists filled in the gaps.” In: *Nature* (2021).

[25] F Sanger. “Frederick Sanger—Biographical”. In: *Nobelprize. org* (2005), pp. 1–4.

[26] Haixiang Shi et al. “A parallel algorithm for error correction in high-throughput short-read data on CUDA-enabled graphics hardware”. In: *Journal of Computational Biology* 17.4 (2010), pp. 603–615.

[27] Euripides Sotiriades and Apostolos Dollas. “A general reconfigurable architecture for the BLAST algorithm”. In: *The Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology* 48.3 (2007), pp. 189–208.

[28] Daniel Summerer et al. “Microarray-based multicycle-enrichment of genomic subsets for targeted next-generation sequencing”. In: *Genome research* 19.9 (2009), pp. 1616–1621.

[29] *The Accepted Genome Assembly Data Formats*. URL: https://ena-docs.readthedocs.io/en/latest/submit/fileprep/assembly.html#.

[30] *The Alveo U50 Acceleration Card*. URL: https://www.xilinx.com/products/boards-and-kits/alveo/u50.html#specifications.

[31] *The Human Genome Project*. URL: https://www.genome.gov/human-genome-project.

[32] *The NCBI database search engine*. URL: https://www.ncbi.nlm.nih.gov/genome/browse#!/overview/.

[33] *The SOAPdenovo assembler*. URL: https://www.animalgenome.org/bioinfo/resources/manuals/SOAP.html.

[34] *The Velvet Software*. URL: https://www.ebi.ac.uk/~zerbino/velvet/.

[35] *The Vitis HLS optimization techniques*. URL: https://www.xilinx.com/html_docs/xilinx2020_2/vitis_doc/vitis_hls_optimization_techniques.html#ttf1584326469577.

[36] *The Vitis unified software platform*. URL: https://www.xilinx.com/products/design-tools/vitis.html.

[37] *Using Arbitrary Precision Data Types*. URL: https://www.xilinx.com/html_docs/xilinx2020_2/vitis_doc/use_arbitrary_precision_data_type.html.

[38] B Sharat Chandra Varma, Kolin Paul, and Mundanthra Balakrishnan. *Architecture exploration of FPGA based accelerators for BioInformatics applications*. Springer, 2016.

[39] B Sharat Chandra Varma et al. "FAssem: FPGA based acceleration of de novo genome assembly". In: *2013 IEEE 21st Annual International Symposium on Field-Programmable Custom Computing Machines*. IEEE. 2013, pp. 173–176.

[40] Xilinx. *Virtex-6 Family Overview*. 2015. URL: https://www.xilinx.com/support/documentation/data_sheets/ds150.pdf.

[41] *Xilinx, Semiconductor manufacturing company*. URL: https://www.xilinx.com/.

[42] Wenyu Zhang et al. "A practical comparison of de novo genome assembly software tools for next-generation sequencing technologies". In: *PloS one* 6.3 (2011), e17915.