TECHNICAL UNIVERSITY OF CRETE

DIPLOMA THESIS

FPGA-Based System Design for Applications of de Bruijn Graphs

Author:
Emmanouil-Eleftherios Rompogiannakis

Thesis Committee:
Prof. Apostolos Dollas
Assoc. Prof. Sotirios Ioannidis
Dr. Euripides Sotiriades

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

in the

School of Electrical and Computer Engineering
Microprocessor and Hardware Laboratory

March 4, 2022

TECHNICAL UNIVERSITY OF CRETE

School of Electrical and Computer Engineering

Abstract

FPGA-Based System Design for Applications of de Bruijn Graphs

by Emmanouil Eleutherios ROMPOGIANNAKIS

The mathematical properties of **De Bruijn graph** [1] were originally introduced in 1951 by the Dutch mathematicians Tanja van Aardenne-Ehrenfests and Nicolaas Govert de Bruijn. The De Bruijn graph is a directed graph representing overlaps between sequences of symbols; it has several uses in the field of telecommunications in protocols and networks and in the field of Bioinformatics, specifically in De novo genome assembly.

The properties of De Bruijn graph and its promising uses in De novo genome assembly have been presented in several scientific articles [2] [3]. In this thesis we have implemented an FPGA-based prototype hardware system for de Bruijn graph applications in de Novo genome assembly. We used the Russian genome assembler named SPAdes13.0 [4] as a case study for the use of de Bruijn Graphs. The SPAdes.13.0 genome assembler is a current-generation tool, and it is widely used in the field. The SPAdes.13.0 is also used for the verification of our experimental results. The data sets used in this thesis come from to European Nucleotide Archive (ENA) [5]. The FPGA Alveo U50[6] has been used as the target technology for experimental results in this thesis. The resulting speedup is modest (up to 1.14x-1.35x) for small data sets and the system has worse performance than SPAdes for large data sets, the bottleneck being the resources and the memory subsystem. Different accelerator cards with more storage capacity and resources could better exploit parallelism with more compute units. Thus, this thesis is more of a first-generation feasibility study, and can form the baseline for future accelerator architectures.

TECHNICAL UNIVERSITY OF CRETE

School of Electrical and Computer Engineering

Abstract

FPGA-Based System Design for Applications of de Bruijn Graphs

by Emmanouil Eleutherios ROMPOGIANNAKIS

Οι μαθηματικές ιδιότητες του γράφου **De Bruijn**[1] παρουσιάστηκαν το 1951 από τους Ολλανδούς μαθηματικούς Tanja van Aardenne-Ehrenfest και Nicolaas Govert de Bruijn. Ο γράφος De Bruin είναι ένας κατευθυντικός γράφος που αναπαριστά επικαλύψεις μεταξύ ακολουθιών από σύμβολα. Οι γράφοι De Bruijn έχουν αρκετές εφαρμογές σε Τηλεπικοινωνίες (πρωτόκολλα και δίκτυα), και σε Βιοπληροφορική, ειδικά σε De novo genome assembly.

Οι ιδιότητες του γράφου De Bruijn και οι διαφαινόμενες χρήσεις του στο De novo genome assembly αναπτύσσονται σε αρκετά επιστημονικά άρθρα[2][3]. Στην παρούσα διπλωματική εργασία υλοποιήσαμε ένα πρωτότυπο σύστημα βασισμένο σε αναδιατασσόμενη λογική για εφαρμογές γράφων De Bruijn σε De Novo genome assembly.Χρησιμοποιήσαμε τον Ρωσικό genome assembler ονόματι SPAdes.13.0[4] επειδή χρησιμοποιεί γράφους De Bruijn, ανήκει σε τρέχουσας γενιάς τεχνολογία, και είναι πολύ διαδεδομένο στην επιστημονική κοινότητα. Το ίδιο εργαλείο χρησιμοποιήσαμε για την επαλήθευση των αποτελεσμάτων μας. Τα αρχεία που χρησιμοποιυνται στην διπλωματική εργασία προέρχονται από το Ευρωπαικό Νουκλεοτιδικό Αρχείο[5]. Το σύστημα FPGA Alveo U50[6] που χρησιμοποιείται στα πειράματα της διπλωματικής πετυχαίνει μία μέτρια επιτάχυνση 1.14x-1.35x για τα μικρα αρχεία ενώ για τα μεγάλα αρχεία έχει χειρότερη επίδοση από το SPAdes.13.0, αλλά μεγαλύτερες FPGA οι οποίες έχουν περισσότερους πόρους μπορούν να πετύχουν μεγαλύτερο παραλληλισμό με υλοποίηση περισσότερων μονάδων υπολογισμού. Επομένως η παρούσα διπλωματική εργασία μπορεί να χρησιμεύσει σαν μία πρώτη αρχιτεκτονική αναφοράς.

Acknowledgements

During the implementation of the current 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.

First of all, I would like to thank my supervisor, Professor Apostolos Dollas, whose experience and expertise in researching themes was crucial for the completion and presentation of my diploma thesis. Also, I would like to thank him for his patience, his valuable guidance to plan my thesis and his tips that help me to solve different problems. Further, his insightful feedback pushed me to sharpen my thinking and brought my work to a higher level. Also, I gratefully acknowledge the contribution of my committee's members, who, including Professor Apostolos Dollas, are Professor Sotirios Ioannidis and Dr. Evripides Sotiriadis.

In addition, I would like to honor my colleague on my work Dr. Evripides Sotiriadis who guided me during the whole process of my thesis and gave me useful tips in order to choose the right direction and successfully complete my thesis. I appreciate his patient support and all of the opportunities I was given to further my research. I want to thank Mr Palvo Malakonaki who helped me to understand the way the tools operate and for his tips. Pavlo, I am glad to have worked with you and learn from you.

I would like to thank my parents for their psychological assistance to complete my thesis. Last but not least, I could not have completed this dissertation without the support of my brothers, girlfriend, and close friends, who provided stimulating discussions as well as happy distractions to rest my mind besides my research.

Contents

List of Figures

List of Tables

List of Algorithms

List of Abbreviations

ALU	**A**rithmetic **L**ogic **U**nit
ASIC	**A**pplication **S**pecific **I**ntegrated **C**ircuit
BRAM	**B**lock **R**andom **A**ccess **M**emory
CPU	**C**entral **P**rocessor **U**nit
CS	**C**omputer **S**cience
DDR4	**D**ouble **D**ata **R**ate type texbf4 memory
DRAM	**D**ynamic **R**andom **A**ccess **M**emory
DSP	**D**igital **S**ignal **P**rocessor
FF	**F**lip **F**lops
FPGA	**F**ield **P**rogrammable **G**ate **A**rray
GDDR6	**G**raphics **D**ouble **D**ata **R**ate type **6** memory
GPU	**G**raphic **P**rocessor **U**nit
HBM	**H**igh **B**andwidth **M**emory
HDL	**H**ardware **D**escription **L**anguage
HLS	**H**igh **L**evel **S**ynthesis
HPC	**H**ight **P**erformance **C**omputing
LUT	**L**ook **U**p **T**able
MPSoC	**M**ulti **P**rocessor **S**ystem **o**n **C**hip
PL	**P**rogrammable **L**ogic
PS	**P**rocessing **S**ystem
RAM	**R**andom **A**ccess **M**emory
SDK	**S**oftware **D**evelopment **K**it
SIMD	**S**ingle **I**nstruction **M**ultiple **D**ata
SSE	**S**treaming **S**IMD **E**xtensions
SSD	**S**olid **S**tate **D**rive
TDP	**T**hermal **D**esign **P**ower
URAM	**U**ltra **R**andom **A**ccess **M**emory
USD	**U**nited **S**tates **D**ollar

Chapter 1

Introduction

De Bruijn graphs [1] were originally introduced in 1951 by the Dutch mathematicians Tanja van Aardenne-Ehrenfests and Nicolaas Govert de Bruijn. The De Bruijn graph is a directed graph representing overlaps between sequences of symbols. There are three important uses of de Bruijn graphs, as follows:

1. De Bruijn graph used in Bioinformatics, specifically in De novo genome assembly for the representation of sequencing reads into a genome. There are several genome assemblers which use a de Bruijn graph as the main structure to store parts of the genome as they read them from the data sets.

2. Some grid network topologies are de Bruijn Graphs. A grid network is a computer network consisting of a number of computer systems connected in a grid topology. A parallel computing cluster or multicore processor is often connected in regular interconnection network such as a de Bruijn graph.

3. Last but not least,the distributed hash table protocol Koorde uses a De Bruijn graph for routing. The distributed hash table (DHT) is a distributed system that provides a lookup service similar to a hash table: key-value pairs are stored in a DHT.In peer-to-peer networks, Koorde is a distributed hash table, the nodes are the peers and the values are the messages between the peers. In a d-dimensional de Bruijn graph, there are 2^d nodes, each of which has a unique d-bit ID. The node with ID i is connected to nodes 2i modulo 2^d and 2i+1 modulo 2^d. Thanks to this property, the routing algorithm can route to any destination in d hops

1.1 Motivation

An important motivation for the current thesis is that the simplification of De Bruijn graph is an computationally intensive algorithm which can be executed in parallel and have several important applications such as in genome assembly. Nowadays, genome assembly is used daily for the detection of corona virus in PCR tests, so it would be useful to contribute in the acceleration of the process.The development of the modern hardware tools motivate us because they have fast data transceivers, large amount of resources, storage space. In addition, there are tools that produce important information about the implementation and visualize the hardware system, so make it easy for the user to understand and manage the hardware system. Thus, it is great inspiration to apply them in practice and present the results of the modern tools. In our case, we have to manage a priority queue which over time forms a challenging problem for hardware systems, so we wanted to see how designs with present-day CAD tools would fare in the problem-at-hand.

1.2 Thesis Contributions

In the current thesis we work on a simplification algorithm of de Bruijn graphs which have great impact in the execution time of genome assembly. We focus on an algorithm named Bulge Removing algorithm which removes bulge errors from de Bruijn graph. At the beginning, the algorithm get as input from the constructed de Bruijn graph information which is used in order to calculate the distance between a starting vertex with its neighbours, and then their neighbours, until the end of the path. The calculated distances used in order to form the paths that may have to removed from the graph in case that satisfy some thresholds. We implemented in hardware the calculation of the distances, with the rest of the steps being implemented in software.

In the current thesis we study in detail a state of the art software named SPAdes.13.0[4] which uses de Bruijn graphs. It is created in the Center for Algorithmic Biotechnology[7] which is a part of the Russian Institute of Translational Biomedicine SPbU.At the beginning, I designed our architecture according to the suggestions of the tools in order to make an initial design in a short time, implement it in the hardware system, and optimize it subsequently. In the beginning, the hardware implementation had a very low performance, so we decided to make changes in the architecture in order to

optimize it. Due to the pandemic it was not possible to have in-person help on the tools, and so I had to learn the tools largely on my own, and then to make significant changes in the implementation to improve performance. Even so, we managed to implement a functional, fully operational system in the FPGA, which produces correct results. We use SPAdes.13.0 as verification platform in order to verify the correctness of FPGA results. Alveo U50 [6] accelerator card used for the experimental results of the current thesis. The hardware implementation achieved 1.14x-1.35x speedup over SPAdes.13.0 for the smallest data sets but it has worse performance for the largest data sets vs. SPAdes.13.0. We managed to implement only 5 compute units due to lack of storage space of Alveo U50, so we looked into other accelerator cards with more storage space in order to compare their results against SPAdes.13.0 results. In conclusion, the tools ease the user in hardware design and implementation but it can not design an optimal system - the designer has to change the design in order to improve performance. The main contribution of this thesis is a baseline architecture for the problem-at-hand, which can be further improved and parallelized on larger platforms in subsequent generations of system design.

1.3 Thesis Outline

This thesis follows the outline, presented below:

- **Chapter 2 - Theoretical Background:** it describes the necessary theoretical background that we need to know about de Bruijn Graphs, genome assembly and the genome assembler that we use.
- **Chapter 3 - Study and Design:** it has the profiling results and Related Works to this thesis, as well as the main algorithm of Bulge removing and the approach of this thesis.
- **Chapter 4 - FPGA Implementation:** it describes the hardware tools and the FPGA that we use in the current thesis. Furthermore, it presents the data path configurations which we implement in the FPGA, and block diagrams for each configuration.
- **Chapter 5 - Results:** it presents the results of the implemented system and compares these results with the execution time of SPAdes. Furthermore we make performance projection on other platforms in order to estimate performance on high-end systems.

- **Chapter 6 - Conclusions and Future Work:** this chapter presents conclusions about the final results and the projection on other platforms. Also, we suggest changes that will improve the final implementation.

Chapter 2

Theoretical Background

The second chapter of thesis is about to understand the theoretical background that will be used in the thesis. The first part of the chapter describes the process of the **De Bruijn Graph**, explain the usage of this process in Genome Assembly . Also, referred information about **SPAdes.13.0** and related articles that were studied in order to inspire us. Further, presented the tools that we use in thesis which are Vitis Unified Software Platform and Vitis High-Level Synthesis (HLS). In addition, referred specifications of ALVEO U50 accelerator card that used for the experimental results.

2.1 De Bruijn Graph

De Bruijn graph [1] was originally introduced in 1951 by the Dutch mathematicians Tanja van Aardenne-Ehrenfests and Nicolaas Govert de Bruijn, the **De Bruijn Graph** construction begins with the definition of the the value of the k which determine the number of consecutive bases in one read of the assembler and each node represents a k-mer. There will be a directed arc between two nodes if there is an overlap with k–1 bases and continuously emerge in one read. A read with the length of r can be divided into r–k+1 overlapping k-mers. The de Bruijn graph is classified into two types, **Hamiltonian** and **Eulerian** approach, according to the method of expressing the nodes and edges.

In **Hamiltonian** [8] [9] [10] approach, the k-mers are the nodes, whereas they are the edges in the Eulerian approach. The Hamiltonian graph approach the node is the sequence and the edge is the overlap. In Hamiltonian graph approach, the sequences are assembled by finding Hamiltonian paths that traverse all nodes, each of which is visited only once. This scenario is known as the NP-complete problem when the number of nodes is not trivial.

Normally, the computational complexity of finding the Hamiltonian paths is $O(2^n m)$,where m is the total number of nodes, and n is the number of branching nodes . The **Hamiltonian** approach is widely used in de novo assembler such as SOAPdenovo[11], Velvet[12].

In **Eulerian** [8] [9] [10] graph approach, the sequences are assembled by finding Eulerian paths that traverse all edges, each of which is visited only once without simplification in polynomial time $O(2n)$. The **Eulerian** approach is widely used in de novo assembler such as SPAdes[13] and ALLPATHS[14]. The Eulerian de Bruijn graph based assemblers generally perform better in the assembly of a large genome than the Hamiltonian graph based assemblers.

For example if the read of the assembler is **ATGGCGTGCAATG** then we will have the form that seems in Figure 2.1 for the Hamiltonian graph and Figure 2.2 for the Eulerian graph. Starting from the arc with the weight 1 and keep going to arc with weight 2-3-4 etc. In the end we will take back the right form of the read (**ATG** → ATG**G** → ATGG**C** → ATGGC**G** → ATGGCG**T** → ATGGCGT**G** → ATGGCGT**C** → ATGGCGTC**A** → ATGGCGTCA**A** → ATGGCGTCAA**T** → ATGGCGTCAAT**G**).

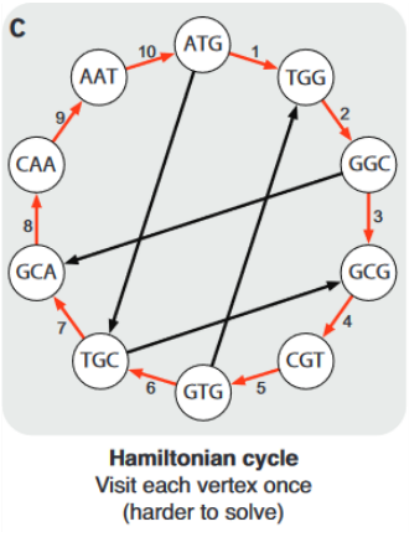

FIGURE 2.1: Hamiltonian Path.

Reference: How to apply de Bruijn graphs to genome assembly

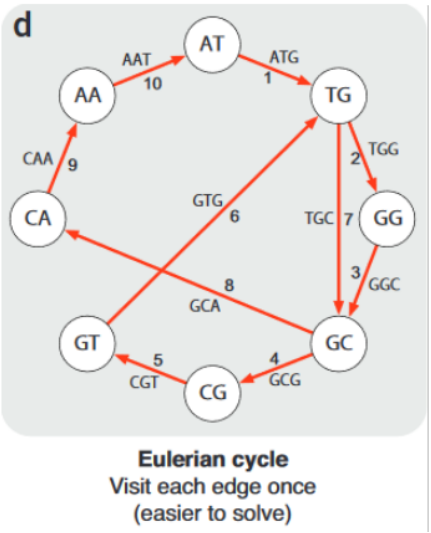

FIGURE 2.2: Eulerian Path.

Reference: How to apply de Bruijn graphs to genome assembly

2.1.1 Multisized De Bruijn Graph

Assembly methods based on de Bruijn graphs begin,somewhat counter-intuitively, by replacing each read with the set of all-overlapping sequences of a shorter,fixed length . The length is often denoted by **k**, and the sequences **k-mers**. The **value of k** is important for constructing de Bruijn graph because:

- A **large value of k** will remove some short repetitive regions while reducing the number of nodes in de Bruijn graph, but will give rise to more unconnected sub-graphs which means that the number of gap regions increases.
- A **small value of k** will reduce some gap regions while increases the connectivity of de Bruijn graph, but will add more nodes and increase short repetitive regions.

Therefore, the value of **k** can not be too large or too small but we should determine the value of the **k** so have the best results.

2.2 Genome Assembly

Genome assembly [15] constitutes a process in Computational Molecular Biology in which a tool reads parallel many small length nucleotide sequences and putting them into correct order and aim to construct the right completed genome. The above process is split in two categories which are **Reference Guide Genome Assembly**[16] and the **De Novo Genome Assembly** [17].

In **Reference Guide Genome Assembly**[16] there is a reference genome, this means that the tool try to assemble a gene by reading sequences from a file and order them as the reference genome.

On the other hand in **De Novo Genome Assembly** [17] there isn't a reference genome and the tool every time after the parallel reading of the nucleotide sequences try to order the reads via overlaps and in the end the target is to produce a completed and correct genome without gaps. **De Novo Genome Assembly** is a very interesting section of Genome Assembly and at the same time very difficult because of the no reference genome to struct. Genome Assemblers are the tools which construct from the many small size nucleotide sequences into a complete genome.

We refer more information about genome assembly in appendix A

2.3 Spades.3.13.0

Spades.3.13.0 [4] is the software that used in the dissertation for genome assembly of the genome. It is a next generation genome assembler that created at the Center for Algorithmic Biotechnology[7] in Russia, the source code of **SPAdes.3.13.0** is in **C++** program language.Also, have some modules in **Python**. It works with Illumina or Ion Torrent reads and is capable of providing hybrid assemblies using PacBio, Oxford Nanopore reads with Illumina or Ion Torrent reads . Furthermore, it supports **single-end**, **mate-pair** and **paired-end** reads, it designed for small size genomes such as fungal,bacterial not for big size genomes such as mammalian.

Illumina, **Ion Torrent**, **PacBio**,**Oxford Nanopore** are research companies which do research in Next Generation Sequencing Technology and create tools which used to produce **fastq.gz** format files for Genome Assembly.

For **single end**, **paired-end** and **mate-pair** reads at first break up the DNA into fragments of 200-500bps. The term **single end** read used when the read starts from the one end of the fragment. The term **paired end** read used when we have 2 reads, the first read starts from the one end of the fragment and the second read starts from the other end of the fragment. The two reads that arise is a **paired end read**. The third type of reads are the **mate-pair** reads which is exactly the same as the paired-end reads but the difference is that the size of the fragments in mate-pair are larger than the paired-end. The usage of short insert and long insert fragment, the advantages and the disadvantages of each different read will explained in next.

We refer more information related to SPAdes.13.0 in appendix B

2.4 Related work

De bruijn graphs is an important part of De novo genome assembly,there are plenty scientific articles about de Bruijn graphs and ways to succeed speedup on execution time of de Novo genome assembly. Also, in the most of the articles use FPGA in order to succeed speedup . We present some of them at next:

Rayan Chikhi and Guillaume Rizk [18] propose a new encoding of the de Bruijn graph, which occupies an order of magnitude less space than current representations.The encoding is based on a Bloom filter, with an additional structure which mark nodes have already been visited in order to remove

critical false positives. They implement a new memory-efficient method for de novo assembly named Minia which compared against ABySS[19] and SOAPdenovo [11] about magnitude space. In conclusion,they execute a human genome data set, Minia allocates 59 time less space than ABySS and 25 times less space than SOAPdenovo and all of them produce correct contigs.

Carl Poirier, Benoit Gosselin and Paul Fortier [3] presented an FPGA implementation of a DNA assembly algorithm, called Ray[20] which use de Bruijn graphs, initially developed to run on parallel CPUs.In article referred the modifications that done in the five steps of Ray algorithm in order to optimize and implement each step in FPGA. They compare the execution time of CPU Intel Core i7-4770 against Intel FPGA Altera OpenCL SDK and the power consumption of each platform.In conclusion,upon running the new program on some datasets, it becomes clear that FPGAs are a very capable platform that can fare better than the traditional approach, both on raw performance and energy consumption.

At Nanyang Technological University, in Singapore Haixiang Shi, Bertil Schmidt, Weiguo Liu, Wolfgang Müller Wittig [21] present a scalable parallel algorithm for correcting sequencing errors in high-throughput short-read data so that error-free reads can be available before DNA fragment assembly, which is of high importance to many graph-based short-read assembly tools.The algorithm is based on spectral alignment and uses the Compute Unified Device Architecture (CUDA) programming model .They present 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 pre-processing and parallelized error correction compared to the publicly available Euler-SR program.

Anand Ramachandran and his team [22] develop the first FPGA-based error correction accelerator for Illumina reads called FADE.FADE uses BLESS [23] as base algorithm, which is one of the most accurate DNA error-correction tools. The main data structure used in BLESS is a Bloom filter [24]. BLESS counts the occurrence of all k-mers by writing them into multiple files which consumes a large amount of time around 40% of total time. They implement hash functions for the process of k-mer counting in FADE which is a significant change in the algorithm because they do not need to wright in files any more. After the significant change of k-mer counting FADE is 28.3x-43x faster than BLESS.

At University of Hong Kong, Varma [2] chose to accelerate a pre processing algorithm on the FPGA to reduce the short read data for the CPU assembly algorithm.The simplified data sets used in the execution of Velvet genome assembler which use de Bruijn graphs. They reported a 13x speedup over the software. Furthermore, they conclude that the speedup depends from the quality of the input dataset. They also proposed an improved FPGA implementation exploiting the hard embedded blocks such as BRAMs and DSPs.

2.5 Vitis Unified Software Platform

It is the platform where we implement our host and kernel code in OpenCL and C program language respectively,also we debug,compile and run our implementation in this platform.For FPGA-based acceleration, the Vitis Unified Software Platform [25] lets us build a software application using an API, such as the OpenCL API, to run hardware kernels on accelerator cards, like the Xilinx Alveo Data Center acceleration cards that we use in this thesis . The Vitis core development kit also supports running the software application on an embedded processor platform running Linux,such as on Zynq UltraScale+ MPSoC devices. For the embedded processor platform, the Vitis core development kit execution model also uses the OpenCL API and the Linux-based Xilinx Runtime (XRT) to schedule the Hardware kernels and control data movement.

The Vitis core development kit tools support the Alveo U50, U200, U250, and U280 Data Center accelerator cards, as well as the zcu102_base, zcu104_base, zc702_base, and zc706_base embedded processor platforms. In addition to these off-the-shelf platforms, custom platforms are supported too.In our case we use Alveo U50 Data Center accelerator card in this dissertation.

2.6 Vitis High Level Synthesis (HLS)

Xilinx Vitis High-Level Synthesis (HLS) [26], is a tool included in the Xilinx Vitis Design Suite, allowing for a higher level of abstraction design of HDL systems. Vitis HLS synthesizes C/C++, SystemC and OpenCL functions into IP blocks, generating their VHDL and Verilog HDL designs that can then be implemented into hardware systems using Vitis and its Block Design tool.

HLS accepts non-hardware-optimized code, and its goal is to optimize them. So, provides a set of directives that can be used to optimize the code. Directives are optional and do not affect the behavior of the code. Further,they are not specialized into certain programming language while they generalized for all acceptable ones. Their correct usage can improve the performance of the implementation while the wrong usage of directive can worsen the performance of the code . Furthermore, constraints, like clock period, clock uncertainty, and FPGA target, are added to the HLS synthesized IP blocks to direct directives to resources and ensure the desired behavior and performance.

In addition Vitis High-Level Synthesis (HLS) produce as output useful information about the proportion of resources that utilized in the FPGA by the implementation such as LUTs,Block RAM,DSPs,Registers. Also, provides information about the loops of the configuration such as whether the loops can become pipeline, interval of the pipeline loop and any other optimizations that made in the configuration.In the best case the interval of a pipeline loop is 1. In case that interval is greater than 1 the tool give as output information about the reasons of interval greater than 1 in order to make appropriate changes to the implementation to succeed interval 1 for each loop.Furthermore, the tool refers the latency in clock cycles of the whole implementation and the latency of each module. The above information presented in a table after the finish of synthesis of the implementation.

2.7 The FPGA Perspective

The FPGA that we use is Alveo U50 [6] of Xilinx. The specific FPGA is available in the Microprocessor Hardware Lab and it is connected with the Vitis tool of the server. The Xilinx Alveo U50 Data Center accelerator cards provide optimized acceleration for workloads in financial computing, machine learning, computational storage, and data search and analytics. Built on Xilinx UltraScale+ architecture and packaged up in an efficient 75-watt, small form factor, and armed with 100 Gbps networking I/O, PCIe Gen4, and HBM, Alveo U50 is designed for deployment in any server.

Alveo accelerator cards are adaptable to changing acceleration requirements and algorithm standards, capable of accelerating any workload without changing hardware, and reduce overall cost of ownership.It follows in Figure 2.3 the view of the FPGA.

FIGURE 2.3: Alveo U50.
Reference: Xilinx Alveo U50 Data Center Accelerator Card

Also, in the Table 2.1 presented the specs of the Alveo U50 FPGA

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 S-RAM Capacity	28 MB
Internal S-RAM Total Bandwidth	24 TB/s
Clock Precision	IEEE 1588
Vitis Platform	Gen3x16 XDMA, Gen3x4 XDMA3
Maximum Total Power	75W

TABLE 2.1: Alveo U50 Specifications.

In the current Chapter referred several related articles that succeed acceleration in error correction of de Bruijn graphs with the use of older FPGA models. Thus, we want to try accelerate an error correction algorithm with the use of modern technology. In Chapter 3 described the error correction

algorithm which studied, the selected data sets and the profiling results of the specific algorithm.

Chapter 3

Study and Design

The meaning of the term **Study** is that we learn about an issue and try to understand it so afterwards be able to **Design** the solution of the issue. At the beginning, SPAdes.13.0 was selected as a reference software for this thesis which is a Russian genome assembler that use de Bruijn graph as main structure for genome assembly. SPAdes.13.0 is a popular tool in the field of Bioinformatics for genome assembly and used in several scientific articles because of its high accuracy in output results. Code of SPAdes.13.0 organised in many different folders which contains files in C++ language, so it was complicated to analyze the whole tool, understand correctly the way that works and find the proper algorithm for us. At last, we succeed to detect the proper algorithm and start profiling the algorithm manually with a library of C++ named <chrono>. We use the profiling results as guide for the design of our architecture.At the end of chapter we describe the modification that we did in the algorithm to make it easier for us to implement it in the FPGA and may improve its execution time.

3.1 Bulge Removing

In the first Stage, SPAdes read the data sets, cut the reads into k-mers and store them in a **De Bruijn Graph**. When the process of Graph Construction ends then follows the procedure of Graph Simplification, so SPAdes simplify the produced De Bruijn Graph. Over four Graph Simplification procedures **Bulge Removing**[4] is the one that consume the most of the time , around 90% of graph simplification and 30% of SPAdes Algorithm. **Bulge Removing** is a graph simplification process that SPAdes execute when from the same hub starts two or more paths and these two or more paths ends in the same

hub as it seems in Figure B.1 A. Bulge removing separated into four steps which represented in Figure 3.1

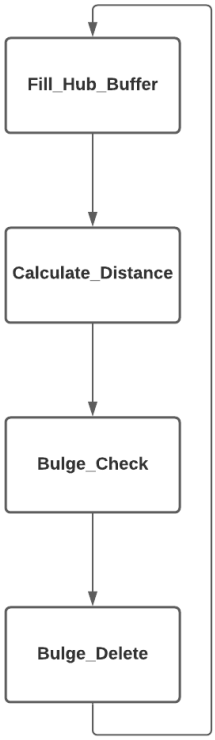

FIGURE 3.1: Bulge Removing Algorithm.

Some details about each step of the algorithm are necessary for the complete understanding of them and of the whole algorithm. So:

1. **Fill_Hub_Buffer** constitutes the first step of the algorithm in Figure 3.1, in this step 10.000 different hubs stored in a buffer in order to check if any of them is the start of a bulge error. This step executed until all the hubs are checked.

2. **Calculate_Distance** is the second stage of the algorithm in Figure 3.1, in this stage detected the neighbours of the starting hub and the distance between the starting hub and each neighbour. Also, after the detection of the neighbours of the starting hub, the algorithm detects the neighbours of the neighbours and their distance between them. So, in this step get known all the possible paths that starts from the starting hub and it is very useful for the next step.

3. In the begging of the third stage **Bulge_Check** used the Dijkstra Algorithm for searching the graph and find the paths from the ending hub

to the starting hub. The Dijkstra Algorithm is a greedy algorithm, so it uses the results of the previous step to determine the path that will create in order to reach from the ending hub to the starting hub. After the creation of the path follows the checking of an alternative path and in the end if the path fulfills the criteria to be a bulge error. The criteria for a path to be a bulge error are two:

(a) The length of the path must be less than 150 vertices("small")

(b) The difference between path that created from the Dijkstra Algorithm and the alternative path should be less or equal 3("similar")

Nevertheless, the criteria can be changed if it is needed for any occasion.

4. Last but not least, the **Bulge_Delete** stage executed only for the paths that are bulges, in case that the path creates a bulge error then deleted from the graph. Also before the operation of deletion of the path from the graph the edges and the vertices projected to the alternative path because they may contain useful information about the genome assembly so we shouldn't discard them. The projected edges and vertices may need in the second stage of the SPAdes but after the end of the second stage the projected edges and vertices deleted once for all.

The above steps repeated until all the starting hubs that have two or more outcoming edges checked for bulge error. As a result SPAdes do many times the same steps since it checks only 10.000 different hubs every time.

3.2 Data Sets Selection

The data sets that executed in SPAdes downloaded from the **European Nucleotide Archive (ENA)**[5] which is the European platform for the management,sharing,integration, archiving and dissemination of sequence data of European Molecular Biology Laboratory (EMBL) [5] .ENA is developed and operated under the support of the European Molecular Biology Laboratory (EMBL) and through grants from external bodies that include the European Commission, the British Biotechnology and Biological Sciences Research Council (BBSRC) and the Wellcome Trust (WT).The **European Nucleotide Archive (ENA)** captures and presents information relating to experimental workflows that are based around nucleotide sequencing. **ENA** have an open online database that contains variety of eukaryotic de novo sequence data sets. Data

arrive at ENA from a variety of sources. These include submissions of raw data, assembled sequences and annotation from small-scale sequencing efforts, data provision from the major European sequencing centres and routine and comprehensive exchange with their partners in the International Nucleotide Sequence Database Collaboration (INSDC).

We decide to work on human data sets which have high complexity and they contain more errors than prokaryotic datasets.Thus, our graph simplification algorithm will have more errors to remove from the graph. We select five human data sets with different data size so we can have clear results about the execution time that need each one. In the Table 3.1 presented all the data sets that we use for the profiling and the experiments.

Data Set Name	Data set Size(MB)	Total Reads	Total Base Pairs(Mbps)
DRR015506-1	**5**	**28.230**	**2**
DRR015506-2	**5**	**28.230**	**2**
DRR015482-1	**9**	**50.256**	**5**
DRR015482-2	**9**	**50.256**	**5**
DRR015499-1	**17**	**93.964**	**9**
DRR015499-2	**17**	**93.964**	**9**
DRR015480-1	**24**	**130.473**	**13**
DRR015480-2	**24**	**130.473**	**13**
DRR015512-1	**36**	**173.462**	**17**
DRR015512-2	**36**	**173.462**	**17**

TABLE 3.1: Data Sets.

The value of **Total Reads** and **Total Base Pairs** computed from a script in C language that I create, the name of the script is **Read-lines.c** and explained in the following subsection.Also, as we can see in the table 3.1 we have pairs of data sets, the **suffix '1'** means that the data set have **forward reads** and the **suffix '2'** means that the data set have **backward reads**. We need pair of data sets because use the **Pair end library** of the SPAdes.3.13.0 so it is necessary to use pair of data sets. Furthermore, we have variety of Data Size because we need to test different cases.

We create a script in C language which compute the **Total Reads** and the **Total Base Pairs** of every data set. At the beginning, the program count the

total number of the lines of data set and take advantage of the format that have all the **fastq files**. We know that a cluster have 4 lines and in these 4 lines there is one sequence read. So

$$TotalReads = \frac{TotalLines}{4}$$

and

$$TotalBasePairs = \frac{TotalReads \cdot ReadLength}{10^6}$$

.

3.3 Profiling

Profiling is a critical process for the thesis because define the part of the code that consume the most of the time in a project. Also, the results of the **profiling** can be used to calculate the percentage of execution time over total time of the algorithm and with the **Amdahl 's Law** about the parallelization we can determine the upper limit of the speedup that can get the specific code in parallel in the FPGA. The name of the server that used for the profiling is **kronos.mhl.tuc.gr (KRONOS)** which belongs to **MHL** sector of **Technical University of Crete**, in the following subsection presenting the specs of the computer in KRONOS server. Furthermore, we did **manual profiling** in SPAdes for the **Bulge Removing Algorithm** to calculate the execution time of the algorithm and the amount of time that consume each process of the algorithm in percentage over the total time . We use the C++ library **<chrono>** for manual profiling.

3.3.1 KRONOS Server Specs

As it referred above the specific subsection describe the specs of the server KRONOS computer. The specs presented in the table 3.2:

CPU	GPU	RAM(GB)	Hard Dive Size(TB)
Intel Xeon E5-2630 V4 @2.2 GHz	**Matrox Electronics System Ltd G200eR2**	**258**	**7.3**

TABLE 3.2: Server KRONOS Specs.

For the genome assembly the assembler use big amount of RAM because it stores all of the useful information such as the assembly graph in RAM, for big data sets SPAdes need over 150 GB RAM. Also, server have plenty of physical memory for the genomes that output the SPAdes. Generally the Xeon processors are designed for servers, they have more cores, but at lower processing speed so they can execute many tasks but with higher processing time than i5 cores ,i7 cores.

3.3.2 Profiling Results

Generally, SPAdes used in our thesis as a verification platform in order to be sure that our profiling results are correct. The **Bulge Removing Algorithm** is part of the assembler,so we do the profiling measurements in real time in assembler. For the measurements we use the C++ library <chrono> as we do manual profiling. Firstly, we measure the **Total Time** that **Bulge Removing Algorithm** needs in total and afterward,we measure the **Process Time** that needs each of the four steps individually as it seems in Figure 3.1. The Data sets that referred in Table 3.2 used to run SPAdes and get the Profiling Results that presented in Table 3.3. Also, we change the source code of the SPAdes to run in one thread for the necessities of our measurements.In Table referred The **Process Time** of **Second** step of the **Bulge Removing Algorithm** because in second step consumed the majority of the total execution time. We placed the data sets in ascending order of the **Total Time**.

Data Set	Total Time(s)	Process Time(s)	Percentage(%)
DRR015506	**27.03**	**24.6**	**91.2**
DRR015482	**51.5**	**46.4**	**90.6**
DRR015499	**107.1**	**96.4**	**89.7**
DRR015480	**149.8**	**130.8**	**87.3**
DRR015512	**338**	**305.6**	**90.4**

TABLE 3.3: Profiling Results

The profiling results in our case give us that the second step of the **Bulge Removing Algorithm** is the one that consume the most of the time,in percentage range **80%-90%**. So we use the results of the Table 3.3 in the following sections of Chapter 3 as a guide in order to study the specific part of the code correctly and design it. Also, the percentage of time that the Process

Time takes over Total Time is significant, so we use the Amdahl's Law to determine the upper limit of speedup in our implementation.

We use the **Amdahl's law** formula in order to determine the upper speedup limit in case we use multiple compute units in an ideal system.We take advantage of the profiling results for the calculation of the theoretical speedup. The Amdahl's law function presented at next:

$$S_{latency}(s) = \frac{1}{1-p}$$

- $S_{latency}(s)$: is the theoretical speedup of the execution of the whole task.
- p : is the proportion of a system or program that can be made parallel.

In our case the proportion of program that can be made parallel is in range [0.8-0.9]. In Table 3.4 we present the upper limit of theoretical speedup for each data set.

Data Set	Data Set size(MB)	Proportion of Process Time(s)	Upper Limit Speedup
DRR015506	**5**	**0.912**	**11.36x**
DRR015482	**9**	**0.906**	**10.63x**
DRR015499	**17**	**0.897**	**9.7x**
DRR015480	**24**	**0.873**	**7.87**
DRR015512	**36**	**0.904**	**10.41**

TABLE 3.4: Amdahl's Law Theoretical Speedup

As we expect for each data set Amdahl's Law formula gives different results because they consume different proportion of execution time of the program. Furthermore, the range of theoretical speedup is in range [7.87x-11.36x], also in Chapter 5 we describe in details the final results from our implementation in FPGA.

3.4 Profiling Results Conclusion

Initially, the profiling result become our guide about the thesis. So, since the **Second Step(Calculate_Distance)** of the **Bulge Removing Algorithm** consume the 90% of the Total Time we start to study the second stage and try to make improvements on it.As we shown in Figure 3.1 the **Bulge Algorithm** executed step by step and uses one starting hub every time. Therefore,

it checks one by one the starting hubs if they are bulges.The first thing we change from the original Algorithm is to isolate the **Second Step** of the Algorithm and calculate the necessary distances for all the starting hubs. This change will give us the opportunity to handle the step that delay the Algorithm separately from the other steps, implement it in the FPGA as a single process and do the procedure for all the starting hub once. Thus, we make a new diagram to be more understandable the new struct of the Algorithm which seems in Figure 3.2.

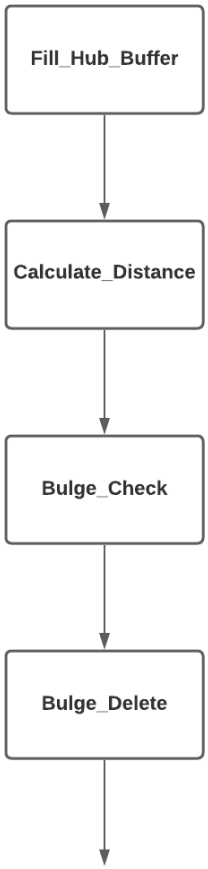

FIGURE 3.2: Modified Bulge Removing Algorithm.

In the modified Bulge Algorithm we take all the starting hubs that we need to process and pass them as input to the second step. After the end of the second step the results used in the third step to determine if the path is bulge and in the last step the algorithm delete the paths that are bulges. Thus, we d not change the substance of the Algorithm but we modified the way that processing the data.

In Chapter 4 we describe the second step of the **Bulge Removing Algorithm** and give details about our thoughts to improve the code and implement it into the FPGA.

Chapter 4

System Implementation

In Chapter 4 we describe in details the steps that followed in order to reach in the final hardware implementation. Firstly, the original code of the second step of **Bulge Removing** algorithm of SPAdes.13.0 modified in order to build it in software implementation of Vitis. Vitis automatically create the hardware implementation in FPGA at next but the results was poor so we decide to make manual improvements in the current hardware implementation. The manual improvements did not offer remarkable results, so we implement a new configuration which is a tree like implementation and constitutes the final configuration. At last, added 4 more compute units in the final implementation in order to speedup its execution time.We use the tool of Xilinx Vitis Unified Software Platform for the FPGA implementation. Also, in Chapter 4 we refer the time latency issues that referred by Vivado HLS log files.

4.1 Software Implementation

The main goal of the algorithm is to calculate the distance between the starting hub and the neighbours until we find the ending hub, in order to achieve that we need to calculate the distance between all of the hubs between the starting and the ending hub.In the begging, the algorithm starts with a starting hub which stored in a **Priority Queue**, the priorities that checked before the insertion in the queue are the four that follows:

1. The **length** of the h-path is the first criterion.
2. The second is the **ID number** of the **current vertex**
3. The **ID number** of the **previous vertex**
4. The number of the **edges between** the current and the previous vertex.

For each starting vertex the priorities are stored in a **Struct** and pass the struct every time we want to check them before insert it in the Queue.The struct contains four integers values ,thus for every node we need 128 bits. At next, the vertex popped from the Queue and check if we have processed the same vertex before, also check some other thresholds about the length of the h-path . If the vertex pass through all the statements we save it in an output vector and process the vertex to find its neighbours,the neighbours stored in the priority queue in order of the values of their priorities. After the first iteration that we process the starting hub then the steps of the algorithm are the same as before.The difference is that we pop the vertex with the highest priority from the **Priority Queue**, processing the hub with the highest priority to find its neighbours and stored them in the priority queue too. Thus, for the vertex with the highest priority we find its neighbours and the distance from them, keep going the process until the **Priority Queue** is empty.The maximum number of neighbours that a hub can have are 4 and the minimum is 0. Also,in our design the maximum number of elements that can stored in the **Priority Queue** are 7000.

We do the procedure that we describe above for all of the starting hubs separately. Therefore, every starting hub do not depends from the other starting hubs or their results. This notice guarantee us that we can make in parallel the processing of every starting hub . We quote pseudo code that describes the procedure of the distance calculation.

Algorithm 1 Main of Second step of the Bulge Removing Algorithm

```
1:  start_counter ← 0
2:  while start_counter < starting_hubs.size() do
3:      queue[7000] ← starting_hubs          ▷ insert a starting hub in the rear
4:      vertex_out ← 0
5:      distance_out ← 0
6:      while queue.size() ≠ 0 do
7:          next ← queue.top()                          ▷ pop the top element
8:          vertex ← next.vertex
9:          distance ← next.distance
10:         if DistanceCounted(vertex, vertex_out) == 0 then
11:             continue
12:         vertex_out.push_back(vertex, vertex_out)      ▷ store output data
13:         distance_out.push_back(distance)
14:         if CheckProcessVertex(distance) == 1 then
15:             continue
16:         AddNeighboursToQueue(distance, vertex, vertex_out)      ▷ push
    neighbours in priority queue
17:     start_counter + +
```

As we can see in line 3 of the Algorithm 1 initialized a queue with 7000 elements, we push elements from the rear of the Queue and pop them from the front. Also we initialize the output vectors before starting processing a new starting hub. The priority queue in software works as a binary tree as it is the most efficient way to implement it. We need O(logN) time for pushing elements and the same time complexity for popping elements,so the overall time complexity of the algorithm is O(logN). The binary tree have logN levels where n are the elements of that are stored in the structure. The space complexity of the binary tree is O(n) because we store N elements in the structure.

Follows the pseudo code of the function that called in line 10 of the Algorithm 1.

Algorithm 2 DistanceCounted(vertex,vertex_out)

```
1:  counter ← 0
2:  i ← 0
3:  while counter < vertex_out.size() do
4:      if vertex_out[i] == vertex then
5:          return 1
6:      i + +
7:      counter + +
```

The Algorithm 2 is a simple serial search in the output vector to check if the current vertex have processed in the past. In the case of match the function

return 1.

In Algorithm 3 referred the pseudo code of the function AddNeighboursToQueue. It is the function with the highest impact in the whole algorithm.

Algorithm 3 AddNeighboursToQueue(distance,vertex,vertex_out)

```
1: while hasNeighbour do
2:     if DistanceCounted(neighbour_vertex, vertex_out) == 0 then
3:         new_dist ← curr_dist + neighbour_dist
4:         if new_dist < 3000 then
5:             push(new_dist, neighbour_vertex, vertex, neighbour_edge)
```

In the line 1 of the Algorithm 3 we check if there is another neighbour for the current vertex. In the line 2 check the neighbour vertex if we have process it before and afterwards we push it in the priority queue. The neighbours of each vertex are stored in the a structure.

From the study of the code we understand that the processing with the highest impact in the execution time is the pushing and the popping of elements in the priority queue. Also the serial searching in the output vector in the Algorithm 2 is a part of the algorithm that we can improve.

4.2 Automatic Hardware Implementation

The software implementation that referred above can become hardware implementation without any effort because Vitis assumes the conversation of the implementation.In the current section presented the hardware implementation which produced automatic by Vitis.In hardware implementation of the algorithm we have two types of configuration which are the host and the kernel configuration. We describe in following subsections the host and the kernel configurations for the first implementation in the FPGA. Also,we present in Figure 4.1 the Abstract Block Diagram of our implementation.

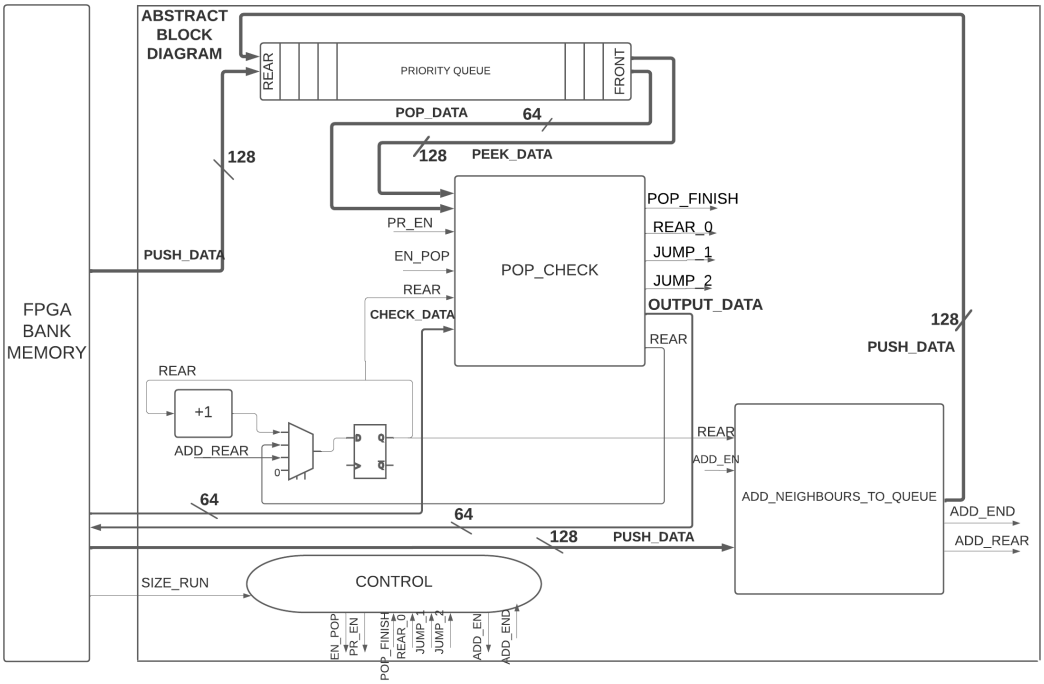

FIGURE 4.1: Abstract Block Diagram.

4.2.1 Host Configuration

The host code prepare the data for the kernel, the data pass in the host as arguments . The main goal of the host is to create the connection between the FPGA and the CPU of the server, also, in the host we read from files the necessary input data for the processing and store them into vectors. The vectors with the input data pass as arguments in the kernel and stored into the HBMs of FPGA. The files created during the execution of the SPAdes, so we have all the data that we need to run the algorithm in the FPGA. Further,during the execution of the SPAdes we keep in a file the output results that we should get from the execution of the algorithm in the FPGA. In the end we compare one by one the output results of SPAdes with the output results of the FPGA with a script in C language to be sure that we have correct results. In the Abstract Block Diagram in Figure 4.1 the FPGA BANK MEMORY is the HBMs where we store the input data.

4.2.2 Kernel Configuration

The Abstract Block Diagram of Figure 4.1 constitutes an overview of the kernel, as we can see the kernel separated in 2 stages: **POP_CHECK** and **ADD_NEIGHBOURS_TO_QUEUE**.The stages are placed in the block diagram in order of execution,also we have the main control which determines

the values of the signals for the synchronization of the system.The value of the register in the middle of the block diagram constitutes the pointer in the rear of the priority queue and we use the pointer to be aware about the available capacity of the priority queue. In each element of the priority queue stored 4 integers values that constitutes the identity of each vertex, so we need 128 bits for each vertex.

Initially,pushed the identity values of the starting hub in the priority queue, the rear pointer increased by one and the main control enable the first stage. The identity values of the starting hub popped from the queue and become a comparison between the identity values and thresholds in **POP_CHECK** stage. If the checks are valid then the vertex and the length of its path stored in the HBM, also the main control interupted in order to enable the second stage. In the second stage we are searching for the neighbours of the popped vertex, each neighbour pushed in the priority queue. After the end of the second stage the main control enable the first stage again and searching the element of the priority queue with the highest priority and popping it. At next the main control enables the second stage and searching for the neighbours of the popped element. The first and the second stage iterated until the priority queue is empty. Then pushed a new starting hub in the priority queue and the process starts again. Every time we push data in the priority queue increase the rear pointer by 1, on the other hand when we pop data from the priority queue then decrease the rear pointer by 1, so we are aware about the available capacity of the queue.

In conclusion, the results of automatic hardware implementation was poor in order to code optimization and execution time.There are two important reasons which are responsible about the poor results and referred at next:

1. Algorithm search the output matrix in serial in order to check if the current vertex have processed. This algorithm called two times,so it slows down the execution of the whole algorithm.

2. Also,numerous accesses in priority queue is an important issue that increase the execution time of algorithm.

Thus, we decide to change manually the code of kernel in order to succeed better performance in the current implementation. We focus in the improvement of Algorithm and in the way that pushed and popped data from the priority queue.

4.3 Manual Improvements on Automatic Hardware Implementation

We relied in the first implementation and we tried to do improvements in the implementation to success speedup. Mainly, we focus on the improvement of the kernel configuration.

4.3.1 Kernel Reconfiguration

An imperative change that we do is a **Look-up-table** that we create in order to find if a vertex have processed before. Until now for this task we search the output data in serial to determine whether a vertex is processed or not. Furthermore, we decide that we should change the process of pushing and popping data from the **priority queue**. The main goal was to make it fully pipeline, so we to make the pop and push process executed in one clock cycle each. In the following subsection referred details about the way that data pushing and popping from the priority queue, changes that we tried to do and a block diagram for the priority queue. Also, referred the reasons that this implementation is not suitable for our case. Despite the fact that we discard the first implementation the **Look-up-table** is an insertion in our code which we do not change until the final implementation.

Look-up-table

Firstly, we have some thoughts about the way that we would make the **DistanceCounted** module executed in one clock cycle. We decide to totally remove the specific module from the implementation and replace it with a **Look-up-table**. The **Look-up-table** created in the host code during the reading of the input files, every vertex have a unique id number that separates it from the other vertexes. Thus, we search for the maximum id number and we make an array in which the index is the key and the value of the key is the state of the vertex. In case the value is 0 the vertex has not processed but in case the value is 1 means that the vertex has processed. In begin the Look-up-table initialized in 0, we change the value of the key in 1 for each vertex that stored in output data, thus we be aware for the vertexes that we have processed.

Finally, the implementation of **Look-up-table** is a very important manual change because it needs just one clock cycle to check if the current vertex have processed and replace a function that needs several execution time.

Priority Queue

The **priority queue** is an issue that occupied us for long time, so that we implement it proper to success speedup. We wanted to make the priority queue fully pipeline, the look-up-table that we implement in the previous step is useful for the priority queue because the push and the pop depend on it. The push process after the removal of the **DistanceCounted** module executed in one clock cycle. However,in the **POP_CHECK** stage traversed all the elements of the priority queue one by one because we search for the element with the highest priority. To overcome this problem we tried to fully partition the priority queue and transform it into registers. In Figure 4.2 presented the block diagram of the priority queue that we try to implement, as it seems we have the main priority queue and an alternative queue, the two queues are partitioned in registers. Also, the **dec** module determines the path that will follow the data, we need the second queue to store the data that have lower priority than the corresponding in the first queue and in the next clock cycle compare it with the next element of the first priority queue. Thus, we push data in parallel in one clock cycle. Also, in case data should pushed between two registers which have already data then the registers that are from the rear side shifted by one register to make a hole for the new data. For the pop process the data come out from the front register and the rest data of the first priority queue shifted to the front register to cover the gap that made.

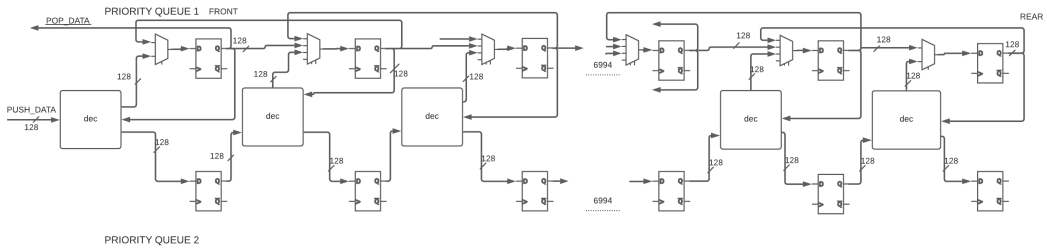

FIGURE 4.2: Pipeline Priority Queue Block Diagram.

In conclusion Vitis do not have the capability to build the above configuration for two reasons.

1. Firstly ,we need a priority queue with 7000 elements to be fully partitioned which is impossible for the tool. The upper limit of Vitis for fully partitioning is vectors with 1024 elements, so we get a warning message from the tool that the implementation can not build.
2. The second reason is that we have 4 id values for each vertex. That means we do 4 compares to decide which vertex have higher priority. Thus, the flowchart of the implementation have high complexity and the tool can not build it

In the next section analyzed the second configuration that we make and is the final configuration.

4.4 Final Hardware Implementation

The second hardware implementation is completely different from the previous configuration. The previous implementation could not build on the FPGA, the problem was that the Vitis have not the ability to fully partitioning the priority queue of 7000 elements. Also, each element has 4 values, so the flowchart is too complicated and the tool can not make the implementation pipeline. Thus, we concluded that the main issue is to implement the priority queue with better way, we make changes in push and pop process of the priority queue. In Figure 4.3 presented the Abstract Block Diagram of the kernel, in the following subsections we describe in details the host and kernel configuration.

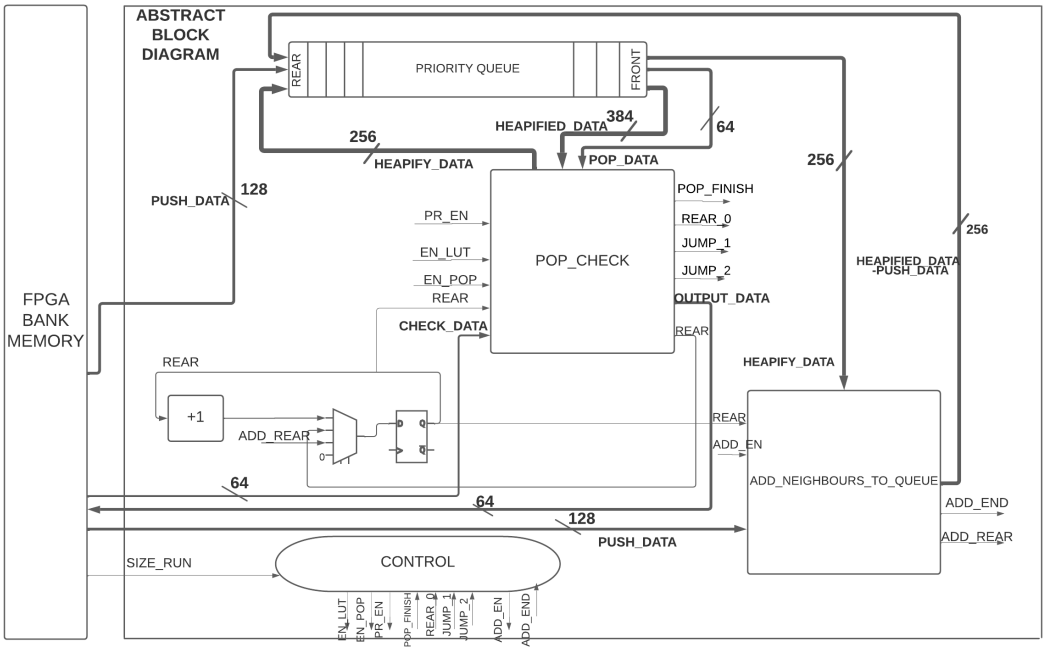

FIGURE 4.3: Abstract Block Diagram.

4.4.1 Host Configuration

The host code of the current configuration is the same as the first configuration. In subsection 4.2.1 described in details the operation of the host code. The host code remains the same because the input data needs the same processing to pass as arguments in the FPGA. The kernel code is completely different for the pop and push process, in the next subsection described in details the kernel.

4.4.2 Kernel Configuration

In the current configuration as it seems in Figure 4.3 the block diagram is the same concerning the stages but it has changed in order to pushed and popped data from the priority queue as in a binary tree. Also, from the first implementation we keep the **Look-up-table** for the check of the processed vertexes but we change radically the push and pop process.

Our first concern is the different implementation of the priority queue,so we decided to implement it as a binary tree. In the binary tree the data pushed in the first empty leaf, then compared the new data with the data of the parent node. In case, the data of parent node have lower priority then the new data swapped with the parent data, so the new data stored in the parent node and the data of the parent node stored in the leaf node. This process

continues until the new data compared with data that have higher priority. The binary tree have the data with the higher priority in the root node, so it pops data from the root node. Afterwards, in the root node stored the data from the right most leaf, then start to compare the new data of the root node with the data of the left neighbour and the right neighbour, in case root node have lower priority of their neighbour then it swapped with the neighbour with the higher priority. This process continues until the new data compared with neighbours that have lower priority. With this way the priority queue remains ordered. The rear pointer remains in the implementation, give us the first available leaf of the priority queue to push data, it increases by 1 in case of pushing data and decreased by 1 in case of popping data from the priority queue.We select to reconfigure the priority queue into binary tree because it needs log(N) levels to store N data, so the complexity of pushing and popping data reduced to O(log(N)) in contrast the first implementation have O(N) complexity. In the following subsections explained the inner process of each stage.

POP_CHECK

The **POP_CHECK** is the first stage of the process,the inner content of the stage presented in Figure 4.4. The main goal of the stage is to pop the root node from the priority queue and sort the priority queue again. The popped data send to the HBM for storing in case that pass through all of the checks. The Figure 4.4 present details about the data path and the modules that are in the stage.

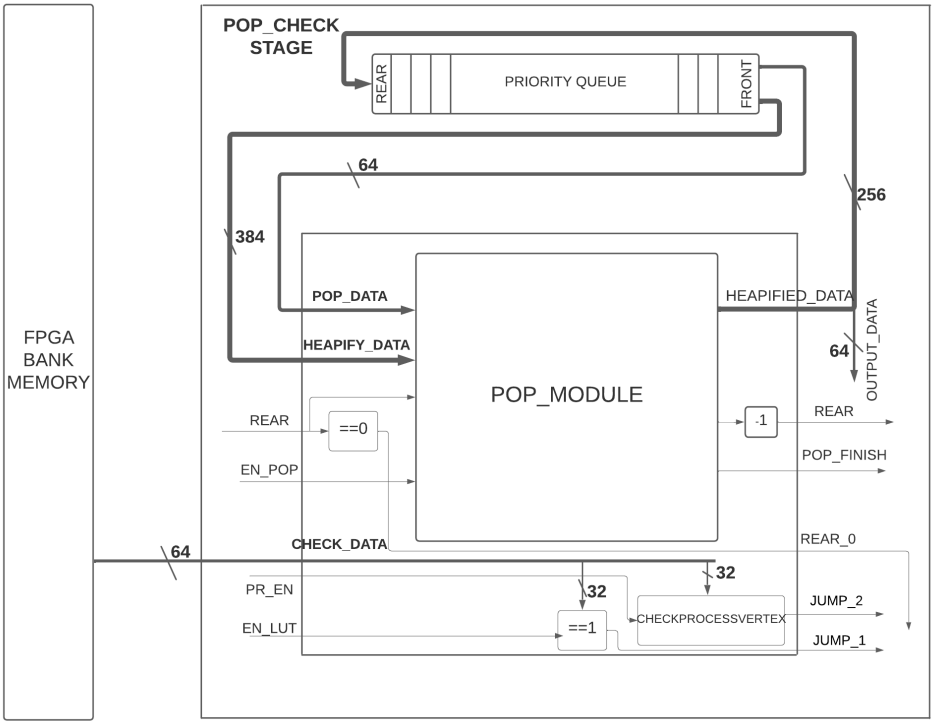

FIGURE 4.4: Pop Check Stage Diagram.

Firstly, we check if the priority queue is empty, the priority is empty when the rear pointer is equal to 0. When we make sure that the priority queue is not empty then enabled the **POP** module which is responsible to pop the root node from the priority queue and sorting the priority queue.**POP_DATA** are 64 bits because from each element of the priority queue stored 2 unsigned integer in the HBM. The **HEAPIFY_DATA** are 384 bits because we need to compare 3 elements to decide whether they need to swap, the elements that compared are the parent node, the left and the right child nodes. After the end of **POP** module checked if they have processed again or if the length of the vertex is greater than a threshold in **CheckProcessVertex**. In case the checks are valid then the output data pushed in the HBMs, more details about the **POP** module referred at next. In addition, we present in Figure 4.5 the inner of the **POP** module.

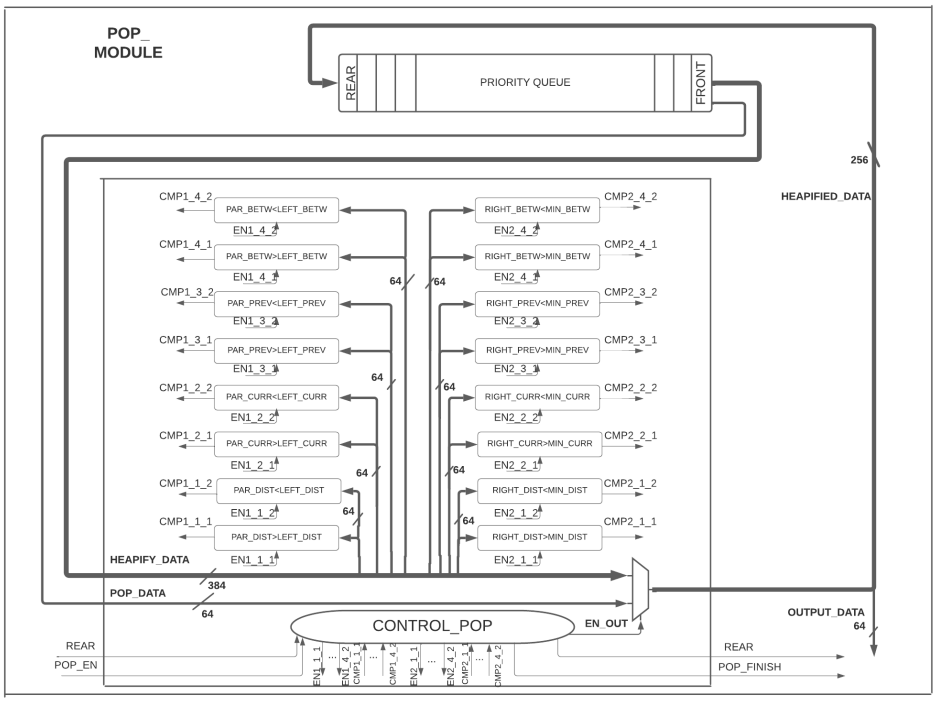

FIGURE 4.5: Pop module Diagram.

The **POP** module is responsible to sort the priority queue, it has a control module which synchronize the process. In the beginning, the **POP_DATA** stored in a register until the priority queue is sorted. The **POP_DATA** are the data of the root node. For the sorting of the priority queue we transfer the data of the right most full element in the front element of the priority queue, then we start the comparison. Firstly,we compare the new data of the root node with the right and the left child nodes, in case the data of the root have lower priority than the children then they swapped with the data of the child with the highest priority between them. This process continues until the data that started from the root node compared with children that both have lower priority.In **HEAPIFIED_DATA** contained the data that will swapped in the priority queue, so we need 256 bits to transfer the data.The control module let the **OUTPUT_DATA** to pass as output when the priority queue is sorted.

Add_Neighbours_To_Queue

The last stage of the process is the **Add_Neighbours_To_Queue** in which we search for the neighbours of the vertex with the highest priority. The identity values for the neighbours are saved in the HBM, thus we load them and check them. The **PUSH** module is responsible for the push of the new data and the ordering of the priority queue. The control of the stage synchronize

the process, at next we explain the inner process of the **Add_Neighbours_To_Queue** stage. Also, in Figure 4.6 presented the block diagram of the current module.

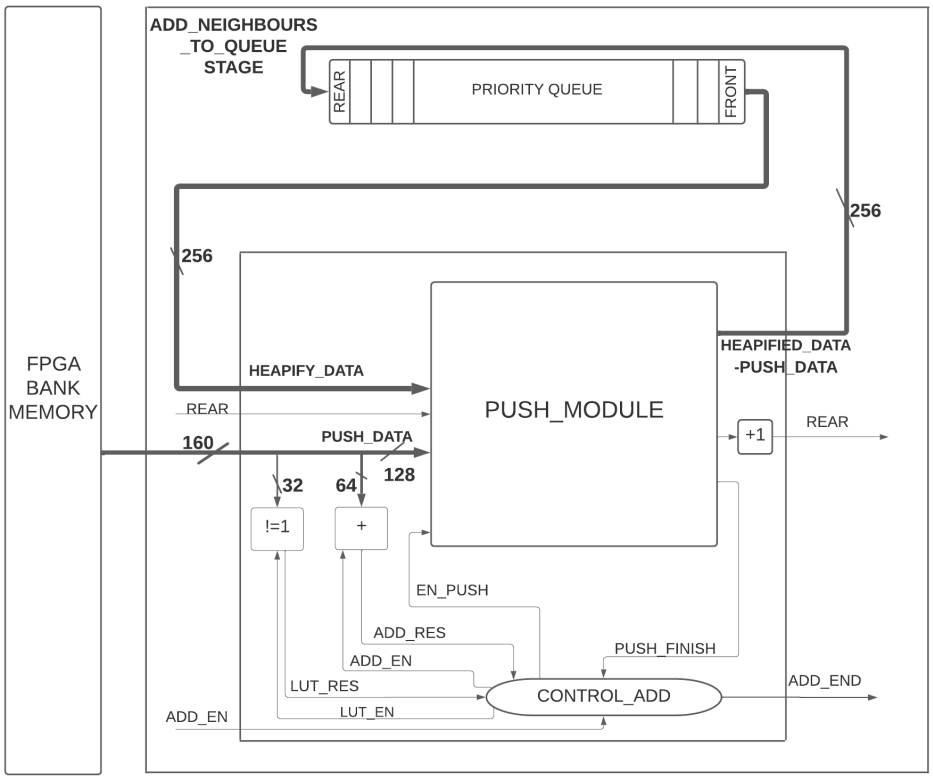

FIGURE 4.6: Add Neighbours To Queue Stage Diagram.

Initially, in the current stage we check if the neighbour have processed before again, we do not want to process the same vertex twice so we reject it in case of match. The Look-up-table is responsible to keep information about the vertexes that have processed, so we check the corresponding value of the Look-up-table for the vertex that we want to process. Furthermore, adding the current distance with the distance of the neighbour and check the result do not be greater than a threshold. At next if all the checks are valid the control module enables the **PUSH** module which is responsible for pushing the new data in the priority queue and ordering them. Also we should iterate the process for all the neighbours, the maximum number of neighbours are four, the control synchronize the iterations. In the end of the process the rear pointer increased by 1 as we pushing data in the priority queue. In Figure 4.7 presented the inner content of the **PUSH** module.

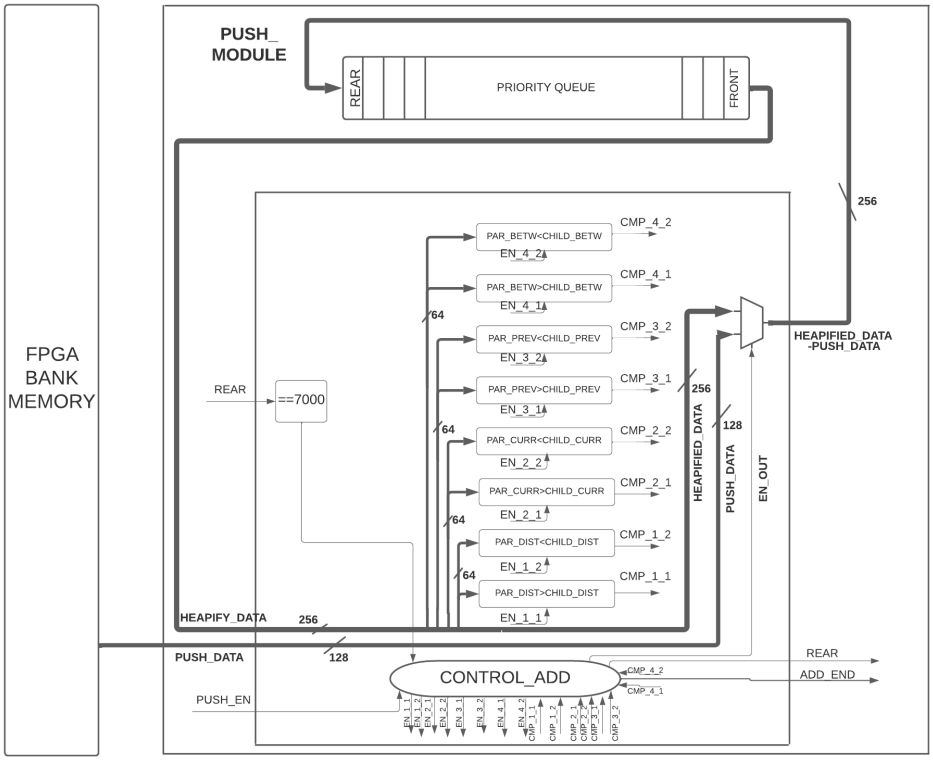

FIGURE 4.7: Push module Diagram.

At start we check the value of the rear pointer, in case the value is greater equal to 7000 then we can not push more data in the priority queue, it is overflowed.In other case, the **PUSH_DATA** pushed in the first empty leaf node of the priority queue.The **HEAPIFY_DATA** that compared are the last inserted data in the leaf node with the parent of the leaf node. If the new inserted data have higher priority then swapped with the parent element data, this continues until the new inserted data compared with a parent node with higher priority.Thus, the control module synchronize the process, it determines the data that will send to the priority queue from the multiplexer.

In conclusion,in the current implementation the priority queue operate like a binary heap,so time complexity of implementation reduced in O(logN) in contrast the previous implementation have O(N) time complexity. We relied in the current implementation in order to implement more compute units to achieve speedup.

4.5 Host Reconfiguration-Parallelization

Firstly, we have to decide the number of compute units that we are available to implement in our system. In total there are 32 HBMs in our FPGA each one has 256MB storage space, so the FPGA have 8GB storage space in total. In the next table presented information about the way that separated the HBMs in the implementation.

HBMs for 1 Compute Unit	HBMs for 5 Compute Unit	Total HBMs
6	**30**	**32**

TABLE 4.1: HBMs Allocation.

Each compute unit allocates 6 HBMs, so we can implement 5 compute units. . The differences in the new host code in relationship with the previous host code are that in the new host code created and connected 5 compute units with the CPU.We modified the reading of the data sets in order to succeed a balanced parallelism, the input data separated into 5 vectors, each one pass as argument in one compute unit. For the correct separation of the input data we wright in the files the value -1 after the last data that a starting hub needs to execute,so -1 is a flag to understand when ends the necessary data that needs each starting hub to processed.

4.6 Time Latency Issues

It is useful to have knowledge about the latency in clock cycles in our implementation and for each module separately because we help us to understand in which module we should focus in order to optimize it and get better performance. After the end of hardware implementation Vitis HLS and Vivado IDE produce one log files each which contain information about the latency of configuration in clock cycles and the frequency of clock of kernel. Also, the log file of Vivado IDE [26] present information about space utilization of the implementation in the FPGA but we will refer to them in details in Chapter 5. There are three meanings that we have to explain in order to understand the results of the log files for the time latency, these meanings referred at next:

- **Latency**: The number of clock cycles required for a complete run of a function or loop.

- **Iteration Latency**: The number of clock cycles required for running a single iteration of a function or loop.
- **Iteration/Initiation Interval (II)**: The number of clock cycles required before a module can accept new input or a loop can initiate a new iteration.
- **Pipelined**: Tag that mentions if a module or loop is implemented using a pipelined architecture.

After the end of the hardware implementation we open the log files in order to see the values for the above meanings in our implementation. At first, the maximum value of clock frequency for the accelerator is 300 MHz, for our implementation the clock frequency initialized in 265 MHz. The system have losses in clock frequency because we do not work in ideal conditions, also there are modules in the implementation which needs bigger clock period than the default.

As we refer in the above section the implementation separated in 3 stages, each stage depends from the latency of the next stage because they executed in serial.So, we present the latency of the stages in backward order.

- **ADD_NEIGHBOURS_TO_QUEUE** is the second stage in order, the loop of the stage executed maximum 4 times with 1332 iteration latency, so the second stage has 5328 latency in total. PUSH module needs 1041 clock cycles to execute over 1332 of the total stage. Thus, the PUSH module consumes around 80% of the total clock cycles of second stage.
- **POP_CHECK** is the first stage of the implementation, it is not possible to define the total number of iterations of the loop because we do not know the number of neighbours that will popped from the priority queue. However, the tool can define the iteration latency of the loop which is 5547 clock cycles, as we can see the second stage consumes 5328 clock cycles over 5547 clock cycles of the second stage which is around 96% of the total cycles.The POP module consume small amount of clock cycles,just 78 clock cycles.

The total latency of our system depends from the number of starting hubs that passed as input in the FPGA because we need 5548 clock cycles in order to process each starting hub.

The modules PUSH and POP referred as pipeline modules from the tool but in fact they are not. The iteration latency of PUSH module loop is equal to 76

and the interval of the loop is 74, so the loop accept new data after 74 clock cycles and the loop ends after 76 cycles. On the other hand the POP module have iteration latency 6 and interval 6, so the loop start to process new data after the end of the previous loop.Thus we understand that anyone of the stages have pipeline module.

In Chapter 5 referred the results from our implementation, the amount of resources that each implementation consumes, the reasons for the implement of 5 compute units .Also, we search for other platforms which will have better results from the Alveo U50 Platform, we refer the specifications of each platform and compare the platforms between them.

Chapter 5

Results

Chapter 5 is useful for the understanding of quality of our implementation because referred the execution time results of our final implementation. For the first implementation we do not have any execution time results because the tool could not build it,so we refer the results for our final configuration. In the current Chapter we compare the results of the final implementation with one compute unit with the results of the final configuration with five compute units, also we describe the quality of our parallelization.Further, we compare the execution time of SPAdes with the corresponding execution time of the final implementation with 5 compute units.Also ,referred the FPGA resources that consumed by the implementations. We notice that the FPGA with 5 compute units needs similar execution time as the SPAdes to execute the same dataset.We can not implement more compute units to succeed better performance because of the lack of storage space in the FPGA. For this reason we search for other FPGAs which will have better performance, so we make a projection in other FPGAs compared to ALVEO U50 specifications.

We decide to include in our projection three platforms which belongs to Xilinx because they are compatible with Vitis, so in future do not need to change the code of the implementation. **ALVEO U250**[27],**ALVEO U280**[28] and **Virtex UltraScale+ HBMVU57P**[29] are the platforms that we select to compare with our platform. **ALVEO U250** and **ALVEO U280** accelerator cards included in the same family with **ALVEO U50** but they are improved models than our model platform, Virtex is a different family from Alveo. The specifications of each platform referred in following section.

5.1 Quality of Parallelism

We are obliged to check the quality of our parallelism, so in Table 5.1 shown the comparison between the serial and the parallel implementation. Execution time is time needed by FPGA to execute the implemented algorithm for all the starting hubs.In addition, we use the library <chrono> of C++ for the calculation of the execution time, we add the necessary code in host in order to calculate the execution time of FPGA.

Data sets	Size of Data set(MB)	Execution Time of 1 Compute Unit(sec)	Execution Time of 5 Compute Units(sec)	Speedup
DRR015506	**5**	**67.2**	**15.1**	**4.45x**
DRR015482	**9**	**112.6**	**25.3**	**4.46x**
DRR015499	**17**	**257.8**	**55.2**	**4.43x**
DRR015480	**24**	**330.5**	**74.6**	**4.43x**
DRR015512	**36**	**779.7**	**176.4**	**4.42x**

TABLE 5.1: Execution Time for 1-5 Compute Units - Speedup

From the above Table we notice that we succeed a parallelization of the order 4.4x-4.45x which we expect it for different reasons.

1. Initially, in the input files we have add the flag -1 at the end of the necessary data for each starting hub, so it is easy to separate them.
2. Each starting hub does not need the same amount of process, so we can not separate the data in order of starting hubs. We have in an input file the neighbours of each starting hub,so we count the number of neighbours in total, then divide the total number of neighbour by five. Thus, we create a threshold which used in order to separate the data in balance and share the amount of process equally in the five compute units.

The upper limit of speedup that we expect is 5x in an ideal system but in real world we do not work on ideal systems, so the speedup of 4.4-4.45x constitutes a significant quality of parallelism. At next section, presented the allocation of FPGA available resources from the configurations.

5.2 Resources Consumption

In the current section we present the resources that consumed by our configuration for 1 and for 5 compute units from the FPGA, the information about the resource consumption given from the Vivado log file. Vivado log file produces from Vivado IDE after the building of our configuration in FPGA. Table 5.2 contains the resources that utilized by the configuration with 1 compute unit from the available resources of FPGA. Table 5.2 is useful because we can understand the needs of our implementation in numbers.

Board Specifications	Used	Available	Utilization(%)
LUTs	**122.595**	**870.016**	**14.09**
Registers	**179.772**	**1.740.032**	**10.33**
DSP Slices	**118**	**5940**	**1.99**
Block Ram	**232**	**1344**	**17.3**
HBM Memory Capacity	**1.5GB**	**8GB**	**18.75**
Total Power	**17.6W**	**75W**	**23.4**

TABLE 5.2: Source Consumption for 1 Compute Unit.

Generally, most of the implementations suffers from lack of **LUTs**, **Block Ram**, **Storage Space** and in **Routing** of the implementation. In our case the **LUTs** and the **Block Ram** needs **14.1%** and **17.3%** of the available resources respectively, so we have the ease to implement more compute units without concern about the lack of those resources. The **Routing** of the configuration is an unstable factor,we can not predict when the available cables of the platform will end . Also, the **Storage space** is utilized around **19%** for 1 compute unit, thus we understand that we can implement maximum 4 more compute units, in total 5 compute units. At next we present the resources consumption for 5 compute units. The FPGA works on 23.4% concerning the Total Power.

The Table 5.3 have exactly the same structure with the Table 5.2 and it presents the resources that the configuration consume from the platform for 5 compute units.

Board Specifications	Used	Available	Utilization(%)
LUTs	**222.210**	**870.016**	**25.54**
Registers	**352.061**	**1.740.032**	**20.23**
DSP Slices	**574**	**5940**	**9.66**
Block Ram	**410**	**1344**	**30.54**
HBM Memory Capacity	**7.7GB**	**8GB**	**96.25**
Total Power	**41.2 W**	**75W**	**55**

TABLE 5.3: Source Consumption for 5 Compute Units.

About the **Storage Space** the configuration have allocate the **96.25%** of the available HBMs,so we reach the upper limit for our implementation. Each compute unit needs 6 HBMs to store input data of each compute unit, the HBMs not connected between them and in total the platform have 32 available HBMs, thus we allocate 30 HBMs over 32 HBMs in total. We should emphasize that the HBMs do not fill completely with data for all of the data sets.However, The largest data set of Table 3.1 fills 3 of 6 HBMs completely with data and the rest of HBMs around 80%,so we understand that we can not run larger data sets with this implementation in specific FPGA.

In Table 5.4 presented the percentage of resources utilization for each extra compute units. The percentage that consume each extra compute units resulting from the difference between percentage utilization of 5 compute units with percentage utilization of 1 compute unit divided by 4.

Board Specifications	Utilization of 1 Compute Unit(%)	Utilization of 5 Compute Units(%)	Utilization of extra Compute Unit
LUTs	**14.1**	**25.5**	**2.85**
Registers	**10.33**	**20.23**	**2**
DSP Slices	**1.99**	**9.66**	**2**
Block Ram	**17.3**	**30.5**	**3.3**
HBM Memory Capacity	**18.75**	**96.25**	**18.75**

TABLE 5.4: Utilization of each extra Compute Unit

From Table 5.2 and Table 5.3 we conclude that the main issue for our results is the lack of **Storage space**.Also, the FPGA have available plenty of **LUTs**, **Registers**, **DSP slices** and **Block Ram**. The FPGA works on 55% concerning the Total Power for 5 compute units ,so we notice that the total power doubled in order to 1 compute unit.

5.3 Final Implementation Results

In the current section we make two major comparisons for our thesis. The first comparison concerns the execution time of our impementation with 5 compute units versus the execution time of SPAdes with 1 thread in use. In the second comparison we compare the execution time of our configuration again against the execution time of SPAdes but this time with 16 threads in use. We calculate as execution time of SPAdes the time that SPAdes needs to execute the part of Bulge Removing algortihm which we implement in the FPGA. We have separate the comparisons in two subsections in order to be undrestood the difference between them.

5.3.1 Execution Time of FPGA with 5 Compute Units - SPAdes with 1 Thread without I/O Operation

We decide to present the execution time of SPAdes for 1 thread because we wanted to include every calculation that we made in our thesis. For the calculation of execution time in SPAdes we use the library of C++ <chrono>, we place the necessary code in SPAdes for the specific calculation.

Data sets	Size of Data set(MB)	Execution Time of 5 Compute Unit(sec)	Execution Time of SPAdes with 1 Thread(sec)	Speedup
DRR015506	**5**	**15.1**	**24.6**	**1.63x**
DRR015482	**9**	**25.3**	**41.2**	**1.63x**
DRR015499	**17**	**58.2**	**96.4**	**1.66x**
DRR015480	**24**	**74.6**	**130.8**	**1.75x**
DRR015512	**36**	**176.4**	**305.6**	**1.74x**

TABLE 5.5: Execution time of SPAdes with 1 thread versus 5 Compute Units implementation in FPGA - Speedup

As it seems in the Table above FPGA succeeds speedup in range 1.63x-1.75x over the non-optimized SPAdes. Furthermore, we notice that as the data sets get larger in size then the speedup increased. Also, the results would be better in case we implement our configuration in a FPGA with more storage space because we will implement more compute units. At next subsection shown the results for fully optimized SPAdes against the FPGA results.

5.3.2 Execution Time of SPAdes with 16 Threads - FPGA with 5 Compute Units without I/O Operation

We run SPAdes with fully optimized options, it uses 16 threads to run the algorithm where we implement in FPGA. We use the library of C++ named <chrono> as in profiling in order to calculate the execution time of Bulge Removing Algorithm in SPAdes with 16 threads.

Data sets	Size of Data set(MB)	Execution Time of 5 Compute Unit(sec)	Execution Time of SPAdes with 16 Threads(sec)	Speedup
DRR015506	**5**	**15.1**	**20.4**	**1.35x**
DRR015482	**9**	**25.3**	**28.8**	**1.14x**
DRR015499	**17**	**43.6**	**55.8**	**0.79x**
DRR015480	**24**	**74.6**	**64.3**	**0.86x**
DRR015512	**36**	**176.4**	**135.8**	**0.77x**

TABLE 5.6: Execution time of SPAdes with 16 threads versus 5 Compute Units implementation in FPGA- Speedup

From Table 5.6 we notice that our implementation succeed a small speedup in the first and second data set in execution time 1.35x and 1.14x respectively. Also, we notice that in larger data sets our implementation have lower performance than software implementation.We expect it because **ALVEO U50** does not have the necessary amount of storage space for the creation of more compute units, so we can not succeed better performance. Further, we can not exploit completely the possibilities of the tool, for example we can not make the priority queue fully pipeline. Priority queue is the main issue of our implementation because we have many accesses in it and specific the process of pushing data in it.

5.4 Further Technology Abilities

The main issue in our configuration is the lack of storage space because our implementation utilize completely the available storage space.We focus on Xilinx platforms specific in ALVEO family accelerator cards and Virtex family accelerator cards because they have the appropriate specifications for our necessities. We search in other companies too but at the end we choose to refer only Xilinx platforms in my thesis. For the correct comparison between the platforms we need to calculate the throughput of our implementation, the function for the calculation of theoretical throughput referred at next:

$$Throughput = Input\_data \cdot Fmax$$

In our case Fmax=262 MHz and from Figure 4.1 we calculate that Input_data=320 bits =40 bytes,so

$$Throughput = 40bytes \cdot 262MHz = 11GB/s$$

However, the real throughput of our system calculated by the next formula:

$$Throughput = Total\_input\_data/Total\_kernel\_latency$$

We calculate the practical throughput of our system for the largest available data set at next:

$$Throughput = 7.5GB/176.4sec = 42.5MB/s$$

We notice that the theoretical throughput is much bigger than the practical throughput because the priority queue reads data from HBM at slow pace.

5.4.1 Comparison of Specification of Platforms

In the current subsection we gather information about the new platforms that we will compare with ours. We found 3 remarkable accelerator cards, so we gather their specifications in three different Tables against the specifications of Alveo U50.

Alveo U50 - Alveo U250

Board Specifications	Alveo U50 Accelerator Card	Alveo U250 Accelerator Card
LUTs	**872K**	**1.728K**
Registers	**1.743K**	**3.456K**
DSP Slices	**5.952**	**12.288**
Block Ram	**1.755**	**2.016**
HBM Memory Capacity	**8GB**	**-**
HBM Total Bandwidth	**316 GB/s**	**-**
DDR Memory capacity	**-**	**64GB**
DDR Total Bandwidth	**-**	**77GB/s**
PCIe	**Gen3x 16, 2 x Gen4x 8, CCIX**	**Gen3 x16**
Maximum Total Power	**75W**	**225W**

TABLE 5.7: Alveo U50-U250 Specifications.

The main difference between them is the storage space. On the one hand, Alveo U50 have 8 GB storage space, on the other hand Alveo U250 have 64 GB storage space. Thus, with 8 times more storage space we can implement **40 compute units** and run larger data sets than Alveo U50. Also, we notice that DDR Total bandwidth of Alveo U250 is 77 GB/s which is enough to cover the needs of our implementation.

At next follows a Table that contains resources consumption for 40 compute units.

Board Specifications	Used	Available	Utilization(%)
LUTs	**1.093.815**	**1.728.000**	**63.3**
Registers	**1.859.581**	**3.456.000**	**53.8**
DSP Slices	**4.564**	**12.288**	**37.14**
Block Ram	**1.950**	**2.016**	**96.7**

TABLE 5.8: Source Consumption for 40 Compute Units.

The resources of Alveo U250 are enough for the implementation of 40 compute units since none of the limits exceeded.

Also, in the next Table presented the execution time of accelerator card against the execution time of implemented algorithm in SPAdes and the speedup that succeed.

Data sets	Size of Data set(MB)	Execution Time of 40 Compute Unit(sec)	Execution Time of SPAdes with 16 Threads(sec)	Speedup
DRR015506	**5**	**2**	**20.4**	**10.2x**
DRR015482	**9**	**2.8**	**28.8**	**10.3x**
DRR015499	**17**	**6.5**	**43.6**	**6.7x**
DRR015480	**24**	**8.3**	**64.3**	**7.74x**
DRR015512	**36**	**19.7**	**135.8**	**6.88x**

TABLE 5.9: Execution time of SPAdes with 16 threads versus 30 Compute Units implementation in FPGA- Speedup

For the calculation of speedup we divide execution time of FPGA with 1 thread over 39.5 because of the losses of non ideal systems. Alveo U250 accelerator card succeed speedup in range [6.7x-10.3x] for all of the data sets and it seems to be suitable for our needs. An unpredictable factor is whether FPGA have the capability to route the total amount of compute units. Also, we notice that have less speedup as the data set size get larger.

Alveo U50 - Alveo U280

Board Specifications	Alveo U50 Accelerator Card	Alveo U280 Accelerator Card
LUTs	**872K**	**1.079K**
Registers	**1.743K**	**2.607K**
DSP Slices	**5.952**	**9.024**
Block Ram	**1.755**	**2.016**
HBM Memory Capacity	**8GB**	**8GB**
HBM Total Bandwidth	**316 GB/s**	**460 GB/s**
DDR Memory capacity	**-**	**32GB**
DDR Total Bandwidth	**-**	**38GB/s**
PCIe	**Gen3x 16, 2 x Gen4x 8, CCIX**	**Gen4x8 with CCIX**
Maximum Total Power	**75W**	**225W**

TABLE 5.10: Alveo U50-U280 Specifications.

Alveo U280 have the same storage space in HBMs like Alveo U50, it has higher throughput than Alveo U50 because use improved model of HBMs but both cover our needs,so we can implement 5 compute units in HBMs too. In addition, Alveo U280 have extra 32 GB in DDR Memory which is 4 times more than 8 GB,so we can implement 20 more compute units in Alveo U280 , **25 compute units** in total.The bandwidth of DDR memory does not consists problem.

In Table 5.11 presented resource consumption of 25 compute units.

Board Specifications	Used	Available	Utilization(%)
LUTs	**720.270**	**1.079.000**	**66.75**
Registers	**1.213.501**	**2.607.000**	**46.54**
DSP Slices	**2.854**	**9.024**	**31.62**
Block Ram	**1.290**	**2.016**	**64**

TABLE 5.11: Source Consumption for 25 Compute Units.

As it seems from the above Table the current accelerator have the necessary amount of available resources in order to implement 25 compute units.

In addition, compared the performance of FPGA against SPAdes and succeed speedup in the last column.

Data sets	Size of Data set(MB)	Execution Time of 25 Compute Unit(sec)	Execution Time of SPAdes with 16 Threads(sec)	Speedup
DRR015506	**5**	**2.75**	**20.4**	**7.41x**
DRR015482	**9**	**4.6**	**28.8**	**6.26x**
DRR015499	**17**	**10.54**	**43.6**	**4.13x**
DRR015480	**24**	**13.51**	**64.3**	**4.76x**
DRR015512	**36**	**31.9**	**135.8**	**4.24x**

TABLE 5.12: Execution time of SPAdes with 16 threads versus 25 Compute Units implementation in FPGA- Speedup

We work as in Alveo U50 for the calculation of theoretical speedup.Alveo U280 accelerator card can improve significant the performance of our implementation too.However the succeeded speedup is in range [4.24x-7.41x] which is less than the succeeded speed up of Alveo U250. Also, it is unpredictable if the FPGA have the necessary cables to route the compute units.

Alveo U50 - Virtex UltraScale+ HBM VU57P

Board Specifications	Alveo U50 Accelerator Card	Virtex UltraScale+ HBM VU57P Accelerator Card
LUTs	872K	1.304K
Registers	1.743K	2.607K
DSP Slices	5.952	9.024
Block Rams	1.755	2.016
HBM Memory Capacity	8GB	16GB
HBM Total Bandwidth	316 GB/s	316 GB/s
DDR Memory capacity	-	-
DDR Total Bandwidth	-	-
PCIe	Gen3x 16, 2x Gen4x 8, CCIX	Gen3x 16, Gen4x8, CCIX
Maximum Total Power	75W	75W

TABLE 5.13: Alveo U50-Virtex UltraScale+ HBM VU57P Specifications.

Last but not least, we found an accelerator card which do not belongs to Alveo family but in Virtex family and it is suitable for our needs. Both accelerator cards have the same bandwidth but the main difference between them is the storage space because Virtex Ultrascale+ have 16 GB storage space in HBMs and Alveo U50 have 8GB. Hence, the new accelerator card have the possibility to support **10 compute units**.

In the next Table checked the amount of resources that consumed by the hardware implementation in FPGA.

Board Specifications	Used	Available	Utilization(%)
LUTs	**346.725**	**1.304.000**	**26.58**
Registers	**567.421**	**2.607.000**	**21.76**
DSP Slices	**1.144**	**9.024**	**12.67**
Block Ram	**630**	**2.016**	**31.25**

TABLE 5.14: Source Consumption for 10 Compute Units.

The available resources are enough for the implementation of 10 compute units.

Further, we present in the next Table the execution time of the current FPGA against SPAdes and the succeed speedup.

Data sets	Size of Data set(MB)	Execution Time of 10 Compute Unit(sec)	Execution Time of SPAdes with 16 Threads(sec)	Speedup
DRR015506	**5**	**7.11**	**20.4**	**2.87x**
DRR015482	**9**	**11.91**	**28.8**	**2.42x**
DRR015499	**17**	**27.3**	**43.6**	**1.6x**
DRR015480	**24**	**34.98**	**64.3**	**1.84x**
DRR015512	**36**	**82.5**	**135.6**	**1.65x**

TABLE 5.15: Execution time of SPAdes with 16 threads versus 10 Compute Units implementation in FPGA- Speedup

The current FPGA double speedup than Alveo U50 which is expected because implemented double amount of compute units. The range of speedup is [1.65x-2.87x] which is the worst in order of the other two accelerator cards.

In conclusion, **Alveo U250** accelerator card have the best performance among the compared accelerator cards because have the most storage space, so it has the capability to support the implementation of 40 compute units. It succeeds speedup around 6.7x-10.3x for the specific data sets, also we notice that as the size of data set increased the speedup decreased.Further, all of the compared accelerator cards have theoretically better performance than

Alveo U50 because they have more storage space in order to implement more compute units. In addition, the data set of 36 MB produce data that allocates almost 8 GB, so **Alveo U50** accelerator card have reach its upper limit in storage space. **Alveo U250** have 8 times more storage space than **Alveo U50**, so it will be possible to execute data sets 8 time larger than 36 MB.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

Finally, De Bruijn graphs allocates big amount of storage space,so we need more storage space in order to execute larger data sets and implement more compute units.Also, due to lack of storage space we are able to execute small data sets of human genomes. Further, results of Alveo U50 accelerator card present an insignificant speedup for the small size data sets and worse performance than SPAdes.13.0 in larger data sets .We conclude that there are several reasons that make our configuration slower than SPAdes.13.0.These reasons are presented at next:

1. The priority queue have 7000 elements, so the tool can not partition the priority queue into shift registers in order to make it pipeline. The upper limit of tool in array partitioning is 1024 elements.
2. Also,each element of priority queue includes 4 unsigned integer, so we need to make 4 comparison in order to determine which data have higher priority. The specific factor complicates our configuration and the tool can not partition even array of 1024 elements.
3. The lack of storage space is a factor that limits the capabilities of our configuration because we can not implement more than 5 compute units.

In addition, we conclude that the process of pushing neighbours in priority queue consumes the vast majority of clock cycles because there are data dependencies between them and they can not pushed in parallel.In conclusion,fully optimized SPAdes.13.0 do not have much better execution time than our configuration because the data sets have small size, so SPAdes.13.0 can not exploit all of its capabilities. We believe that in much larger data sets

SPAdes.13.0 will have far better performance than our configuration in Alveo U50 accelerator card.

Furthermore, we succeed to find three accelerator cards of Xilinx which have more LUTs, Registers,Block Rams and DSPs than Alveo U50.At next we present advantages and disadvantages of each accelerator cards against the others

- **Alveo U250** dominates against the other accelerator cards in resources and in total storage space and is capable to implement 40 compute units in order to succeed the maximum possible speedup .It has a drawback which is the low bandwidth of DDR Memory against HBMs but it is not affects our implementation.
- **Alveo U280** is an accelerator card with more resources than Alveo U50 and more storage space from it. Also, it has both HBM and DDR Memory, so we can exploit them in order to implement 25 compute units and succeed a significant speedup.
- At last **Virtex UltraScale+** have more resources from Alveo U50 and double storage space. It is clearly that Virtex UltraScale+ can be used in order to implement double compute units and double the performance of the implementation.

Also,we conclude that the modern hardware design tools such as Vitis Unified and Vivado HLS eases the hardware designers because offers visualization of the hardware design and suggestions about the improvement of the hardware design. However, the hardware designers are responsible for the proper changes in the inner code in order to improve the hardware implementation.

6.2 Future Work

Our implementation have room for improvements because it allocates almost 14% of the total resources of FPGA and for each extra compute units it allocates 3% more resources.At next we refer some thoughts about future work in my configuration in order to optimize it.

- Firstly, we shown in Chapter 5 that there are accelerator cards which have better specifications than Alveo U50 mainly more storage space. Hence, in case we have available one of these FPGAs or some other in future we can implement our configuration in it for better performance.

- Further, in article [30] Aggelos Ioannou and Manolis Katevenis presenting a different implementation of priority queue in order to convert it into fully-pipeline binary heap.They convert the pushing process of data in priority queue, they push them from root of priority queue ,so there are not dependencies in pushing data.Also, they create many mini heaps that connected in order to not implement a general priority queue but many small priority queues. In future is possible to convert our implementation like this.
- Xilinx provides a feature named **PCIe Peer-to-Peer (P2P)** [31] communication is a PCIe feature which enables two PCIe devices to directly transfer data between each other without using host RAM as a temporary storage. The latest version of Alveo PCIe platforms support P2P feature via PCIe Resizeable BAR Capability.Data can be directly transferred between the DDR/HBM of one Alveo PCIe device and DDR/HBM of a second Alveo PCIe device,A thirdparty peer device like NVMe can directly read/write data from/to DDR/HBM of Alveo PCIe device In Chapter 5 presented accelerator cards that have compatible PCIe between them, so we can connect them by the specific feature.
- Also, we can design a new implementation in order to exploit the throughput of the HBM. We notice that the realistic throughput of HBM is 42 MB/s because of the priority queue. A suggestion for better exploitation of throughput is to implement the priority queue in the main memory of FPGA.

Appendix A

Genome Assembly

A.1 State of the Art

The necessity of the process of Genome Assembly created in 1953 when James Watson and Francis Crick discover the structure of DNA. Until now there are three different generations of technology for Genome Assembly.

The **first generation of technology** [32] created by Frederick Sanger in 1977 and it was the main technology for Genome Assembly for a long time. Unfortunately, the fact that the Sanger's method wasn't capable to produce results in reasonable amount of time for big genomes make the necessity for a new genome sequence method. For this reason the created the **second** and **third generation of technology** which are in the category of **Next Generation Sequence(NGS)** and the priority of this category is to treat the long genomes.

The year 2005 was crucial because companies such Illumina with Solexa sequencing , Roche with 454 sequencing , Applied Biosystem with Solid sequencing start developing the **second generation technology** [17] [32] . The sequencing methods that companies create break-up the genome into fragments and create files with **small reads (30-400 base pairs)** but with different way each method. In the **2nd generation technology** the genome assemblers reads parallel many small length sequences from a file which have produced from a sequencer, the assembler process them and compose them until read all the given genome. With this way it is faster to read the long genome and produce a very good result after the composition of the small sequences than the Sanger's method . The problem with the **2nd generation technology** is that needs too many computation resources to do the parallel read of the small sequences and the main problem is that with the small length reads is very difficult to locate repetives in the genome so it makes gaps in the final result.

In year 2011 two companies Oxford Nanopore Technologies and Pacific Biosciences start to evolve methods sequence the DNA into reads with length **>5000 base pairs** named **long reads** and start the era of **third generation technology**[17][32] in sequencing which is an extension of the 2nd generation , the main goal of the 3rd generation is to solve the problem of repetives. This can be achieved because the reads of the longer sequences (>5000 base pairs) than the previous generation(30-400 base pairs) have the ability to locate easier and faster the repetives that there are in the genome. Furthermore the specific generation needs smaller amount of computational resources than the previous generation. The disadvantage of the **3rd generation technology** is the reads that produced from the sequencing method have large read error rate about 10-15% beside the read error rate of the **second generation technology** is 0.5% and it is the next problem that need to be solved.

A.2 Next Generation Assembly Methods

In the next generation sequencing one of the most important problems is to determine the **genome assembler** that will use to do the genome assembly. The **genome assembler** is a software tool that receive as input a file which consists of sequences from a genome and try to detect overlaps between the sequences as a result to construct contigs and then scaffolds. The sequences that read every time the genome assembler from the file stored with a specific method in a data structure so be possible to process them and create the contigs, nowadays the dominant data structure for the storage of the reads are the graphs. Actually, almost all of the current genome assemblers store them into graphs. There are three common genome assembly methods , the first one is **De Bruijn Graph Assembly** based on **De Bruijn Graphs**, the second one is **Overlap Layout Consensus Assembly** based on **Overlap Graphs** and the third one is **Overlap Layout Consensus Assembly** based on **String Graphs**. The aim of the **Genome Assembler** is to give as a result **Contigs** and then **Scaffolds**.

A.3 Contigs – Scaffolds

A **Contig**[33] from the word "contiguous" is a series of overlapping DNA sequences used to make a physical map that reconstructs the original DNA sequence of a chromosome or a region of a chromosome. A contig can also refer to one of the DNA sequences used in making such a map.

A **Scaffold**[34] is a portion of the genome sequence, composed from contigs and gaps. Gaps occur where reads from the two sequenced ends of at least one fragment overlap with other reads in two different contigs. Since the lengths of the fragments are roughly known, the number of bases between contigs can be estimated. The goal of whole genome assembly is to represent each DNA sequence in one scaffold but it is not always possible and depending on how completely the genome can be reconstructed, or assembled, from the available reads.

Contigs are formed by merging **k-mers** appearing adjacently in reads halting at **k-mers** from repeat boundaries. This has the cost of requiring highly accurate reads, and it initially discards some of the ability for reads to resolve repeats longer than k bases. It has the benefit of not requiring the storage of pairwise overlaps and having a graph structure representative of the repeat structure of the genome. For these reasons,de Bruijn graph is widely used in sequence assembly tools.

A.4 Overlap Layout Consensus Assembly (OLC) – Overlap Graphs

The **Overlap Layout Consensus** [17][35] is an assembly algorithm that developed by Staden in 1980, the algorithm have **three steps** which described in the following paragraphs.

First step is to detect **overlaps(O)** between the reads and the **second step** is to **layout(L)** the reads and the overlaps on a **overlap graph** which is directed graph. The vertexes of the **Overlap Graph** are the reads from the file and the edges define the length of overlapping. For successful overlapping need to compare the suffix of source with the prefix of sink and have match of at least 3 characters. Also the overlap graphs can contain cycles which create because the DNA string of bacterial and mitochondrial is circular but the most common reason is because of the repetitive DNA across the reads.The Figure A.1 presents an example of cycle overlap graph and the way of construction:

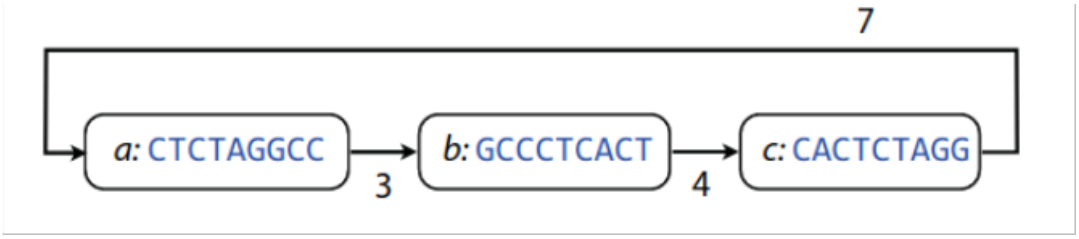

FIGURE A.1: OLC Overlap Graph.
Reference: Computational and Systems Biology Lecture of David Gifford in Biology Department of MIT"

As we can see in the graph is cycle, in the nodes there are the reads and on the edges the length of overlapping.

After the detect and the construct of the graph from the overlaps the **third step** of the algorithm is to compose the contigs that formed by iteratively merging overlapping reads until a read determined to be at the boundary of a repeat so it is unresolved repeat and stop the compose of the contig. Repeats which are shorter than the minimally expected read overlap are often resolved, implying that genome resolution increases with read length. Also, OLC algorithm search for Hamiltonian paths and some genome assemblers with OLC algorithm assembly are AMOS, Arachne and Celera .

A.5 Overlap Layout Consensus Assembly-String Graphs

String Graph presented in the paper of EG. Ehrlich, S. Even, and Robert Tarjan in 1976, the title of the paper is "Intersection graphs of curves in the plane". **The Overlap Layout Consensus Assembly (OLC)** [17][35] as said developed by Staden in 1980 and the algorithm remains the same as said in the previous paragraph, detect **overlaps(O)** between the reads and **layout(L)** the reads and the overlaps on a overlap graph which is directed graph. The vertexes of the Overlap Graph are the reads from the file and the edges define the length of overlapping. For successful overlapping need to compare the suffix of source with the prefix of sink and have match of at least 3 characters. Also the overlap graphs can contain cycles which create because the DNA string of bacterial and mitochondrial is circular but the most common reason is because of the repetitive DNA across the reads.

The difference between the **OLC** based on **Overlap Graph** and the **OLC** based on **String graph** is that after the construction of the overlap graph the graph simplified. After the simplifications from the Overlap Graph derived the String Graph. Firstly, removed duplicate reads from the graph (distinct elements of reads set with the same or reverse complemented sequence) and contained reads (elements in reads set that are a substring of some element in reads set or their reverse complements), removing edges that skip one or two node.After the end of simplification arise the **String Graph** with some non-branching stretches which are the contigs but there are branching stretches that are unresolvable repeats. Also check the subgraphs for repeats and sequencing errors and determine which subgraph is not a contig. Furthermore the algorithm use **Hamiltonian** approach to compose contigs from graph, Genome Assemblers that use OLC based on String graphs are SGA, FALCON, Canu.

A.6 Algorithm Comparison

The three algorithms that described above are very useful for Genome Assembly and used almost in all Genome Assemblers to represent their reads and store them. All of the algorithms have advantages and disadvantages and there is not any best algorithm but we can choose the better algorithm for our task.

Between **OLC algorithms** the **String Graph** need more time to construct because of the simplifications but need less memory because remove unnecessary information , also in the **String Graph** is easier to compose the contigs because of the simplification. The repeat detection is easier in **String Graph** but is equally difficult to solve them.

One of the advantages of **OLC algorithms** against **De Bruijn Graphs (DBG) Algorithm** is that use less memory because **DBG** decompose every read into r-k+1 nodes(r read length, k consecutive one read length from assembler) but the **OLC** need one node for each read. On the other hand the **DBG** need less time by far to compose the contigs than the **OLC** because the **OLC** use **Hamiltonian** approach which have larger time complexity than the **DBG** which use **Eulerian** approach. An other advantage for **OLC** is more suitable from **DBG** in long reads and in general in large genomes because of the k value the **DBG** need too much memory to store large genomes. Also the long reads have big read error rate which can controlled easier from the

OLC. The **DBG** are very good in short reads and in short genomes because have the ability to change the value of k and get better results.

Appendix B

SPAdes.3.13.0

B.1 Forward-Backward Reads Data set

The data sets split in two categories in the matter of the form of reads. These two categories are **Forward Reads Data set** and **Backward Reads Data set**, the **Backward Reads Data set** also named as **Reverse Reads Data set**.

The **Forward Reads** are the reads that start from the start point of the DNA to the endpoint (left to right).

The **Reverse Reads** are the opposite reads from the **Forward Reads** they start from the endpoint of the DNA to the start point of the DNA (right to left).

For that reason there are data sets with the same name but ends with **'1'** or **'2'** , the data sets that ends with **'1'** have **forward reads** and the data sets that ends with**'2'** have **backward reads**. The single-read library need only the the Forward Reads Data set but the **paired-end library** and **mate-pair library** need both of the data sets **Forward** and **Backward Reads** Data set.

B.2 SPAdes.3.13.0 Libraries

The **SPAdes.3.13.0** [13] use generally three different libraries for genome assembly. These three libraries are:

1) **Single-end Library**

2) **Paired-end Library**

3) **Mate Paired Library**

At **Single-end Library** the assembler need a single end data set as an input.The assembler use the data set to construct the graph and finally get the

contigs from the graph. The advantage of the **Single-end Library** is that finish the process faster than the other two libraries but the result is not as good as the other two libraries result.

At **Paired-end Library** the assembler need 2 data sets, one with **Forward Reads** and one with **Backward Reads**. Again the assembler construct the graph but this time use two data sets and it is easier to detect repetitive regions and complete gaps in the contigs. So the **Paired-end Library** have better results than the **Single-end Library** .

The third library that use the assembler is the **Mate Paired Library** which need 2 data sets too. One with **Forward Reads** and one data set with **Backward Reads** but for this library the length of each sequence read is longer than the sequence read length of the **Paired-end Library** data sets. The process remains the same as the previous libraries but the difference between the length affects in the result. The length of the sequence reads of the **Paired-end** data sets make the assembler to produce contigs with less gaps. On the other hand the length of the **Mate-Paired** data sets make the assembler to locate easier repetitive regions.

For the purpose of the thesis I use the **Paired-end Library** for genome assembly.

B.3 Graph Simplification

Graph simplification [4] is a very important part of the **SPAdes** because it is the final step before the assembler produce the contigs. Also , consume much time from the process almost 35% of the total time. A term that often used is **h-path**. The terminology for the term **h-path** is: a vertex v in a graph G is called a hub if indegree(v) $\neq$ 1 or outdegree(v) $\neq$ 1 . A directed path in G is called a hub-path if it is starting and ending vertices are hubs and it is intermediate vertices are not hubs. The criteria for the assembler to detect an error are the topology,the length and the coverage of the path. SPAdes split the graph simplification into 4 processes,the processes are **tip removal**,**bulge corremoval**, **chimeric h-path removal** and **isolated h-paths removal**. Some details about them in next:

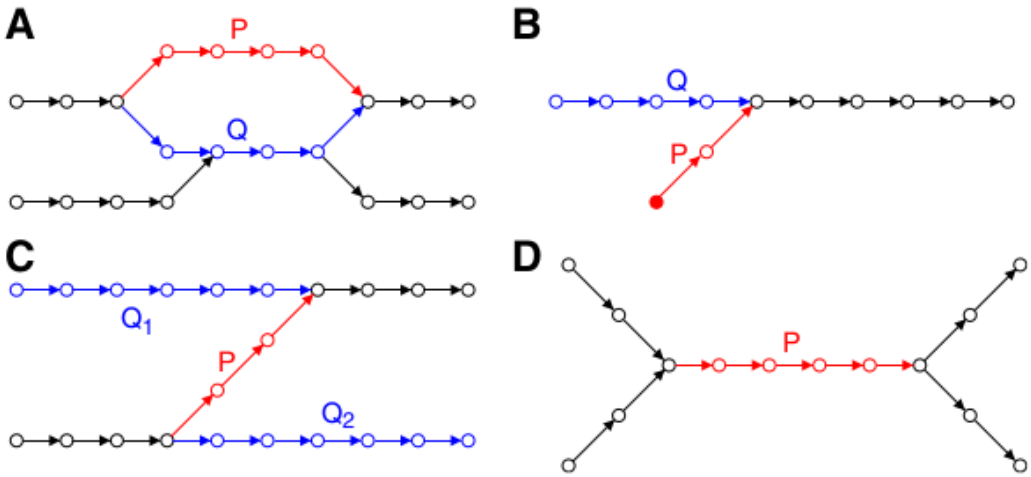

FIGURE B.1: Simplification Cases.
SPAdes: A New Genome Assembly Algorithm and Its Applications to Single-Cell Sequencing

•Bulge Corremoval

Miscalled bases and indels in the middle of a read typically lead to **bulges**. **Bulges** also arise from small variations between repeats in the genome. The assembler detect the bulges and remove them, a bulge as it seems in Figure B.1 A is the path P which does not contain hubs within it, though Q does, so the P path is a bulge and removed from the graph.

•Tip Removal

Errors near the ends of reads may lead to **tips**, short, stray h-paths, with one end having total degree 1, Figure B.1 B present a **tip**. The h-path P starts or ends at a vertex of total degree 1 (re-presented as solid), and there is an alternative h-path Q so the P h-path is a **tip** and removed from the graph.

•Chimeric h-path Removal

Chimeric reads may lead to erroneous connections in the graph, called **chimeric h-paths** . **Chimeric h-paths** may also arise from identical errors near the start of one read and near the end of another. In Figure B.1 C there are 2 alternative h-paths Q1 and Q2 both for the entrance and the exit to P so P h-path is a **chimeric h-path** and removed from the graph.

•Isolated h-path

Input data often contains low quality reads that don't map to the genome and which may result in short, low coverage, **isolated h-paths**. In Figure B.1 D, P starts with a vertex of out-degree one and ends with a vertex of in-degree one and has no alternative h-path so P removed from the graph as an **isolated h-path**.

B.4 File Formats

• **FastA Format**

FastA format is a text-based format for representing nucleotide sequences , in which base pairs are represented using single-letter codes. A sequence in **FastA format** begins with a single-line description, followed by lines of sequence data. The description line is distinguished from the sequence data by a greater-than (">") symbol in the first column. The output files of the **SPAdes** that have the scaffolds and the contigs are in FastA format.

•**FastQ Format**

The **SPAdes** need data sets which are in **fastq.gz** format to run and sequence the genome. In general the **FastQ file** is a text file which contains clusters that produced from Illumina , Ion Torrent, PacBio, Oxford Nanopore sequencing tools and they consist of 4 lines:

1. A sequence identifier(label) with information about the sequencing run and the cluster.

2. The sequence (the base calls: A, C, T, G and N).

3. A separator, which is simply a plus (+) sign.

4. The base call quality scores. These are Phred +33 encoded, using ASCII characters to represent the numerical quality scores.

The ASCII character '!' is the worst quality score and the base call is incorrect for sure as the Probability Error is 1. The P-error is reducing as the ASCII code increasing.

References

[1] Tanja van Aardenne-Ehrenfests, Nicolaas Govert de Bruijn. "Circuits and trees in oriented linear graphs". In: *Simon Stevin 28* (May 1951), pp. 203–217.

[2] Sharat Chandra Varma, Paul Kolin, Madan Balakrishnan, Dominique Lavenier. "FAssem : FPGA based Acceleration of De Novo Genome Assembly". In: *21st Annual International IEEE Symposium on FCCM* (Apr. 2013), pp. 1–5.

[3] Carl Poirier, Benoit Gosselin, Paul Fortier. "DNA Assembly with De Bruijn Graphs Using an FPGA Platform". In: *IEEE/ACM Transactions on Computational Biology and Bioinformatics* (May 2018), pp. 1003–1009.

[4] Alexey Gurevich, Alexander Kulikov, Sergey Nikolenko, Son Kim Pham. "SPAdes: A New Genome Assembly Algorithm and Its Applications to Single-Cell Sequencing". In: *Journal of computational biology: a journal of computational molecular cell biology* (Apr. 2012), pp. 1–24.

[5] "European Nucleotide Archive (ENA)". In: (). URL: https://www.ebi.ac.uk/ena/browser/home.

[6] "Alveo U50 datasheet". In: (). URL: https://www.xilinx.com/products/boards-and-kits/alveo/u50.html#overview.

[7] "Center for Algorithmic Biotechnology". In: (). URL: https://cab.spbu.ru/.

[8] Phillip Compeau. "How to apply de Bruijn graphs to genome assembly". In: *Nature Biotechnology* (Nov. 2011), pp. 987–991.

[9] "Open Notes of Ben Langmead for de Bruijn Graphs". In: *Langmead Lab of Johns Hopkins University* (). URL: https://www.cs.jhu.edu/~langmea/resources/lecture_notes/assembly_dbg.pdf.

[10] Ashish Kumar. "De-Bruijn Sequence and Application in Graph theory". In: *International Journals of Sciences and High Technologies* (June 2016), pp. 4–17.

[11] "SOAPdenovo assembler manual". In: (). URL: https://www.animalgenome.org/bioinfo/resources/manuals/SOAP.html.

[12] "Velvet assembler manual". In: (). URL: https://www.ebi.ac.uk/~zerbino/velvet/.

[13] "SPAdes assembler manual". In: (). URL: https://cab.spbu.ru/files/release3.13.0/manual.html.

[14] "ALLPATHS assembler manual". In: (). URL: https://software.broadinstitute.org/allpaths-lg/blog/.

[15] Betsy Foxman. "Molecular Tools and Infectious Disease Epidemiology". In: *Chapter 5 A Primer of Molecular Biology 5.10.2 Gene Assembly* (Jan. 2011).

[16] Wenyu Zhang, Jiajia Chen, Yang Yang, Yifei Tang, Jing Shang, Bairong Shen. "A Practical Comparison of De Novo Genome Assembly Software Tools for Next-Generation Sequencing Technologies". In: *Article in PLoS ONE* 6 (Mar. 2011).

[17] Xingyu Liao, Min Li, You Zou, Fang Xiang Wu, Yi Pan, Jianxin Wang. "Current challenges and solutions of de novo assembly". In: *Quantitative Biology, Higher Education Press and Springer-Verlag GmbH Germany, part of Springer Nature* (June 2019), pp. 90–109.

[18] Rayan Chikhi, Guillaume Rizk. "Space-efficient and exact de Bruijn graph representation based on a Bloom filter". In: *Algorithms for Molecular Biology* (Sept. 2013), 1–12.

[19] "ABySS genome assembler". In: (). URL: https://www.bcgsc.ca/resources/software/abyss.

[20] Sébastien Boisvert, François Laviolette, Jacques Corbeil. "Ray: Simultaneous assembly of reads from a mix of high-throughput sequencing technologies". In: *Computational Biology* (July 2010), 1519–1533.

[21] Haixiang Shi, Bertil Schmidt, Weiguo Liu, Wolfgang Müller-Wittig. "A Parallel Algorithm for Error Correction in High-Throughput Short-Read Data on CUDA-Enabled Graphics Hardware". In: *Journal of Computational Biology 17.4* (June 2010), pp. 603–615.

[22] Anand Ramachandran, Yun Heo, Wen-mei Hwu, Jian Ma, Deming Chen. "FPGA Accelerated DNA Error Correction". In: *Design, Automation and Test in Europe Conference and Exhibition* (Aug. 2015), pp. 1371–1376.

[23] Yun Heo, Xiao-Long Wu, Deming Chen, Jian Ma, Wen-Mei Hwu. "BLESS: Bloom filter-based error correction solution for high-throughput sequencing reads". In: *Oxford University Press (OUP): Bioinformatics*, (May 2014), pp. 1354–1362.

[24] Burton Bloom. "Space/time trade-offs in hash coding with allowable errors". In: *Community ACM* (July 1970), pp. 422–426.

[25] "Vitis Unified Software Platform". In: (). URL: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_2/ug1400-vitis-embedded.pdf.

[26] "Vitis HLS -Vivado IDE user guide". In: (). URL: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_3/ug902-vivado-high-level-synthesis.pdf.

[27] "Alveo U250 datasheet". In: (). URL: https://www.xilinx.com/support/documentation/data_sheets/ds962-u200-u250.pdf.

[28] "Alveo U280 datasheet". In: (). URL: https://www.xilinx.com/support/documentation/data_sheets/ds963-u280.pdf.

[29] "Virtex UltraScale+ HBM datasheet". In: (). URL: https://www.xilinx.com/support/documentation/data_sheets/ds890-ultrascale-overview.pdf.

[30] Aggelos Ioannou, Manolis Katevenis. "Pipelined Heap (Priority Queue) Management for Advanced Scheduling in High-Speed Networks". In: *IEEE/ACM Transactions on Networking (ToN)* (July 2007), pp. 1–6.

[31] "PCIe Peer-to-Peer (P2P)". In: (). URL: https://xilinx.github.io/XRT/master/html/p2p.html.

[32] Mehdi Kchouk, Jean François Gibrat, Mourad Elloumi. "Generations of Sequencing Technologies: From First to Next Generation". In: *Biology and Medicine (Aligarh)* 9 (Mar. 2017), pp. 1–8.

[33] "Term of Contig". In: *National Human Genome Research Institute* (). URL: https://www.genome.gov/genetics-glossary/Contig.

[34] "Term of Scaffold". In: *Joint Genome Institute* (). URL: https://mycocosm.jgi.doe.gov/help/scaffolds.jsf.

[35] "Computational and Systems Biology Lecture of David Gifford". In: *Biology Department of MIT* (). URL: https://ocw.mit.edu/courses/biology/7-91j-foundations-of-computational-and-systems-biology-spring-2014/lecture-slides/MIT7_91JS14_Lecture6.pdf.