TECHNICAL UNIVERSITY OF CRETE

DIPLOMA THESIS

Design and Implementation of a cloud based FPGA accelerator for phylogeny reconstruction

---

Author:
Anastasios BOKALIDIS

Thesis Committee:
Prof. Apostolos DOLLAS
Prof. Michalis ZERVAKIS
Asst. Prof. Nikolaos ALACHIOTIS (U TWENTE)

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

October 15, 2021

TECHNICAL UNIVERSITY OF CRETE

Abstract

School of Electrical and Computer Engineering

Electrical and Computer Engineer

Σχεδιασμός και Υλοποίηση ενός FPGA Επιταχυντή Πάνω σε Νέφος για Ανασυγκρότηση Φυλογενετικών Σχέσεων

by Anastasios BOKALIDIS

Μιά απο τις πιο ενδιαφέρουσες προκλήσεις του 21ου αιώνα, που έχουν να αντιμετωπίσουν οι επιστήμονες είναι η ραγδαία και συνεχόμενη αύξηση των δεδομένων. Πολλοί τομείς της επιστήμης και της τεχνολογίας αντιμετωπίζουν προβλήματα στην διαχείρηση και την επεξεργασία των τεράστιων δεδομένων. Ένας απο αυτούς τους είναι η Βιολογία. Η διαδικασία την φυλογεννετικής ανάλυσης των DNA, RNA, πρωτεϊνών και άλλων τύπων φυλεγέννεσης, καταναλώνει αρκετό χρόνο, η οποία μάλιστα παρουσιάζει και μια μη γραμμική αύξηση όσο αυξάνεται ο όγκος των δεδομένων χρήσης. Επιπλεόν δεν είναι μόνο ο χρόνος που απασχολεί τους επιστήμονες αλλά επίσης και τα υπολογιστικά συστήματα που χρειάζονται για τον παραπάνω σκοπό. Όχι μόνο οι προσωπικοί υπολογίστες δεν μπορούν να εξαλείψουν το πρόβλημα, αλλά ακομή και οι υπερυπολογιστές μπορούν δεν μπορούν να καλύψουν τις ανάγκες μπροστά στον υπερόγκο δεδομένων. Οι πρώτοι έχουν επεξεργαστές οι οποίοι δεν μπορούν να ξεπεράσουν ένα κατώφλι επιτάχυνσης και παραλληλισμού των εφαρμογών που θέλουμε, και οι δεύτεροι χρησιμοποιούνται μόνο για ειδικές μελέτες και υπολογισμούς. Σε αυτή την εργασία, γίνεται μια μελέτη πάνω σε έναν αλγόριθμο φυλογεννετικής ανάλυσης, RAxML, ο οποίος βασίζεται πάνω στην μέθοδο της μέγιστης πιθανοφάνειας. Ο σκοπός αυτής της εργασίας είναι να βελτιώσουμε επιταχύνοντας κάποιες συναρτήσεις του RAxML, οι οποίες καταναλώνουν περισσότερο απο το 80% του συνολικού χρόνου εκτέλεσης και ειδικά όταν βρίσκονται υπό επεξεργασία μεγάλα αρχεία δεδομένων. Οπότε, πρώτο βήμα είναι να μελετήσουμε πως συμπεριφέρεται ο RAxML ανάλογα με τα δεδομένα εισόδου και ως δεύτερο βήμα να σχεδιάσουμε και να κατασκευάσουμε επιταχυντές όπου είναι απαραίτητοι για να βελτιώσουμε την απόδοση. Αυτοί οι επιταχυντές σχεδιάζονται ώστε να εγκατασταθούν πάνω σε FPGAs αλλά και σε αντίστοιχες πλατφόρμες νέφους της Amazon. Τέλος, γίνεται μελέτη και σύγκριση των αποτελεσμάτων του αρχικού αλγορίθμου με τους αντίστοιχους επιταχυντές μας και επίσης παρουσιάζεται ένα θεωρητικό μοντέλο για το πως θα ήταν η βέλτιστη συμπεριφορά των επιταχυντών μας και ποιά η βελτίωση που θα πρόσφεραν σε συνολικό επίπεδο στον αλγόριθμο…

TECHNICAL UNIVERSITY OF CRETE

Abstract

School of Electrical and Computer Engineering

Electrical and Computer Engineer

Design and Implementation of a cloud based FPGA accelerator for phylogeny reconstruction

by Anastasios BOKALIDIS

One of the most interesting challenges through the 21st century, that researchers have to encounter is the rapid and continuous increase of data. Many fields of science and technology face problems in the management and processing of vast data. One of them is Biology. The process of phylogenetic analysis of DNA, RNA, Protein, and other types of phylogenies, consumes a lot of time which performs a non-linear increase while the volume of the data for processing tends to increase. In addition, it is not only the time which is of concern to the scientists but also, the computing systems which are needed for this purpose. Not even personal computers can eliminate this problem, but also high-performance computers are inadequate to face up vast data. The first ones have CPUs that can not surpass a threshold in speed up and parallelism and the second ones are used only for special studies and computations. In this project, there is a study on a phylogenetic analysis algorithm, RAxML, which is based on the maximum likelihood method. The purpose of this project is to optimize by accelerating some functions of RAxML which consume more than 80% of the total execution time and especially under the processing of big data sets. So, the first step is to research the way that RAxML behaves according to the input data and the second step is to design and construct hardware accelerators required for optimal performance. These accelerators are designed to be mapped and routed on FPGAs and also on similar platforms of the Amazons' cloud. Finally, there is a study and comparison between the results coming from the initial algorithm and the results that come from the accelerators. Moreover, a theoretical model is introduced which shows the optimal performance of the accelerators and how it can affect the overall performance of the algorithm…

Acknowledgements

First and foremost, I would like to thank my supervisor, Prof. Apostolos Dollas, for his invaluable support and guidance throughout both the course of my studies and the progress of this thesis, being an inspiration at a professional and personal level, generously providing his expertise and experience. I would also like to thank him for his advice about future career choices and the opportunity he gave me to collaborate with the MHL team of the TUC.

Definitely, my thanksgivings go to the Asst. Prof. Nikolaos Alachiotis of the University of Twente who was my advisor for my thesis. There was an excellent collaboration between us and his willingness to guide me during the progress of my work was praiseworthy. These guidelines were also attributed to his expertise in bioinformatics and computer science.

Moreover, I would like to thank Mr. Dimitrios Theodoropoulos for his guidance on how to proceed with the flow of my thesis.

In addition, I would like to acknowledge the TUC MHL staff, including but not limited to Mr. Andreas Brokalakis, Mr. Pavlos Malakonakis, and Mr. Markos Kimionis, whose support was essential for this work as they helped me on using the required tools.

Last but not least, I would like to express my deepest gratitude to my friends and especially to my family who believed in me to achieve one of my life goals and supported me till the end of my graduation.

Bokalidis Anastasios,
Chania,2021

Contents

Abstract **iii**

Acknowledgements **v**

Contents **vi**

List of Figures **ix**

List of Tables **xii**

List of Algorithms **xiv**

List of Abbreviations **xv**

1 Introduction **1**

1.1 Introduction 1

1.2 Thesis Outline 3

2 Theoretical Background **4**

2.1 Phylogenetics - Basic Principles 4

2.2 Phylogeny - Phologenetic Trees 5

2.2.1 Types Of Trees 6

2.2.2 Enumerating trees 7

2.3 Implementation Of Phylogenies With The Use of Software . . 8

2.4 RAxML 9

3 Related Work **11**

3.1 Sequence Comparison 11

3.2 Multiple Sequence Alignment 12

3.3 Prediction of RNA and protein secondary structure 12

3.4 Gene identification 13

3.5 Phylogenetic Trees 13

3.6 Thesis Approach 15

4 Maximum Likelihood Method **17**
4.1 Maximum Likelihood 17
4.2 Models of Amino Acid Replacement 18
4.3 Calculating Tree's Probability 19
4.4 Adjustments on ML for RAxML 24
4.5 Phylogenetic Likelihood Library 30

5 Hardware Architecture Of Accelerator **31**
5.1 Profiling of PLL 31
5.2 DAER Architecture 34
5.3 Kernel Architecture 36
5.3.1 Fetch Units 36
5.3.2 Processing Units 37
A) PLF 37
B) SumGammaPROT 49

6 FPGA Implementation **53**
6.1 Tools Used 53
6.1.1 Vivado High Level Synthesis (HLS) 53
Synthesis Report 54
Optimization Directives 55
6.1.2 Vivado SDx 56
SDSoc 56
SDAccel 57
6.1.3 Vivado IDE 57
6.2 FPGA Platforms 58
6.2.1 ZCU102 58
6.2.2 AWS EC2 F1 Instance 59
6.3 OpenCL Host 60

7 Results **63**
7.1 Datasets 63
7.2 Software Performance 64
7.3 Hardware Performance 69
7.3.1 ZCU102 69
7.3.2 AWS F1 Instance 74
7.4 Performance Model 81
7.5 Analysis and Final Performance 87

8 Conclusions and Future Work **90**
8.1 Conclusions 90
8.2 Future Work 91

References **92**

List of Figures

2.1 A phylogenetic tree showing the three life domains: bacteria, archaea, and eukaryota. URL: `https://commons.wikimedia.org/wiki/File:Phylogenetic_Tree_of_Life.png` 6
2.2 Sample of (A) a rooted tree and (B) an unrooted tree. 7

4.1 A-B-C tree 20
4.2 Calculation of sequence A, assumed being ancestor of B and C 21
4.3 Calculation of final likelihood of common ancestor A 23
4.4 Substitution Matrix $4 \times 20 \times 20$ of Discrete Γ Model. On the single 20×20 matrix P symbolizes the probability and the other two letters the correlation of all amino acids following the table 4.1 25
4.5 Calculation of sequence A, assumed being ancestor of B and C under RAxML modulations 26
4.6 EigenVectors EV Matrix 28
4.7 Final operations between Eigenvectors and Likelihoods 29

5.1 Min percentage-PLF 32
5.2 Max percentage-PLF 33
5.3 Proposed Decoupled Access-Execute Architecture for Reconfigurable accelerators (DAER). 35
5.4 Decoupled Access-Execute Architecture On Our Designs . . . 36
5.5 Multiplications For The Calculation of UMP for X1,X2 input vectors 40
5.6 Summation of TMP vector to produce UMP_{X1} and UMP_{X2} and their multiplication for X1pX2 41
5.7 Calculation of likelihoods for a j site 42
5.8 Data Flow of Processing Unit : Likelihood Calculation 43
5.9 Multiplication Of EV with X1pX2 and a recursive addition for X3 44
5.10 Final Calculation of a j site of X3 vector 45
5.11 Scaling process of X3 vector 46

5.12 Data Flow of Processing Unit : EV Final Calculation 47
5.13 Data Flow of Main Core 48
5.14 SumGAMMA - Flow Chart 50
5.15 SumGAMMA - Detailed Data Flow 51
5.16 Data Flow of Main Core 52

6.1 OpenCL Memory Model 61

7.1 1.024 Alignment Patterns-SW 65
7.2 2.048 Alignment Patterns-SW 65
7.3 4.096 Alignment Patterns-SW 66
7.4 8.192 Alignment Patterns-SW 66
7.5 16.384 Alignment Patterns-SW 67
7.6 32.768 Alignment Patterns-SW 67
7.7 65.536 Alignment Patterns-SW 68
7.8 131.072 Alignment Patterns-SW 68
7.9 262.144 Alignment Patterns-SW 69
7.10 Execution Times Of SumGAMMAPROT Compared with Sequential Version 70
7.11 Execution Times Of NewViewGTRGAMMAPROT Compared with Sequential Version 71
7.12 Performance of Speedup for NewViewGTRGAMMAPROT Compared With Sequential Version-ZCU102 72
7.13 Execution Times Of NewViewGTRGAMMAPROT Compared with AVX Version 72
7.14 Execution Times Of SumGAMMAPROT kernel On AWS F1 (1) 74
7.15 Execution Times Of SumGAMMAPROT kernel On AWS F1 (2) 75
7.16 Total Execution Time Of SumGAMMAPROT Kernel on AWS F1 76
7.17 Execution Times Of NewViewGTRGAMMAPROT kernel On AWS F1 (1) 77
7.18 Execution Times Of NewViewGTRGAMMAPROT kernel On AWS F1 (2) 78
7.19 Execution Times Of NewViewGTRGAMMAPROT Kernel . . . 79
7.20 Performance of Speedup for NewViewGTRGAMMAPROT Compared With Sequential Version-AWS F1 81
7.21 Model of the Kernels Performance using 10 taxa 82
7.22 Model of the Kernels Performance using 25 taxa 83
7.23 Model of the Kernels Performance using 50 taxa 84
7.24 Model of the Kernels Performance using 100 taxa 85

7.25 Final Performance Of Speedup of NewViewGTRGAMMAPROT kernel (1) 86
7.26 Final Performance Of Speedup of NewViewGTRGAMMAPROT kernel (2) 87

List of Tables

2.1 All possible trees with and without root resulting in species species 8

3.1 Accelerated Algorithms on different technologies according to their processing Data 16

4.1 Amino acid frequencies (P_i). Source : `https://www.ebi.ac.uk/goldman-srv/WAG/wag.dat` 19

6.1 Platform's Software Components 59
6.2 Used Resources on ZCU102. 59
6.3 Used Resources on AWS F1 instance. 60

7.1 Server's hardware specifications 64
7.2 Performance of SumGAMMAPROT Hardware Kernel Compared with Sequential on ZCU102 70
7.3 Performance of NewViewGTRGAMMAPROT Hardware Kernel Compared with Sequential on ZCU102 71
7.4 Performance of NewViewGTRGAMMAPROT Hardware Kernel Compared with AVX on ZCU102 73
7.5 Performance of SumGAMMAPROT Hardware Kernel On AWS F1 76
7.6 Performance of NewViewGTRGAMMAPROT Hardware Kernel Compared With Sequential Version 80
7.7 Performance of NewViewGTRGAMMAPROT Hardware Kernel Compared With AVX Version 80
7.8 Performance of Model of NewViewGTRGAMMAPROT Hardware Kernel Compared With AVX Version (10 Taxa) 82
7.9 Performance of Model of NewViewGTRGAMMAPROT Hardware Kernel Compared With AVX Version (25 Taxa) 83
7.10 Performance of Model of NewViewGTRGAMMAPROT Hardware Kernel Compared With AVX Version (50 Taxa) 84

7.11 Performance of Model of NewViewGTRGAMMAPROT Hardware Kernel Compared With AVX Version (100 Taxa) 85

List of Algorithms

1 PLF , based on a three vector tree with root A 38
2 SumGamma Calculation 49

List of Abbreviations

AVX	**A**dvanced **V**ector **E**xtensions
AWS	**A**mazon **W**eb **S**ervices
BRAM	**B**lock **R**andom **A**ccess **M**emory
CPU	**C**entral **P**rocessor **U**nit
DAER	**D**ecoupled **A**ccess **E**xecute **AR**chitecture
DDR4	**D**ouble **D**ata **R**ate type **4** memory
DRAM	**D**ynamic **R**andom **A**ccess **M**emory
DSP	**D**igital **S**ignal **P**rocessor
FF	**F**lip **F**lops
FIFO	**F**irst **I**n **F**irst **O**ut
FPGA	**F**ield **P**rogrammable **G**ate **A**rray
DDR4	**D**ouble **D**ata **R**ate type **4** memory
GPU	**G**raphic **P**rocessor **U**nit
HBM	**H**igh **B**andwidth **M**emory
HLS	**H**igh **L**evel **S**ynthesis
HPC	**H**ight **P**erformance **C**omputing
II	**I**nitiation **I**nterval
LUT	**L**ook **U**p **T**able
ML	**M**aximum **L**ikelihood
MPI	**M**essage **P**assing **I**nterface
MPSoC	**M**ulti **P**rocessor **S**ystem **o**n **C**hip
PL	**P**rogrammable **L**ogic
PLF	**P**hylogenetic **L**ikelihood **F**unction
PLL	**P**hylogenetic **L**ikelihood **L**ibrary
PS	**P**rocessing **S**ystem
RAM	**R**andom **A**ccess **M**emory
RAxML	**R**andomized **A**xelerated **M**aximum **L**ikelihood
SDK	**S**oftware **D**evelopment **K**it
SDSoc	**S**oftware **D**evelopment **S**ystem **O**n **C**hip
SSE	**S**treaming **SIMD** **E**xtensions
URAM	**U**ltra **R**andom **A**ccess **M**emory

Dedicated to my family and friends. . .

Chapter 1

Introduction

The first chapter contains information about the process of evolution and the motivation of this work. In addition, there is a brief structure of this thesis.

1.1 Introduction

Biology is one of the main fields of science and its rapid evolution attracts many people and other fields of science to be involved in it. Through biology, humankind has discovered species, organisms, diseases and other living species, which contributed to the expansion of his knowledge about ourselves and nature. One of the most remarkable achievements is that scientists found out humans' ancestors and their origins since the first eras of mankind. Human DNA, as well as the DNA of other organisms, are interconnected creating a large tree of nature, a tree of life. Apart from DNA, biologists achieved to analyse and compare more genetic materials such as RNA, Proteins, Acids to interconnect not only the same organisms, but also micro-organisms, viruses, bacteria, and fungi.

Although more species are discovered every day and they get into the tree of life, the magnitude of big data problems gets tremendous dimensions. It is not only the unique features of species that vary between others but also some features that differentiate two units of the same species. So it is obvious how data volume rapidly increases. So, according to this thesis, the evolution grade of the tree of life is getting bigger while the number of species and their unique features is differentiated, in other words, each newly inserted piece of information burdens the problem separately.

Nowadays, many algorithms construct the tree of life and they vary in species, implementation techniques, and variables, but all of them face the problem of big data. Approximately, the time which is spent to analyze, compare and construct the trees of life is equal among these algorithms, passing on them the same volume of data. Theoretically, the problem could have been resolved, considering that high-performance processors-computer systems have been integrated into our daily life. Even more, supercomputers that have operations to speed up different types of algorithms could resolve this issue. Although the above solutions seem to be optimistic, these technology architectures focus on executing a wide variety of instructions and approach a fulfilling performance. The instructions of the algorithms demand many operations such as additions, subtractions, and multiplications, to be executed in a rapid and recurring way. In case that these operations are converted to commands of one CPU, the CPU will execute the operations with several unnecessary stages between them. Moreover, some CPU circuits cannot be executed to meet a user's requirements, because CPUs are constructed in such a way so as to include all possible required stages in their execution, whether it is a simple operation or a set of complex commands. As a consequence, this delays every CPU execution.

Since 1984, the FPGA industry delivered the first reprogrammable logic device which urged scientists to invest in them. Till the beginning of 20^{th} century, FPGA devices were integrated in many technology fields. This overusage is credited to some capabilities of FPGA. That is the allocation of their resources in an efficient way that users can design from a simple logic gate to a complicated circuit, the low power efficiency, and their ability to be reconfigured both locally and globally on their designs. So, exploiting the advantages of FPGAs and recognizing the specificity of bioinformatics computing systems, the contribution of this thesis is to:

- Find a solution to a problem that burdens the surveys on bioinformatics and specific on phylogeny.
- Describe some possible accelerators of time-consuming functions.
- Design and construct units that implement the requisite computations of the accelerator.
- Distribute and manage the available resource in order to surpass a simple CPU's speed.
- Propose hardware platforms that can load and execute effectively the demands of the accelerator.

- Make this implementation able to be improved on future work, given its restrictions.

1.2 Thesis Outline

- **Chapter 2 - Theoretical Background:** There is a reference on some principles of Phylogenetics and how they are used in algorithms.
- **Chapter 3 - Related Work:** There is a reference on some relevant previously implemented works and a detailed reference on previous works on the same topic as ours.
- **Chapter 4 - Maximum Likelihood Method:** The method of ML under the use of protein data is described and its differentiation on RAxML. There is also a reference to the PLL's functions.
- **Chapter 5 - Hardware Architecture Of Accelerator:** The results of the profiling on the PLL are presented and then the architecture of the accelerators of the chosen functions.
- **Chapter 6 - FPGA Implementation:** There is a description of the tools that are used for this work, the FPGA platforms that accelerators installed on, and a brief reference on OpenCL, which is used on the implementation of the host.
- **Chapter 7 - Results:** The results that came from software executions are compared with the results that came from hardware executions. There is also a theoretical framework on how the design could perform with the usage of a larger platform.
- **Chapter 8 - Conclusions And Future Work:** The conclusion which came from the execution of the design and its implementation is described and some ideas for improvement are also proposed.

***It is of high importance to mention that a part of this thesis is presented on the paper [1] which was submitted and verified on IEEE Micro this year.*

Chapter 2

Theoretical Background

In this chapter, the subject which is under discussion is the problem of identifying phylogenetic relationships, which are derived from different organisms. In other words, how we can determine the evolutionary relationships of these organisms with criteria, the amino acid sequence of some proteins (or the sequence of the corresponding genes). Additionally, there is a reference on how the RAxML software works and its use in this study.

2.1 Phylogenetics - Basic Principles

Over the years, as many fields of science are being developed, Biology and its fields try to make their revolution, and in order to achieve it there must be a collaboration with computer science and new technology innovations. On this topic, it can be assumed that there is a combination of different fields of Biology with computer science, such as Bioinformatics and Phylogenetics (part of Structural Biology). But, among these fields, Phylogenetics is the one which concerns us on this thesis. Phylogenetics is a part of systematics that addresses the inference of the evolutionary history and relationships among or within groups of organisms (e.g. species, or more inclusive taxa). These relationships are hypothesized by phylogenetic inference methods that evaluate observed heritable traits, such as DNA and protein sequences or morphology, often under a specified model of the evolution of these traits. The result of such an analysis is a phylogeny (also known as a phylogenetic tree) — a diagrammatic hypothesis of relationships that reflects the evolutionary history of a group of organisms.

Following the principles of Phylogenetics, biologists make a big effort to collect all the appropriate data so as to draw out conclusions about the evolutionary history of species, their ancestors, or maybe group some species according to common ancestors. Nevertheless, it is necessary to know what we are going to compare. If we want to evaluate phylogenetics relationships from the sequences of some genes, we have to compare corresponding genes, so as to trace their homology. Called ***Homologous*** proteins (or genes) are generally the proteins that have arisen through the evolution of a common ancestor. The corresponding genes in different organisms are also referred to as ***orthologues***, and it is considered that any differentiation has occurred due to specialisation. In contrast, homologous proteins, or genes, within the same species are called ***paralogues***, and we believe that they have arisen from gene duplication and independent evolution within the kind. Finally, there are the so-called ***xenologues*** genes, which are homologous genes that have arisen from a process of horizontal gene transfer (usually from a prokaryotic organism). An example of the first case is the alpha-chains of mammalians hemoglobin (e.g. humans, chimpanzees, dogs, etc.), while for the second case we could mention within the same species (e.g. humans), the alpha, betta, gamma-chains of hemoglobin but also myoglobins.

2.2 Phylogeny - Phologenetic Trees

All the above theories are associated with the phylogenetic tree. A phylogenetic tree is a branching diagram or "tree" showing the evolutionary relationships among various biological species or other entities—their phylogeny—based upon similarities and differences in their physical or genetic characteristics. All living species on Earth are part of a single phylogenetic tree, indicating common ancestry.

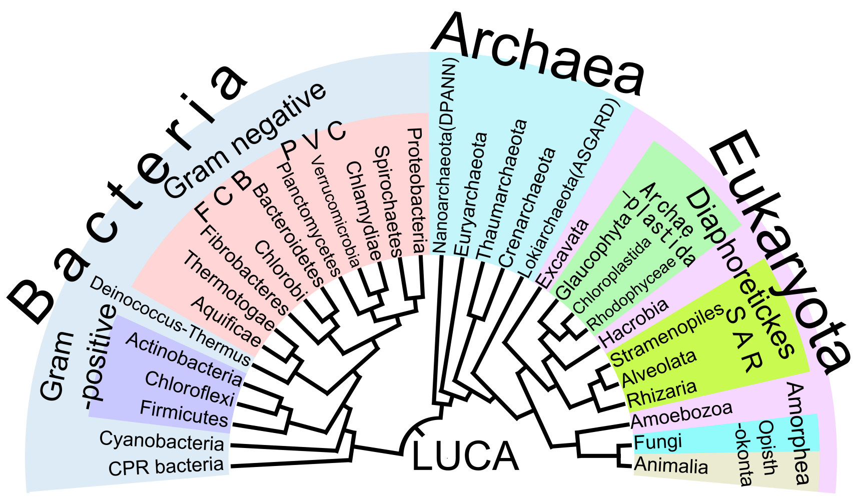


FIGURE 2.1: A phylogenetic tree showing the three life domains: bacteria, archaea, and eukaryota. URL: https://commons.wikimedia.org/wiki/File:Phylogenetic_Tree_of_Life.png

2.2.1 Types Of Trees

Phylogenetic Tree has some properties. First of all, the species represented in it, are called taxa (singular - taxon). A node of the tree can contain more than one taxon and they called sister groups while they have a common ancestor. Moreover, into the tree some nodes might have a common ancestor with a separate sister group, so they called outgroups of the sister group. Next trees are classified as rooted or unrooted trees. A ***Rooted*** phylogenetic tree is a directed tree with a unique node — the root — corresponding to the (usually imputed) most recent common ancestor of all the entities at the leaves of the tree. The root node does not have a parent node but serves as the parent of all other nodes in the tree. The root is therefore a node of degree 2 while other internal nodes have a minimum degree of 3 (where "degree" here refers to the total number of incoming and outgoing edges). The most common method for rooting trees is the use of an uncontroversial outgroup—close enough to allow inference from trait data or molecular sequencing, but far enough to be a clear outgroup. ***Unrooted*** trees 2.2b illustrate the relatedness of the leaf nodes without making assumptions about ancestry. They do not require the ancestral root to be known or inferred. Unrooted trees 2.2a can always be generated from rooted ones by simply omitting the root. By contrast, inferring the root of an unrooted tree requires some means of identifying ancestry. This is normally done by including an outgroup in the input data so that the root is necessarily between the outgroup and the rest of the taxa in the tree, or by introducing additional assumptions about the relative rates of evolution on each branch, such as an application of the molecular clock hypothesis.

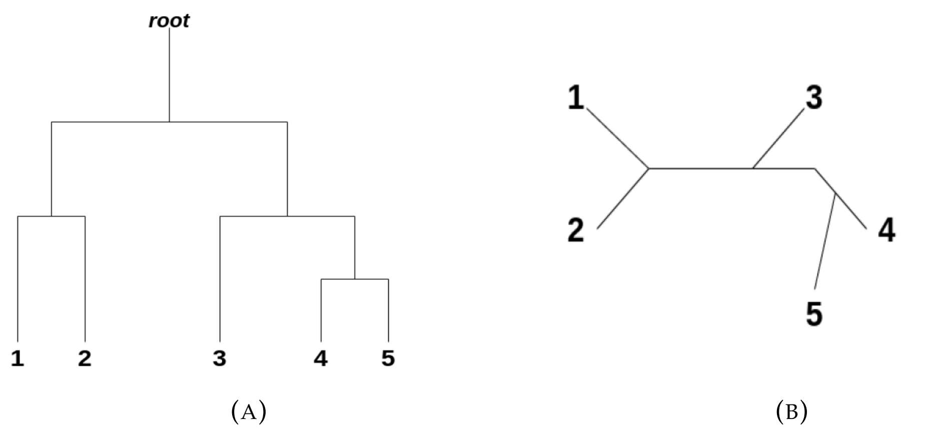


FIGURE 2.2: Sample of (A) a rooted tree and (B) an unrooted tree.

After classifying a phylogenetic tree as a rooted or unrooted tree, we can then ascribe on it some other attributes. Both rooted and unrooted trees can be either ***bifurcating*** or ***multifurcating***. A rooted bifurcating tree has exactly two descendants arising from each interior node (that is, it forms a binary tree), and an unrooted bifurcating tree takes the form of an unrooted binary tree, a free tree with exactly three neighbors at each internal node. In contrast, a rooted multifurcating tree may have more than two children at some nodes and an unrooted multifurcating tree may have more than three neighbors at some nodes. Moreover, both rooted and unrooted trees can be either ***labeled*** or ***unlabeled***. A labeled tree has specific values (names or codes of taxa, level of taxa on the tree hierarchy, frequencies of appearance) assigned to its leaves, while an unlabeled tree, sometimes called a tree shape, defines a topology only.

2.2.2 Enumerating trees

The number of possible trees for a given number of leaf nodes depends on the specific type of tree, but there are always more labeled than unlabeled trees, more multifurcating than bifurcating trees, and more rooted than unrooted trees. The last distinction is the most biologically relevant; it arises because there are many places on an unrooted tree to put the root. For bifurcating labeled trees, the total number of rooted trees is:

$$N_{\text{rooted}} = \frac{(2n-3)!}{2^{n-2}(n-2)!} \tag{2.1}$$

For bifurcating labeled trees, the total number of unrooted trees is:

$$N_{\text{unrooted}} = \frac{(2n-5)!}{2^{n-3}(n-3)!} \tag{2.2}$$

Among labeled bifurcating trees, the number of unrooted trees with **n** leaves is equal to the number of rooted trees with **n-1** leaves. The number of rooted

Labeled leaves	**Binary unrooted trees**	**Binary rooted trees**
1	1	1
2	1	1
3	1	3
4	3	15
5	15	105
6	105	295
7	295	10,395
8	10,395	135,135
9	135,135	2,027,025
10	2,027,025	34,459,425

TABLE 2.1: All possible trees with and without root resulting in species species

trees grows quickly as a function of the number of tips. For 10 tips, there are more than 34×10^6 possible bifurcating trees. The conclusion is that, as the number of species increases, the possible trees increase significantly, making it difficult to identify the authentic phylogenetic tree that contains all known species so far.

2.3 Implementation Of Phylogenies With The Use of Software

Following the principles of phylogenetics and its fields of study, scientists are trying to develop stable algorithms based on mathematical, biological models, and using different types of valid input data they can evaluate and construct the tree of life. Past efforts came up positive and nowadays there is a wide variety of algorithms demonstrating the evolution of species. It is estimated that there are more than 200 algorithms that focus on the evolution of the tree of life, each one uses a different mathematical model. Many of them are available on different websites such as in the Department of Genome Sciences, sector of the Medicine school of University of Washington (`http://evolution.genetics.washington.edu/phylip/software.html`). In this thesis, the using algorithm is RAxML (Randomized Axelerated Maximum Likelihood)[2].

2.4 RAxML

RAxML is a maximum likelihood criterion (ML)-based program and its goal is to identify which tree, out of all possible trees, best estimates the true evolutionary history of the data analyzed. It can use a variety of different character sets (PHYLIP type (.phy) [3]), including nucleotide, amino acids, binary, and multi-state character data.

The first step of the search strategy employed by RAxML is the alignment of input sequences (better previously aligned) and the generation of a starting tree. This starting tree is constructed by adding the sequences one by one in random order, and identifying their optimal location on the tree [4]. The random order in which sequences are added is likely to generate several different starting trees every time a new analysis is run (especially for data sets with more than a few sequences), which allows better exploration of the tree space. Moreover, RAxML optimizes the length of the tree's branches and then calculates the score of the likelihood for each subtree under the ML. The second step of the search strategy involves a method known as lazy subtree rearrangement (LSR) [5]. Briefly, under LSR, all possible subtrees of a tree are clipped and reinserted at all possible locations as long as the number of branches separating the clipped and insertion points is smaller than N branches. RAxML estimates the appropriate N value for a given data set automatically. The LSR method is first applied on the starting tree, and subsequently multiple times on the currently best tree as the search continues until no better tree is found. Finally, after some optimization, all best trees are compared among them and the final best tree is generated.

It is important to be mentioned that the computation load of the maximum likelihood criterion is increased dramatically while the number of species and their length of sequences are increased. This is a problem that still remains even though some newer versions have been published, implemented with SSE, AVX, PThreads, and MPI. In this thesis, there is an effort to optimize this overall load by achieving a significant factor of parallelism of tasks and good management of available resources.

Chapter 3

Related Work

Bioinformatics applications are characterized by immense computational loads and extremely large datasets. This is a problem that arises from the rapid increase of genetic data discovered by scientists, and they vary in many structure states which prevent the results of comparisons from coming out of one differentiation. Compact computing systems used in time, could not surpass a threshold of speedup on those processes, directing scientists to find some other innovation. By the time FPGAs came to the surface and started to be used in frequent and multipurpose problems, bioinformatics turned to using them.

Testing and verifying their tasks and designs on FPGAs, some important speedups are observed. However, FPGAs have not constituted the unique solution. Passing the times, GPUs were upgraded and started to obtain a standalone functionality, which was specialized on the task level parallelism. So, using these technologies some problems started to get solved. The following sections describe some of the basic problems that are met on bioinformatics and especially on the algorithms that are used for implementing their surveys.

3.1 Sequence Comparison

One basic problem that was resolved is the Sequence Comparison on some bioinformatics algorithms. Sequence comparison algorithms compute the degree of matching between two or more biomolecular sequences. Biologists may use sequence comparison results either as a proxy to infer sequence homology or as part of larger computational pipelines. One basic algorithm used for this purpose is BLAST (Basic Local Alignment Search Tool) [6], used to find similarities between genetic sequences (queries) and sequence databases. Dollas et al. [7] [8] [9], has proposed several different architectures and implementations of standalone FPGA-based platforms for BLAST achieving a great speedup of execution time. In addition to the TUC architectures, other research groups have looked into the BLAST algorithm [10] [11] [12] [13]. Another similar algorithm with BLAST is CAST [14] an iterative algorithm for the complexity analysis of sequence tracts. In these algorithms, the most important factor that affects execution speed and the quality of their results is the occurrence of low complexity regions (LCRs) in protein sequences. So, to reduce it, there was an FPGA-based approach of CAST by [15] where the execution time was accelerated by exploiting some inherent parallel characteristics.

3.2 Multiple Sequence Alignment

Apart from sequence comparison algorithms, there are some for Multiple Sequence Alignment (MSA) which bring scientists in trouble. Some of them are MAAFT[16] and T-Coffee[17] algorithms which are progressive MSA methods based on the Fast Fourier Transform (FFT). Their phase of sequence alignment takes up to 80% of the total execution time. So, Lakka et al. [18] proposed an FPGA-based IP core for each algorithm that implements this phase and achieves speedups 10x to 50x faster than sequential execution for MAAFT and 1× to 10× for T-Coffee.

3.3 Prediction of RNA and protein secondary structure

Prediction of RNA and protein secondary structure is of great importance in Medicine and Biology as it may highlight structural and functional properties of molecules. Analyzing the algorithms correlates with this procedure, the weak spots came to the surface, such as the excessive execution time and the huge search space. So, for some algorithms such as Zuker [19] and Predator [20] , Smerdis et al. [21] [22] designed FPGA-based IP blocks to accelerate and eliminate these spots, achieving a speedup of 3x up to 10x for Zuker and 30x up to 50x for Predator.

3.4 Gene identification

Another issue that is of concern to scientists is Gene identification. It is one of the most important steps in the process of understanding the information contained in (complete) genome sequences. The gene prediction problem refers to the identification of biologically functional stretches of sequences (genes) in genomic DNA. An algorithm used for this purpose, Glimmer [23] , got under analysis and profiling by Chrysos et al. [24] who proposed a reconfigurable architecture implementing this algorithm. The result was execution speedups from 1.14x up to 2.37x when it was compared with the official software implementation on a general-purpose PC.

3.5 Phylogenetic Trees

In previous Chapter 2, the definition and some principles of phylogeny were described. It was also shown that increasing the number of species (data), the number of trees under comparison reaches major values. For this purpose, the survey on phylogenetic trees is a significant burden on scientists' shoulders. To reach a solution, scientists focus on Phylogenetic Likelihood Function (PLF) which is the most widely used optimality criterion to score and, thus, choose among distinct evolutionary scenarios (phylogenetic trees). The PLF is used by many program packages like RAxML[2], GARLI[25], MrBayes[26], PAML[27], and PAUP[28].

A series of reports for the optimization of PLF, made by Alachiotis et al. [29] [30]. The more recent work was an optimized reconfigurable system for computing the PLF on DNA data that extends its usage in RAxML algorithm. The main goal of this work was to implement some pipeline stages to operate each different state of data. Next, integrating a scaling unit and designing proper host-side management, it reached a finished system. The targeted platforms were Virtex 6 SX475T-2 FPGA and Virtex 5 SX95T FPGA. Overall, an improvement of approximately 57% was achieved, comparing the execution on FPGA and the execution on PC with the version of AVX intrinsics. In addition, one similar work was conducted by Alachiotis et al. [31] and was compared with the OpenMP version of RAxML

As PLF is not only specialized in RAxML, Stephanie Zierke and Jason D

Bakos [32], presented an FPGA acceleration of PLF for Bayesian MCMC inference methods, using the MrBayes 3 tool as a framework for designing the co-processor. MrBayes uses the PLF to evaluate the likelihood of trees (which consumes nearly all of the execution time) and uses the Metropolis-coupled Markov chain Monte Carlo (MCMC) search to move through the tree space. In this application, three components are designed which are involved in computing the log-likelihood of a tree. Each step depends on the previous so they must be performed sequentially but can be parallelized using a single deep pipeline. So, connecting them properly and reducing I/O times, a range of 4.7 to 8.7 speedup vs software execution time was achieved (speedup based on datasets).

Another technology used for accelerating PLF or part of it, is Intel Many Integrated Core (MIC). Kozlov et al. [33] describe an optimized implementation of the PLF kernel for the novel Intel Many Integrated Core (MIC) architectures. Using a MIC-based accelerator (Xeon Phi 5110P), they achieved speedups ranging from 1.9x to 2.8x for different PLF kernels, compared to a highly optimized AVX implementation running on a dual socket Xeon E5-2680 system. By integrating the optimized PLF into the phylogenetic inference program RAxML-Light [34], the overall execution times were reduced by up to a factor of two. To assess the scalability on multiple Xeon Phi cards, a hybrid MPI-OpenMP version of the ExaML [35] code was developed. When ExaML is executed on two coprocessors on the same node, we obtain speedups of up to a factor of 3.7 (vs. a CPU baseline) and 1.8 (vs. a single MIC). As expected, speedups increase with growing dataset size and become stable for alignments that require processing of 1-2 million sites per MIC card.

The technologies used on surveys on phylogenetic trees are not restricted only on FPGA-based platforms and acceleration cards but also extend to GPUs, a rapidly evolving hardware component. Fernando Izquierdo-Carrasco, Nikolaos Alachiotis and Simon Berger [4] developed a generic Vectorization Scheme and a GPU kernel for Phylogenetic Likelihood Library (see Chapter 4). They presented a novel scheme for vectorizing computations and organizing conditional likelihood arrays (CLAs) in such a way that they do not need to be transferred at all between the GPU and the CPU. There was also a comparison between the performance of the GPU implementation for DNA data with a highly optimized x86 version of the PLL that relies on manually tuned AVX intrinsics into the RAxML-Light. The metrics for the calculations sprang by timekeeping floating-point computations of which the amount increased with the squared number of states (a range of 1.024 through 262.144). That GPU implementation accelerated the likelihood computations by a factor of two compared to the currently fastest available x86 implementation.

3.6 Thesis Approach

This thesis aims to develop accelerators for Phylogenetic Likelihood Library used in RAxML. The difference between this work and the previous ones is that it focuses on the usage of PLF with protein sequence input datasets while all the previous ones focused on DNA sequence datasets whose structure is more simple and accurate. Cumulatively, all works referred to on this chapter are shown in 3.1, while the bold one is our work. The final calculation of PLF, with the protein sequences as input, requires 5 times as many stages (it is the difference between nucleotides and amino acids) and as a sequence requires more operations and computation time. This is the stimulus and the goal to find the spots that slow down the total execution time and bind the majority of available resources. Achieving it, the next step is to design accelerators for them. These hardware accelerators are routed and placed onto a reconfigurable computing platform (FPGA) and then on a cloud-based reconfigurable computing platform (FPGA). The purpose of these two implementations is that users that want to use the whole algorithm for the construction of a tree of life, can even have quicker results using a recommended or individual FPGA or executing it remotely on the cloud-based FPGA.

Technology	Algorithms	Data
FPGA	BLAST[6]	DNA
	CAST[14]	Proteins
	MAAFT[16]	DNA
	T-Coffee[17]	DNA
	Zuker[19]	Proteins, RNA
	Predator[20]	Proteins, RNA
	Glimmer[23]	Genes
	Bayesian[26] (PLF)	DNA
	RAxML[2] (PLF)	DNA
	RAxML (PLF)	**Proteins**
Intel MIC	RAxML-Light[34] (PLF)	DNA
	EXaML[35] (PLF)	DNA
GPU	RAxML-Light (PLF)	DNA
	EXaML (PLF)	DNA

TABLE 3.1: Accelerated Algorithms on different technologies according to their processing Data

Chapter 4

Maximum Likelihood Method

In this chapter, the Maximum Likelihood Method is described and also how this method can be computed. There is also a reference on some amino acid models used in RAxML and consecutively for the execution of ML. In addition, there is a brief description of the functions of PLL (Phylogenetic Likelihood Library) that implements the ML.

4.1 Maximum Likelihood

Maximum likelihood is a general statistical method for estimating unknown parameters of a probability model. A parameter is descriptor of the model. A familiar model might be the normal distribution of a population with two parameters: the mean and variance. In phylogenetics, there are many parameters, including rates, differential transformation costs, and, most importantly, the tree itself. The Likelihood is defined to be a quantity proportional to the probability of observing the data given the model, P(D | M). Thus, if we have a model (i.e. the tree and parameters), we can calculate the probability of the observations that would have actually been observed as a function of the model. We then examine this likelihood function to see where it is greatest, and the value of the parameter of interests (usually the tree and/or branch lengths) at that point is the maximum likelihood estimate of the parameter. In this case, Maximum Likelihood can be used as an optimality measure for choosing a preferred tree or set of trees. It evaluates a hypothesis (branching pattern), which is a proposed evolutionary history, in terms of the probability that the implemented model and the hypothesized history would have given rise to the observed data set. Essentially, a pattern that has a higher probability is preferred to one with a lower probability.

4.2 Models of Amino Acid Replacement

In this thesis, all the models used for the study of ML, assume that all amino acid sites in an alignment evolve independently and according to the same Markov process [36]. This means they are stationary and homogeneous so that the amino acid frequencies and the model of evolution are assumed constant through time and across all sites in an alignment. Additionally, they have the property to be time-reversible, implying that their evolution of going forward and backward in time, is the same.

The probability of amino acid *i* being replaced by amino acid *j* over time *T* is P_{ij}(T), where *i* and *j* take the values 1, 2, . . . , 20, representing the 20 different amino acids, as shown on 4.1. These probabilities can be written as 20×20 matrices, P(T), called replacement matrices, and are essential in most methods to infer protein phylogenies [37]. These matrices are expected to capture the biological and physicochemical properties of amino acids. They are used in distance-based methods to estimate the evolutionary distance—the expected number of substitutions per site—between sequence pairs. In the maximum likelihood (ML) method, they are used to compute substitution probabilities along tree branches and hence the likelihood of the data. These matrices are calculated as $P(T) = exp(TQ)$, where Q is the rate matrix, with off-diagonal elements Q_{ij} being the instantaneous rates of change of amino acid *i* to amino acid *j* and with diagonal elements Q_{ii} being fixed so that the row sums of Q equal 0. The off-diagonal elements of the matrix Q can be described by the off-diagonal elements of the matrix product

$$\begin{pmatrix} — & s_{1,2} & s_{1,3} & \cdots & s_{1,20} \\ s_{1,2} & — & s_{2,3} & \cdots & s_{2,20} \\ s_{1,3} & s_{2,3} & — & \cdots & s_{3,20} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ s_{1,20} & s_{2,20} & s_{3,20} & \cdots & — \end{pmatrix} \cdot \mathrm{diag}(\pi_1, \ldots, \pi_{20})$$

so Q can be defined by two sets of components, s_{ij} and π_i. The variables s_{ij} represent the exchange abilities of amino acid pairs *(i, j)* and the π_i values represent the equilibrium or stationary frequencies of the 20 amino acids. These frequencies may all be set to values estimated from the original data used to estimate the s_{ij}, but as these applications are currently rare in phylogenetics, these frequencies derive empirically from phylogenetic analysis of several similar data sets and are assumed to be constant. It must be mentioned that there are many different models of amino acid sequences such as JTT, DAYHOFF, WAG, LG, MTREV, RTREV, CPREV and all of them have their own frequencies. For this purpose, the referred models have been tested and there was no difference in execution times or implementations, so we randomly selected all data sets to follow the WAG model and their frequencies, and the probabilities of amino acids are shown in the next table4.1.

Amino Acids Frequencies		
Amino Acid	Letter Code	Frequency
Alanine	Ala - A	0.0866279
Arginine	Arg - R	0.043972
Asparagine	Asn - N	0.0390894
Aspartic acid	Asp - D	0.0570451
Cysteine	Cys - C	0.0193078
Glutamine	Gln - Q	0.0367281
Glutamic acid	Glu - E	0.0580589
Glycine	Gly - G	0.0832518
Histidine	His - H	0.0244313
Isoleucine	Ile - I	0.048466
Leucine	Leu - L	0.086209
Lysine	Lys - K	0.0620286
Methionine	Met - M	0.0195027
Phenylalanine	Phe - F	0.0384319
Proline	Pro - P	0.0457631
Serine	Ser - S	0.0695179
Threonine	Thr - T	0.0610127
Tryptophan	Trp - W	0.0143859
Tyrosine	Tyr - Y	0.0352742
Valine	Val - V	0.0708956

TABLE 4.1: Amino acid frequencies (P_i). Source : `https://www.ebi.ac.uk/goldman-srv/WAG/wag.dat`

4.3 Calculating Tree's Probability

Following the Maximum Likelihood method, the probability of the tree can be estimated by calculating a score for each assumption made for all data. This method follows Felsenstein's Pruning Algorithm (FPA) [38] which offers a practical and rapid calculation of the maximum likelihood. This algorithm proceeds by removing one pair of amino acids at a time from the tree, leaving behind a single fictional amino acid population each time. After p-1 such removals, we have p-1 independent trees. Their joint likelihood is equal to the likelihood of the original tree. To analyze and understand it, we start with a simple example.

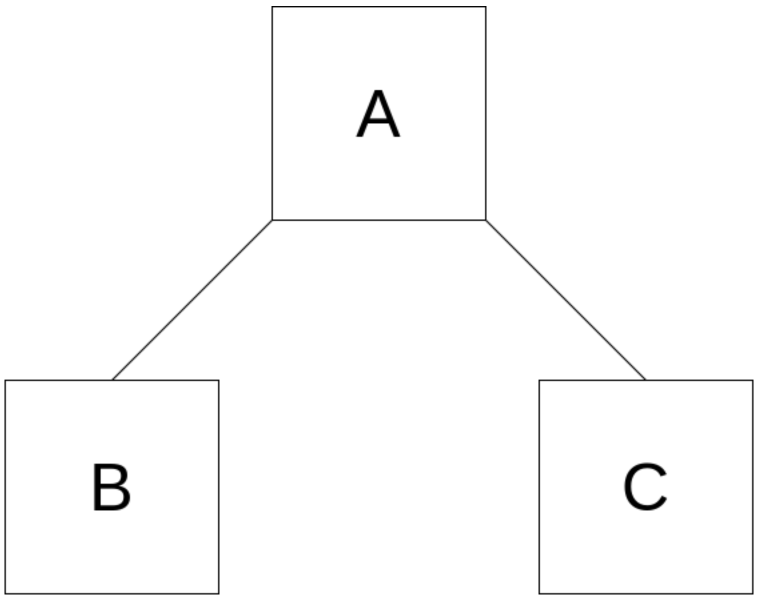


FIGURE 4.1: A-B-C tree

For each site N of Amino Acid sequence, there is a supposed tree with root A and three vectors A, B, C , on figure 4.1. ML using the FPA tries to estimate the reliability of the tree according to the given data.

According to the figure, it assumed that B and C have A as common ancestor. In order to find the accuracy of this case, the likelihood of root A must be calculated. For each site N of B and C, the likelihood of site A must be calculated under the next equation.

$$L\left(X_{Aj}=i\right)=\left[\sum_z P_{iz}\left(S_{AB}\right)L\left(X_{Bj}=z\right)\right]\left[\sum_z P_{iz}\left(S_{AC}\right)L\left(X_{Cj}=z\right)\right] \tag{4.1}$$

In equation 4.1 the variable X represents an amino acid sequence as a vector, variable j represents the site of sequence X and the variables i,z represent the states of twenty amino acids. P symbolises the replacement matrix of transitions of state i to z one for a given length S. L symbolises the likelihood of sequence X sited on j, for all z and i states. If a sequence X is known , then the likelihood is equal to 1, under the condition that state j and z are similar. Contrariwise, the likelihood is equal to 0. Each sequence of A,B,C split into $N * 80$ vectors for each site N of amino acid sequence. Each separate vector consists of 4 sites and each of them contains 20 probabilities corresponding to the 20 amino acids. Finally, the probabilities of N vectors frames a 4×20 matrix for each site of N sequences.

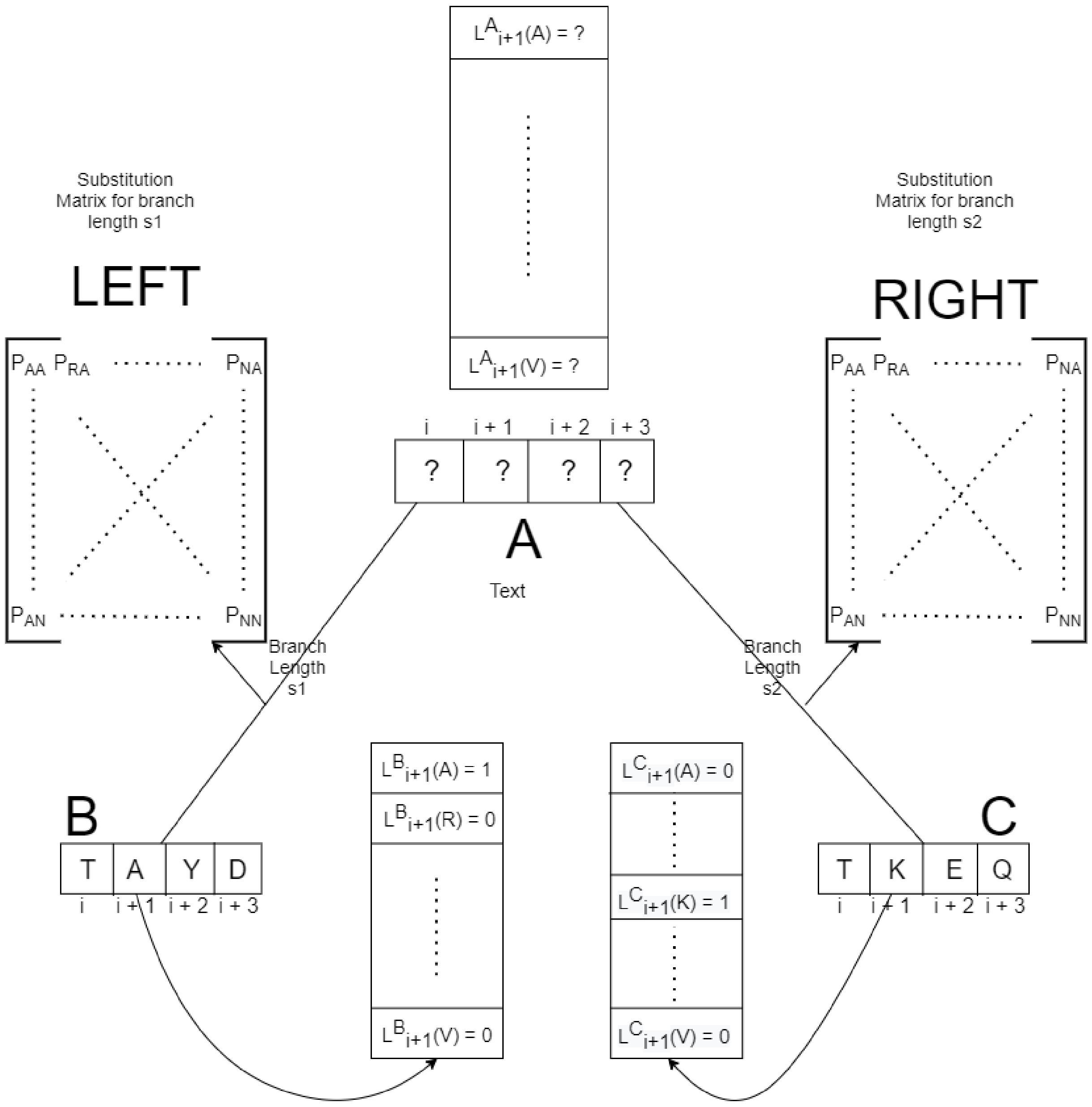


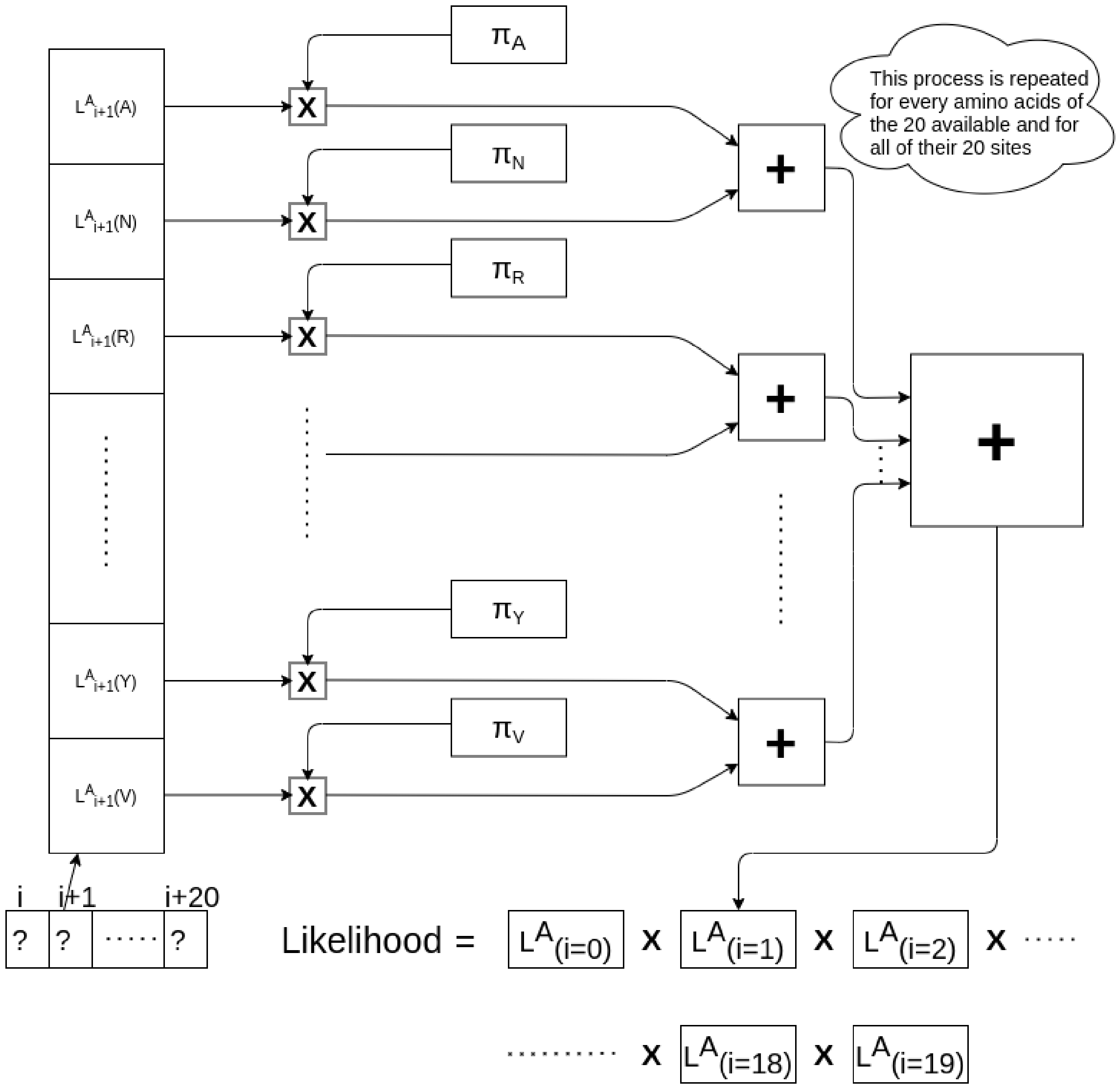
FIGURE 4.2: Calculation of sequence A, assumed being ancestor of B and C

In the above figure, it was randomly selected site *i+1* of A, B and C likelihood matrices. Each one contains 20 sites corresponding to the 20 different amino acids. In the case of B, the site of *i+1* corresponds to the amino acid A (Alanine) and that is the reason for having the value of 1 while the remaining 19 amino acids have the value of 0. In the same way, matrix C has on its *i+1* site the amino acid K (Lysine) and it is the only one with value 1. The substitution matrix of the B sequence is called Left and contains all probabilities of transformation of the sites of B, for a given branch length *s1*. On the other side, this matrix is called Right and matches to the sites of C, for a given branch length *s2*. So , according to the equation 4.1, the likelihood of site *i+1* is calculated as :

$$
\begin{aligned}
L^{A}_{i+1}(\mathrm{A}) = &\Big[\mathrm{P_{AA}(s1)L^{B}_{i+1}(A) + P_{AR}(s1)L^{B}_{i+1}(R) + \ldots + P_{AY}(s1)L^{B}_{i+1}(Y) + P_{AV}(s1)L^{B}_{i+1}(V)}\Big] \\
\mathrm{x} &\Big[\mathrm{P_{AA}(s2)L^{C}_{i+1}(A) + P_{AR}(s2)L^{C}_{i+1}(R) + \ldots + P_{AY}(s2)L^{C}_{i+1}(Y) + P_{AV}(s2)L^{C}_{i+1}(V)}\Big] \\
L^{A}_{i+1}(\mathrm{R}) = &\Big[\mathrm{P_{RA}(s1)L^{B}_{i+1}(A) + P_{RR}(s1)L^{B}_{i+1}(R) + \ldots + P_{RY}(s1)L^{B}_{i+1}(Y) + P_{RV}(s1)L^{B}_{i+1}(V)}\Big] \\
\mathrm{x} &\Big[\mathrm{P_{RA}(s2)L^{C}_{i+1}(A) + P_{RR}(s2)L^{C}_{i+1}(R) + \ldots + P_{RY}(s2)L^{C}_{i+1}(Y) + P_{RV}(s2)L^{C}_{i+1}(V)}\Big] \\
&\vdots \\
&\vdots \\
L^{A}_{i+1}(\mathrm{Y}) = &\Big[\mathrm{P_{YA}(s1)L^{B}_{i+1}(A) + P_{YR}(s1)L^{B}_{i+1}(R) + \ldots + P_{YY}(s1)L^{B}_{i+1}(Y) + P_{YV}(s1)L^{B}_{i+1}(V)}\Big] \\
\mathrm{x} &\Big[\mathrm{P_{YA}(s2)L^{C}_{i+1}(A) + P_{YR}(s2)L^{C}_{i+1}(R) + \ldots + P_{YY}(s2)L^{C}_{i+1}(Y) + P_{YV}(s2)L^{C}_{i+1}(V)}\Big] \\
L^{A}_{i+1}(\mathrm{V}) = &\Big[\mathrm{P_{VA}(s1)L^{B}_{i+1}(A) + P_{VR}(s1)L^{B}_{i+1}(R) + \ldots + P_{VY}(s1)L^{B}_{i+1}(Y) + P_{VV}(s1)L^{B}_{i+1}(V)}\Big] \\
\mathrm{x} &\Big[\mathrm{P_{VA}(s2)L^{C}_{i+1}(A) + P_{VR}(s2)L^{C}_{i+1}(R) + \ldots + P_{VY}(s2)L^{C}_{i+1}(Y) + P_{VV}(s2)L^{C}_{i+1}(V)}\Big]
\end{aligned}
\tag{4.2}
$$

After these calculations, the likelihoods of the common ancestor A are multiplied with the base frequencies π of the 20 different amino acids. Finally, the total probability of the tree is computed by multiplying the probabilities of its i sites of common ancestor A between them. The mathematical equation of this process follows as :

$$L(i) = \sum_j \pi_j * L\left(X_i^A = j\right), \text{ where } j = A, R, N, \cdots Y, V \tag{4.3}$$



π_A
π_N
π_R
π_Y
π_V
$L^A_{i+1}(A)$
$L^A_{i+1}(N)$
$L^A_{i+1}(R)$
$L^A_{i+1}(Y)$
$L^A_{i+1}(V)$
X
+
This process is repeated for every amino acids of the 20 available and for all of their 20 sites
i
i+1
i+20
?
?
·····
?
Likelihood = $L^A_{(i=0)}$ x $L^A_{(i=1)}$ x $L^A_{(i=2)}$ x ·····
·········· x $L^A_{(i=18)}$ x $L^A_{(i=19)}$

FIGURE 4.3: Calculation of final likelihood of common ancestor A

As it is shown in the above figure the final likelihood is calculated by multiplication of all 20 likelihoods of i sites and it is estimated by the equation:

$$L = \prod_i L(i) \tag{4.4}$$

However, since all entries p_ij of the substitution probability matrix are ≤ 1, the individual L 4.2 values and the L(i) can be very small, a fact that may bring on arithmetic overflows. To cope with this problem, a logarithmic approach, like Michael Otts implementation [39] on the case of nucleotides, is used instead of the multiplication of equation 4.4. So the last step of the Likelihood calculation of common ancestor A can be referred as :

$$L = \sum_i \log(L(i)) \tag{4.5}$$

4.4 Adjustments on ML for RAxML

As it was shown in the previous sections, the Left and Right matrices are 20×20. However, for every different estimation of a 3-node tree, these matrices tend to change due to their heterogeneity. This practically means that every vector has different frequencies for the substitution of their amino acids during an interval dt. In detail, every vector N differs on the substitution frequencies of their amino acids from the rest vectors, ending on vast data usage. Due to this vast volume, Yang suggested a model which is called the Discrete Γ Model. Discrete Γ Model uses a new substitution matrix with $20 \times 20 \times 4$ dimensions, for the whole length N of all sequences. These matrices were generated to accumulate uniformly all different substitution matrices of all i sites of N vectors. In this thesis, the Discrete Γ Model is used and combined with the GTR model through RAxML, results in the GTR-GAMMA model [40] [41].

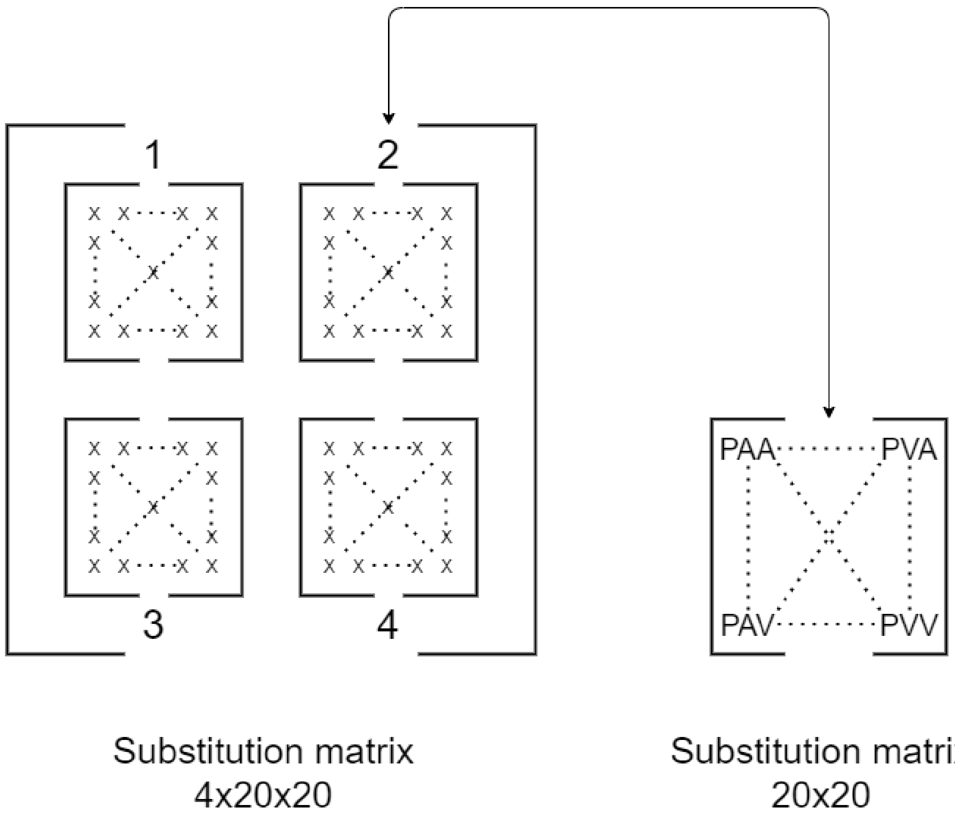


FIGURE 4.4: Substitution Matrix $4 \times 20 \times 20$ of Discrete Γ Model. On the single 20×20 matrix P symbolizes the probability and the other two letters the correlation of all amino acids following the table 4.1

According to the above figure, the new Γ substitution matrix consists of 4 substitution matrices of 20×20 for a given branch length. Integrating this new matrix, results in a significant reduction of data volume under process, without any increase in operation volume. To expand, each N vector is not represented by each own substitution matrix but by a unified substitution matrix for all N vectors. This conclusion derives from the following equation:

$$L\left(X_{Aj} = i\right) = \left[\sum_z P_L k_{iz}\left(S_{AB}\right) L\left(X_{Bj} = z\right)\right]\left[\sum_z P_R k_{iz}\left(S_{AC}\right) L\left(X_{Cj} = z\right)\right] \tag{4.6}$$

Comparing equation 4.6 and 4.1, there are quite similar. The only difference is that, for each i site of A under its likelihood computation, the likelihoods of B and C sequences are multiplied by different substitution matrices k.

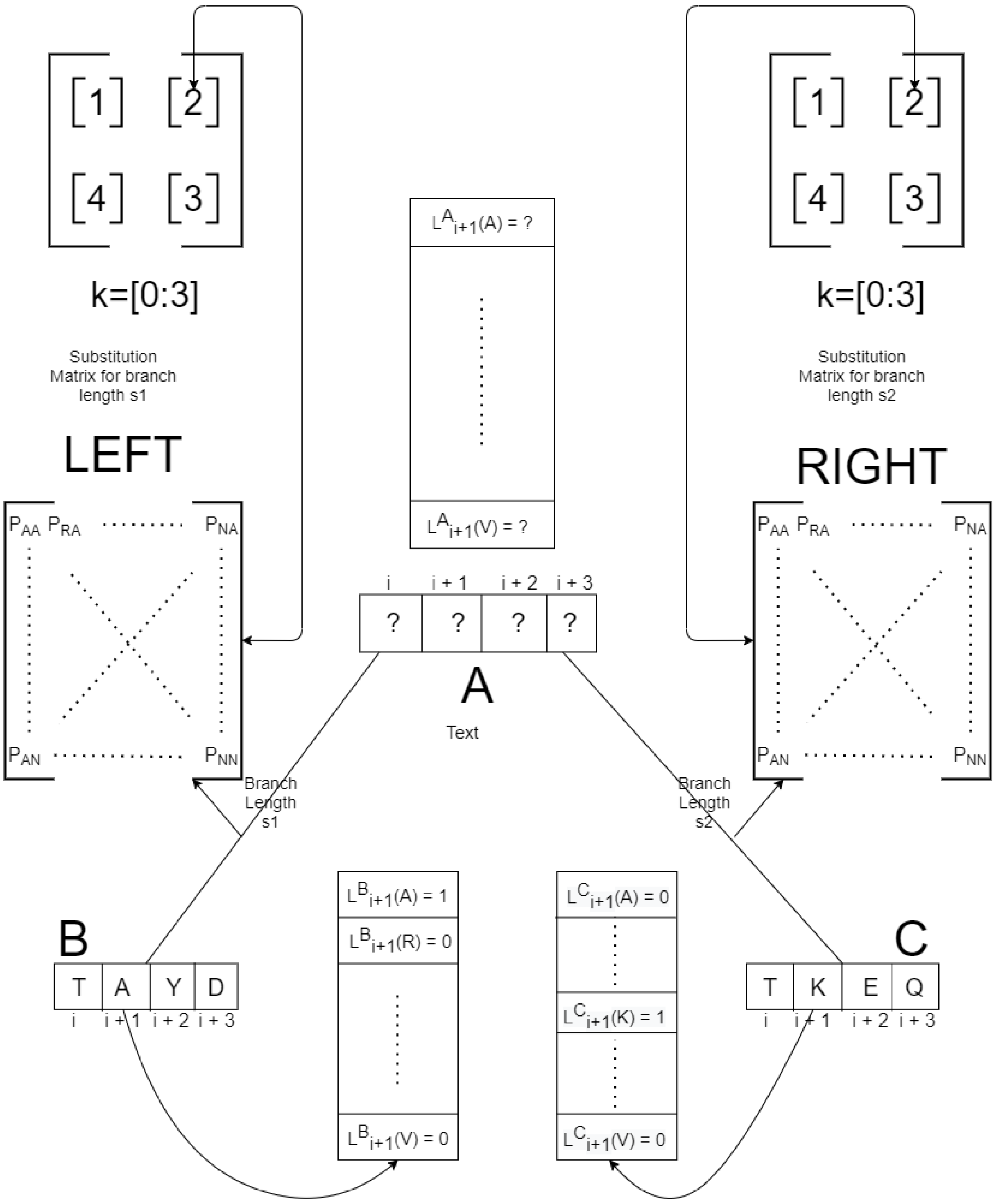


FIGURE 4.5: Calculation of sequence A, assumed being ancestor of B and C under RAxML modulations

On the above figure, different substitution matrices are used, $P_{Left}k(s1)$ and $P_{Right}k(s1)$ for every twenty likelihoods of i sites of B and C. Following equation 4.6 and with the use of new Left and Right matrices, the final operations for the computation of maximum likelihood can be derived by:

$$
\begin{aligned}
\mathrm{L_{i+1}(A)} &= \Big[\mathrm{P_L2_{AA}(s1)L^B_{i+1}(A) + P_L2_{AR}(s1)L^B_{i+1}(R) + \ldots + P_L2_{AY}(s1)L^B_{i+1}(Y) + P_L2_{AV}(s1)L^B_{i+1}(V)}\Big] \\
&\quad \mathrm{x} \Big[\mathrm{P_R2_{AA}(s2)L^C_{i+1}(A) + P_R2_{AR}(s2)L^C_{i+1}(R) + \ldots + P_R2_{AY}(s2)L^C_{i+1}(Y) + P_R2_{AV}(s2)L^C_{i+1}(V)}\Big] \\
\mathrm{L_{i+1}(R)} &= \Big[\mathrm{P_L2_{RA}(s1)L^B_{i+1}(A) + P_L2_{RR}(s1)L^B_{i+1}(R) + \ldots + P_L2_{RY}(s1)L^B_{i+1}(Y) + P_L2_{RV}(s1)L^B_{i+1}(V)}\Big] \\
&\quad \mathrm{x} \Big[\mathrm{P_R2_{RA}(s2)L^C_{i+1}(A) + P_R2_{RR}(s2)L^C_{i+1}(R) + \ldots + P_R2_{RY}(s2)L^C_{i+1}(Y) + P_R2_{RV}(s2)L^C_{i+1}(V)}\Big] \\
&\vdots \\
&\vdots \\
\mathrm{L_{i+1}(Y)} &= \Big[\mathrm{P_L2_{YA}(s1)L^B_{i+1}(A) + P_L2_{YR}(s1)L^B_{i+1}(R) + \ldots + P_L2_{YY}(s1)L^B_{i+1}(Y) + P_L2_{YV}(s1)L^B_{i+1}(V)}\Big] \\
&\quad \mathrm{x} \Big[\mathrm{P_R2_{YA}(s2)L^C_{i+1}(A) + P_R2_{YR}(s2)L^C_{i+1}(R) + \ldots + P_R2_{YY}(s2)L^C_{i+1}(Y) + P_R2_{YV}(s2)L^C_{i+1}(V)}\Big] \\
\mathrm{L_{i+1}(V)} &= \Big[\mathrm{P_L2_{VA}(s1)L^B_{i+1}(A) + P_L2_{VR}(s1)L^B_{i+1}(R) + \ldots + P_L2_{VY}(s1)L^B_{i+1}(Y) + P_L2_{VV}(s1)L^B_{i+1}(V)}\Big] \\
&\quad \mathrm{x} \Big[\mathrm{P_R2_{VA}(s2)L^C_{i+1}(A) + P_R2_{VR}(s2)L^C_{i+1}(R) + \ldots + P_R2_{VY}(s2)L^C_{i+1}(Y) + P_R2_{VV}(s2)L^C_{i+1}(V)}\Big]
\end{aligned}
\tag{4.7}
$$

It is noticed that a different substitution matrix between k=[1:4] corresponds to each i site of B and C sequences. In detail, during the study on i site of B and C, the matrix P_{L1} and P_{R1} are used. Forwarding to the i+1 site, P_{L2} and P_{R2} are used, and so on.

In addition, after computing all likelihoods of site i, RAxML uses another 20×20 matrix in combination with likelihoods for some further calculations. It is called Eigen Vector (EV) and as its initials indicate, it is the eigenvectors of the relationship P(s)=e^{Qs}. The reason why RAxML performs these additional operations on the GTR-GAMMA model is to simplify the calculations at the root of tree A as well as the duration of optimizing the length of the tree branches [42]. Practically, the process which uses EV is a multiplication followed by the addition of the twenty likelihoods of i site with the whole EV. The mathematical equation is shown below:

$$L\left(X_{Aj} = i\right) = \sum_K \sum_z EV_K(z) L\left(X_{Aj} = K\right) \tag{4.8}$$

Parameter K symbolizes the twenty likelihoods of one i site of the vector A and consequentially twenty eigenvectors of matrix EV. As it is distinguished in equation 4.8 each likelihood of position i of the vector A is multiplied by twenty of eigenvectors in the EV array and then the result is added cumulatively, creating the final probabilities of the vector A. More specifically, the table of EV vectors is as follows:

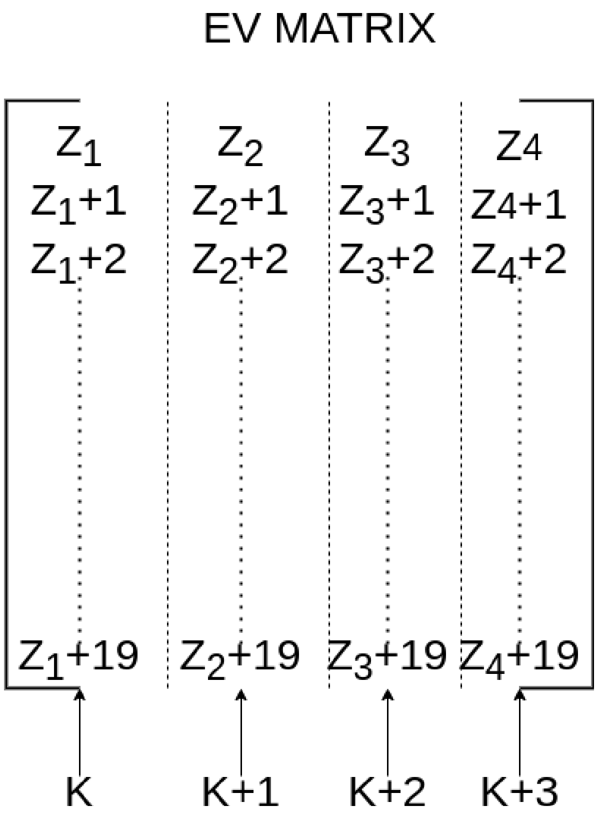


FIGURE 4.6: EigenVectors EV Matrix

Looking at the above figure, it is obvious that each twenty elements of Z_1, Z_2, Z_3, and Z_4 of the EV matrix correspond to a different value of k. Below, it is schematically described:

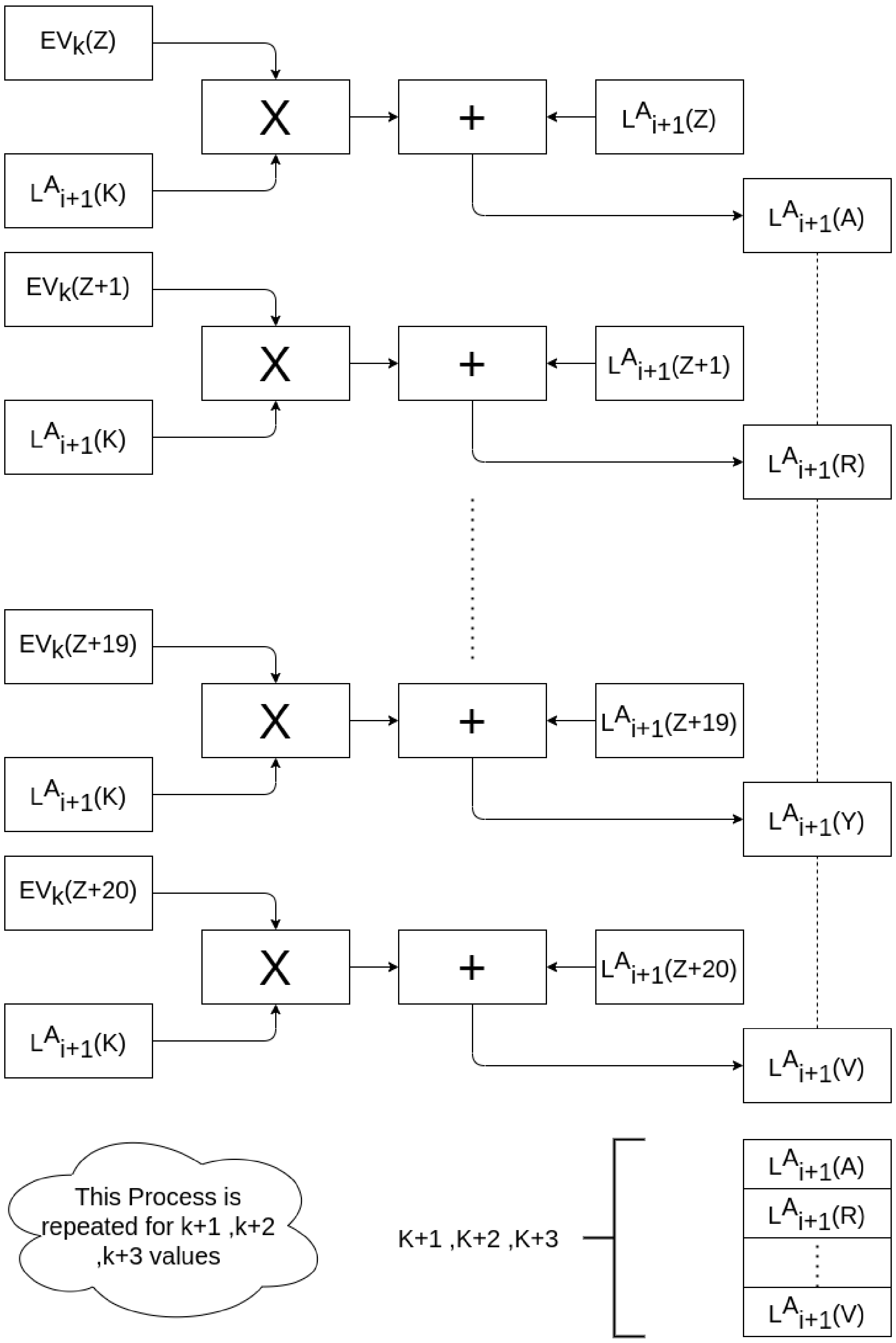


FIGURE 4.7: Final operations between Eigenvectors and Likelihoods

To the end of the above processes, remains the last step, a scaling process which follows the 4.5 to avoid arithmetic overflows and get more accurate results.

4.5 Phylogenetic Likelihood Library

The Phylogenetic Likelihood Library (PLL) is a parallelized and highly optimized software library derived from RAxML. The current PLL code comprises implementations of state-of-the-art algorithms and data structures along with low-level technical and hardware-dependent optimizations. The library can calculate (and optimize) the likelihood on a phylogenetic tree for a plethora of statistical models and data types. In this case, the data type is amino acids (AA) and there is the usage of the GTR-GAMMA model as it was previously mentioned. For computing the likelihood on a tree and for optimizing some requisite and useful parameters, the below core functions are needed:

• The **newview()** or PLF (phylogenetic likelihood) function updates a conditional likelihood vector given two child nodes and given two transition probability matrices P for the respective branch lengths leading to these child nodes. It is also the main computation core for the ML as described in previous subsections.
• The **evaluate()** function is called at the virtual root that has been placed into the unrooted tree for scoring it. Given the two conditional likelihood arrays at either end of the rooted branch and the branch length, this function evaluates and computes the overall log-likelihood of the tree.
• The **coreDerivative()** function computes the first and second derivative of the likelihood function at a given branch.
• The **sumGAMMA()** function pre-computes the element-wise product of the ancestral probability vectors to the left and the right of the branch under optimization. This product is then re-used repeatedly by iterations of coreDerivative() and allows to save time by avoiding redundant computations.

Chapter 5

Hardware Architecture Of Accelerator

The Fifth chapter refers to some features of the profiling made for PLL functions and why it urged us to focus on some of them. Moreover, it describes in detail the architecture of the accelerators and the framework on which are mapped. The next sub sections analyze the whole implementation of the kernels which contains the fetch units of input-output data, the processing units, and the communication from the host to kernels and vice versa.

5.1 Profiling of PLL

As this work is directed to improve the execution times of PLL functions, it was necessary to delve deeper and find the most time-consuming parts of PLL. The profiling of RAxML was implemented on the PLL functions of the algorithm, considering both the sequential version and the AVX versions of it. The used tools for the profiling were some manual timers and counters into the requisite functions and a Linux-based tool *callgrind* which was able to give us details about timings in an overall framework.

Between the functions of PLL, two of them were found that consume the biggest percentage of total execution time, Phylogenetic Likelihood Function (PLF or NewViewGTRGAMMAPROT) and SumGAMMAPROT. In the following pie charts 5.1 and 5.2, the results of the profiling are depicted. It must be referred that these functions consist of three cases. The first one is when input data derive from 2 nodes-leaves, the second one is when data come from a node-leave and an inner node, and the third one is when data come from 2 inner nodes. Moreover, the following pie charts correspond to the sum of all total calls of these functions.

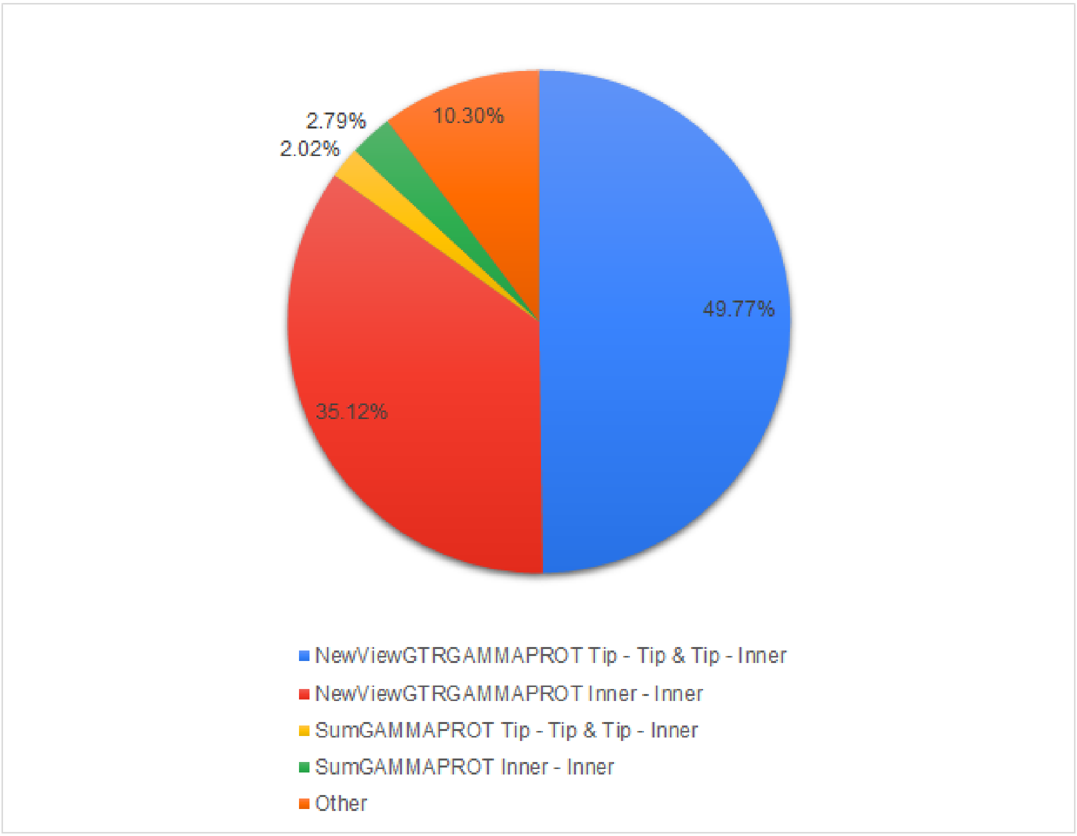


FIGURE 5.1: Min percentage-PLF

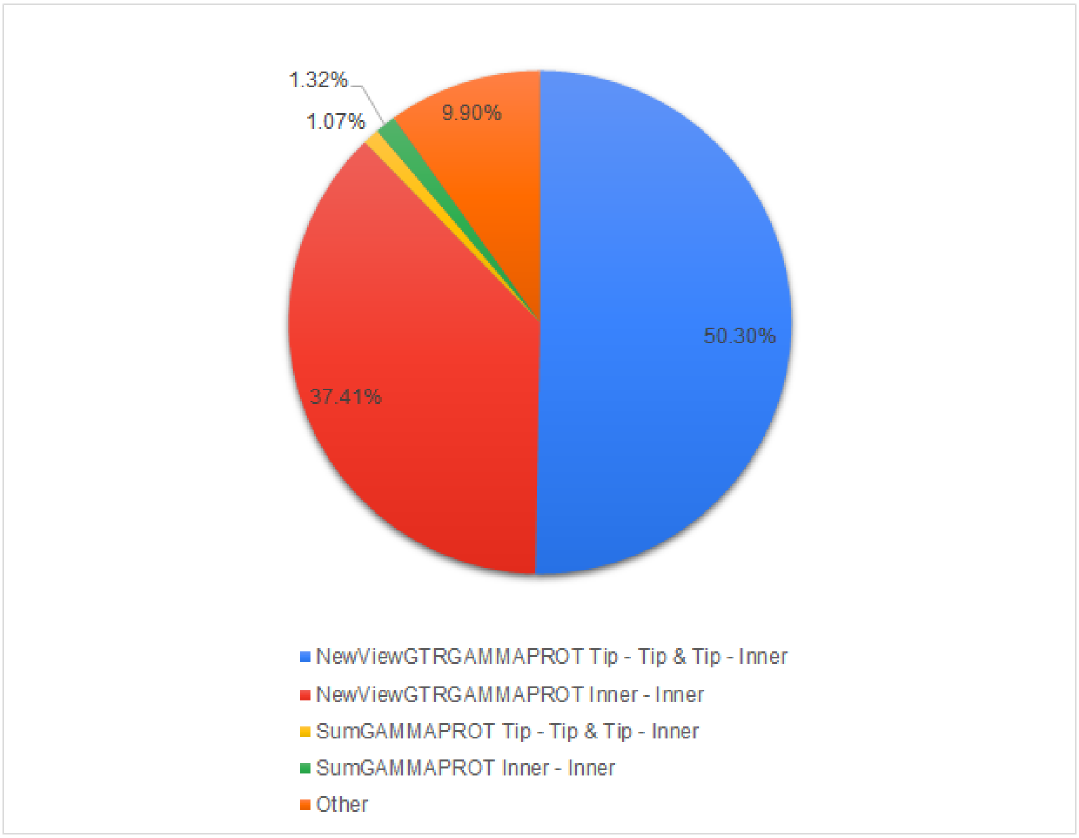


FIGURE 5.2: Max percentage-PLF

As it is distinguished PLF consumes the biggest percentage of the total execution time, with a range of 84.89%-87.71%. Both sequential and AVX version of this function reaches this range, as the total time is depended by the selected executable version of the algorithm. SumGAMMAPROT consumes a little but significant percentage of the total execution time, with a range of 2.37%-4.81%. This percentage might be seen low, but on runs, with big data sets, it can be equal to plenty of hours. It is mentioned that SumGAMMAPROT does not have an AVX implementation and its runtime depends on the runtime of the other functions. So using the AVX version, SumGAMMAPROT can reach the percentage of 13.4% as it will consume the same time while the other functions with AVX implementations will consume less time.

Looking into the inner cases, we found that the third case (inner-inner) is that one that delays the total execution time, occupying the biggest percentage of all three cases. The remaining two are equal. Moreover, the third case requires more operations in order to export the output data, which leads us to simplify the load and try to distribute it on multiple resources, in order to run concurrently. So according to these results, we decided to implement

FPGA-based accelerators for the above two functions, and their third case that input data derive from two inner nodes of a tree.

Completing the profiling of the selected functions, it would be useful to calculate and estimate the theoretical speedup both of each function and whole RAxML, given that we could design an ideal system. Adopting Amdahl's law

$$S_{\text{latency}}(s) = \frac{1}{1-p} \tag{5.1}$$

$$S_{\text{latency}}(s) = \frac{1}{(1-p)+\frac{p}{s}} \tag{5.2}$$

the ideal theoretical speedup of PLF would be x1.82, making to run in parallel only the proportion that is occupied by the inner-inner case. Achieving this speedup, then the whole RAxML could benefit from a speedup of x1.63. If parallelism was applied for all cases of PLF, then using the first leg 5.1 of Amdahl's law, we could achieve an x7.7 speedup of RAxML.

Following the same process for SumGAMMAPROT, making to run in parallel only the proportion that is occupied by the inner-inner case, the theoretical maximum speedup of this function would be x1.92. With this theoretical speedup, RAxML could not bring a significant acceleration, but only a factor of x1.02. This fact leads us to estimate a practical low speedup even though the designs would be ideal without any deviance.

5.2 DAER Architecture

Placing this application into computational intensive applications, its mapping procedure requires two main implementation parts. The first one is Data Plane, as an example, efficient interconnected units that accelerate processing. The second part is the Access Plane, as an example, efficient ways to access data and transfer them to/from the accelerator. Data plane construction is well understood and using High Level Synthesis (HLS) tool it is easy to be implemented. However, the access plane is more challenging while data fetching for big data is even more complex and time-consuming than processing.

So in order to avoid this problem, a Decoupled Access-Execute architecture and framework for Reconfigurable accelerators (DAER) [43] was used. Following DAER, the application is split into two parts, the part of data processing which is used solely for performing calculations, and the data fetching, which is used for memory transactions of input and output data.

The general architecture of DAER is depicted in figure 5.3 and can be used for mapping one or several accelerators, with or without inter-accelerator communication. The architecture of the accelerator is depicted in figure 5.4.

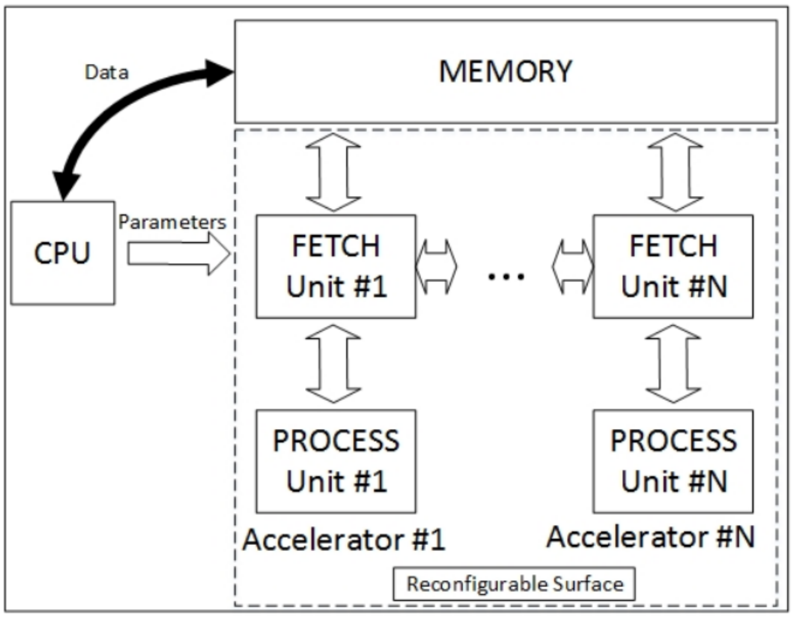


FIGURE 5.3: Proposed Decoupled Access-Execute Architecture for Reconfigurable accelerators (DAER).

The first part of DAER architecture is the fetch units, which are connected both to the CPU and the memory. The CPU connection is used only for passing parameters regarding the application's memory traces, i.e. starting memory addresses, array sizes, etc. The memory connection is used for fetching input data and sending back the results computed by the processing units. The second part of our architecture is the processing units, which consist of logic and/or arithmetic operations. The processing units work as simple data-flow engines. It communicates directly with the fetch units through FIFO-based links. These streams are used for passing data to the processing units and sending back the results. Both units are amenable to code-specific acceleration through HLS directives. In addition, they can be instantiated multiple times according to the needs of the mapped application and available resources. Hence the application code is split into two parts, one that performs memory accesses, and one that implements the main algorithmic workload. The memory access dependencies are resolved by distributing memory accesses to separate pipelined fetch units, which can send read requests and receive data, concurrently.

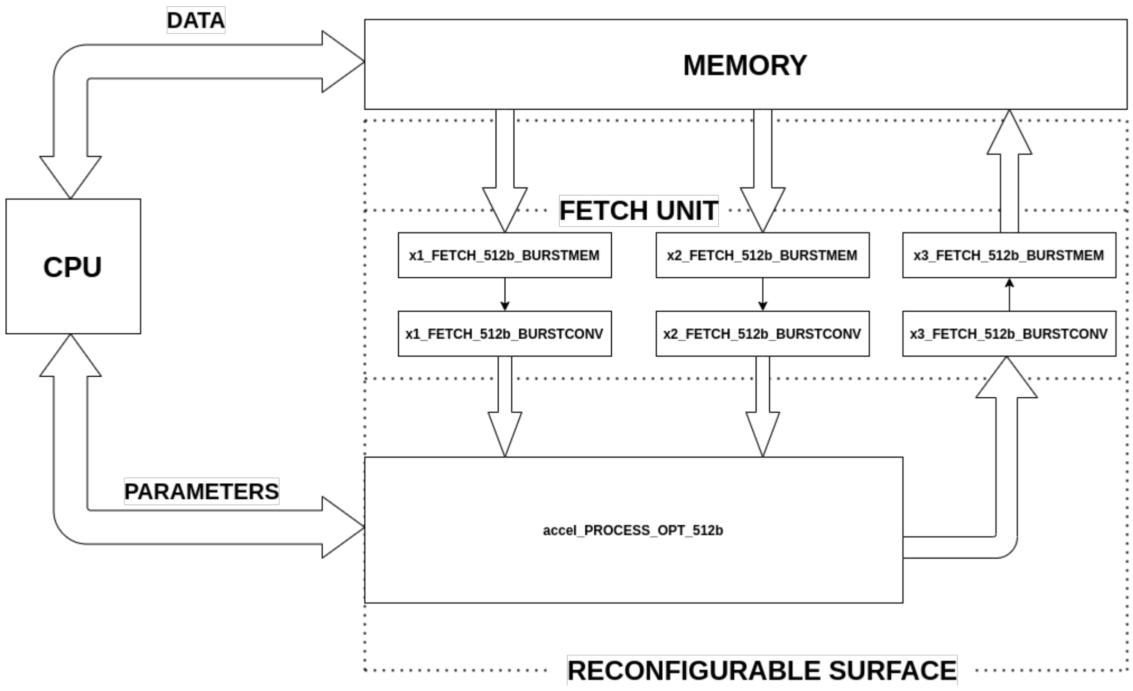


FIGURE 5.4: Decoupled Access-Execute Architecture On Our Designs

5.3 Kernel Architecture

The architecture of these two accelerators consists of fetch units for the transfer of the data in and out of the kernel and the processing units. Both accelerators have the same fetch units as the input and output data are the same datatype. They only differ on the processing units as each one provides different procedures in the RAxML.

5.3.1 Fetch Units

Both Kernels of this project use Fetch Units. They need two Fetch Units for the two input streams (for this purpose x1,x2 values) and another one for the output stream (x3 value). Each Fetch Unit consists of two fetch functions, BurstMem and BurstConv. The use of BurstMem is to pass input values to streams. Input values and streams are ap_uint<512> type. Afterward, these streams get under processing in the BurstConv function. There, input streams are converted into DataType streams which make them able to transfer packages of double values into the processing units instead of one value per cycle.

To achieve the best and feasible conversion, it was necessary to know the maximum bandwidth of FPGAs' streams, the number of essential input values needed to produce the outcome of an alignment pattern, and the type of values. Moreover, the best Pipeline Initiation Interval (II) can be calculated. In the current case and considering the available standards for this project, the above calculation is made following the next steps. First, the values needed for calculations and storage are double type, which means their size is 8 bytes or 64 bits. Afterward, for protein data and under the GAMMA model, 80 values are required per alignment site. Therefore the size of input data is 80 values x 64 bits = 5120 bits. The maximum bandwidths of the platforms' streams are 512 bits and 128 bits. So dividing the total size of input data with the streams' bandwidths, the outcome is 5120 bits / 512 bits = 10 and 5120 bits / 128 bits = 40. This number means that using 10 ap_uint<512> input values (or 40 input values), 80 double type values can be transferred to the processing units.

Ending the Fetch units, after the production of output values by the processing units, similar fetch functions are used to invert the output data to the initial type and then transfer them to global memory again.

5.3.2 Processing Units

In this subsection, the architecture of the two accelerators is described in detail and there are schematics of the whole data flow inside them. The architectures were built under the hardware constraints we had. It is mentioned that the functionality of PLF and SumGAMMAPROT were described in Chapter 4.

A) PLF

As previously referred, maximum likelihood calculation is corresponded by the PLF or else NewViewGTRGAMMAPROT. Below, there is a pseudo code of this function and it is a good reference to start and build the accelerator.

Algorithm 1 PLF , based on a three vector tree with root A

```
for $i = 0 : N$ do
    $X_1[80] \leftarrow B(i)vector$
    $X_2[80] \leftarrow C(i)vector$
    $X_3[80] \leftarrow A(i)vector$
    for $j = 0 : 3$ do
        for $k = 0 : 19$ do
            for $l = 0 : 19$ do
                //Calculation of left distance $(B \rightarrow A)$
                $Tmp_{x=x_1} \leftarrow X_1[j * 4 + l] * Left[j * 400 + k * 20 + l]$
                $Ump_{x=x_1} \leftarrow \sum_{i=0}^{19} Tmp_{x=x_1}$
                //Calculation of right distance $(C \rightarrow A)$
                $Tmp_{x=x_2} \leftarrow X_2[j * 4 + l] * Right[j * 400 + k * 20 + l]$
                $Ump_{x=x_2} \leftarrow \sum_{i=0}^{19} Tmp_{x=x_2}$
            //Multiplication of summarized $ump_{x1}$ and $ump_{x2}$
            $x1px2_j[k] \leftarrow Ump_{x=x_1} * Ump_{x=x_2}$
        for $k = 0 : 19$ do
            for $l = 0 : 19$ do
                //Multiplication of EV and x1px2 $\rightarrow X_3$
                $X_3[j * 4 + l] += x1px2_j[k] * EV[k * 20 + l]$
    //Continue with the scaling
    $scale \leftarrow 0$
    $addscale \leftarrow 0$
    for $l = 0 : 79$ do
        $scale += (ABS(X_3[l] < minlikelihood)$
    if $scale \neq 0$ then
        $X_3 \leftarrow X_3 * factorM$
        $addscale += wgt[i]$
    else
        $X_3 \leftarrow X_3$
```

This algorithm could implement the 4.1 example. The two inputs X_1 and X_2 are representing B and C vectors and output X_3 represents the A vector. As the study examines the usage of amino acids, each vector contains N*80 elements. The other three matrices Left, Right and EV are the substitution matrices and the eigenvectors used by the GTR-GAMMA model which was described in Chapter 4.

Analyzing the pseudo-code, it can be distinguished that the procedure to calculate the final maximum likelihood is divided into two parts. The first one is the calculation of x1px2 which corresponds to the likelihood, a product that comes up from multiplications of two inputs with their substitution matrices. The second one is the final and complete calculation of maximum likelihood accompanied by the scaling process. So, making this observation, on this architecture, these two parts were designed separately and constitute two different process units.

Starting with the first part, the **Calculation Of Likelihood**, it is observed that in the innermost loop, the products of the multiplications of $X_1 * Left$ and $X_2 * Right$ are stored into two values Tmp_{x1} and Tmp_{x2}. As these multiplications can be concurrently executed, we designed and adjusted a circuit to achieve the best flow of the under execution data. We can see that for each j site of input matrices, 20 elements are multiplied with the corresponding substitution matrix which contains 20×20 elements. This means that every site is multiplied 20 times :

1^{st} twenty values of a site for j=0 are multiplied with k=[0:19] elements
2^{nd} twenty values of a site for j=1 are multiplied with k=[0:19] elements
3^{rd} twenty values of a site for j=2 are multiplied with k=[0:19] elements
4^{th} twenty values of a site for j=4 are multiplied with k=[0:19] elements

Unrolling the above relations we can see that each element of the 4 different twenties are multiplied by 20 different elements :

$TMP_{X1,2}$[0] = $X_{1,2}$[j*4 + 0] * Left,Right[j*400 + k*20 + 0]
$TMP_{X1,2}$[1] = $X_{1,2}$[j*4 + 1] * Left,Right[j*400 + k*20 + 1]
$TMP_{X1,2}$[2] = $X_{1,2}$[j*4 + 2] * Left,Right[j*400 + k*20 + 2]
$TMP_{X1,2}$[3] = $X_{1,2}$[j*4 + 3] * Left,Right[j*400 + k*20 + 3]
$TMP_{X1,2}$[4] = $X_{1,2}$[j*4 + 4] * Left,Right[j*400 + k*20 + 4]
$TMP_{X1,2}$[5] = $X_{1,2}$[j*4 + 5] * Left,Right[j*400 + k*20 + 5]
$TMP_{X1,2}$[6] = $X_{1,2}$[j*4 + 6] * Left,Right[j*400 + k*20 + 6]
$TMP_{X1,2}$[7] = $X_{1,2}$[j*4 + 7] * Left,Right[j*400 + k*20 + 7]
$TMP_{X1,2}$[8] = $X_{1,2}$[j*4 + 8] * Left,Right[j*400 + k*20 + 8]
$TMP_{X1,2}$[9] = $X_{1,2}$[j*4 + 9] * Left,Right[j*400 + k*20 + 9]
$TMP_{X1,2}$[10] = $X_{1,2}$[j*4 + 10] * Left,Right[j*400 + k*20 + 10]
$TMP_{X1,2}$[11] = $X_{1,2}$[j*4 + 11] * Left,Right[j*400 + k*20 + 11]

$TMP_{X1,2}$[12] = $X_{1,2}$[j*4 + 12] * Left,Right[j*400 + k*20 + 12]
$TMP_{X1,2}$[13] = $X_{1,2}$[j*4 + 13] * Left,Right[j*400 + k*20 + 13]
$TMP_{X1,2}$[14] = $X_{1,2}$[j*4 + 14] * Left,Right[j*400 + k*20 + 14]
$TMP_{X1,2}$[15] = $X_{1,2}$[j*4 + 15] * Left,Right[j*400 + k*20 + 15]
$TMP_{X1,2}$[16] = $X_{1,2}$[j*4 + 16] * Left,Right[j*400 + k*20 + 16]
$TMP_{X1,2}$[17] = $X_{1,2}$[j*4 + 17] * Left,Right[j*400 + k*20 + 17]
$TMP_{X1,2}$[18] = $X_{1,2}$[j*4 + 18] * Left,Right[j*400 + k*20 + 18]
$TMP_{X1,2}$[19] = $X_{1,2}$[j*4 + 19] * Left,Right[j*400 + k*20 + 19]

Understanding how the multiplications are generated and executed, we go forward to the next diagram which shows the flow of data combined with operations:

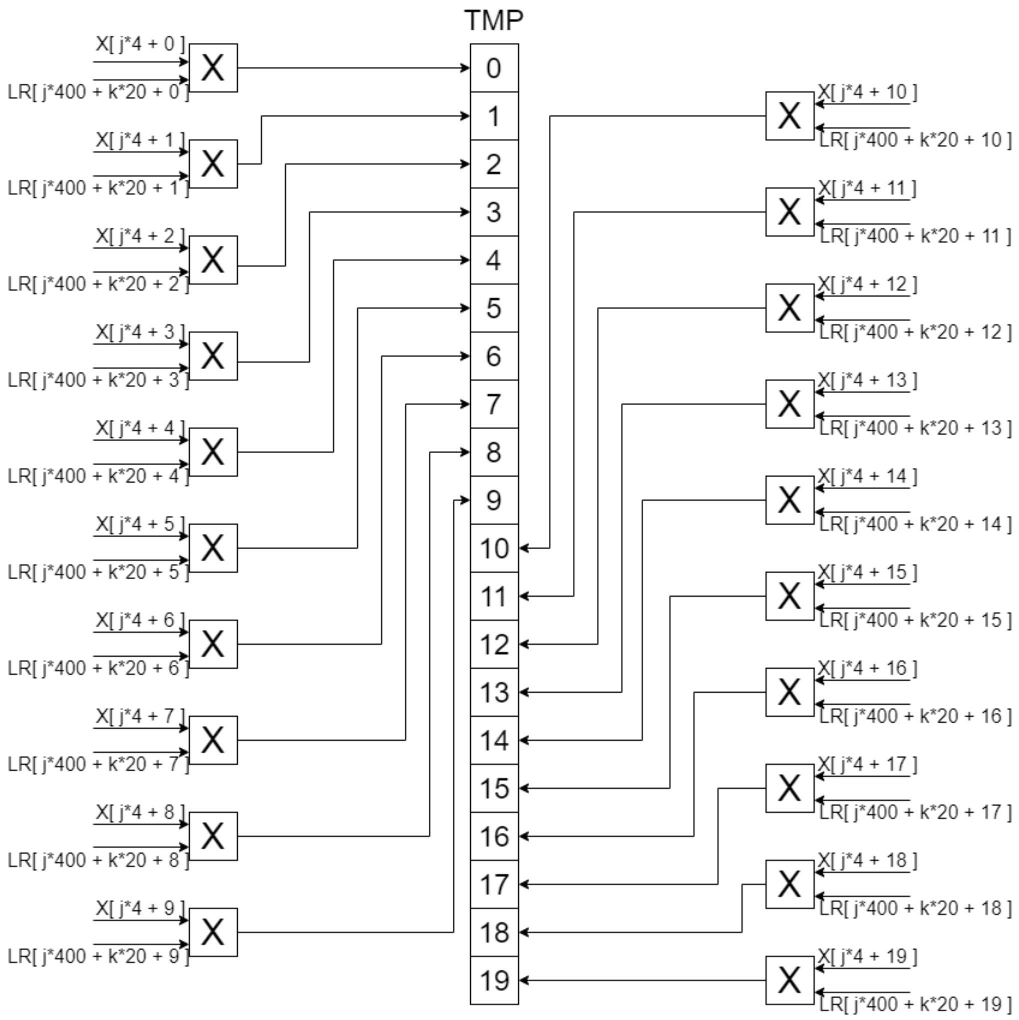


FIGURE 5.5: Multiplications For The Calculation of UMP for X1,X2 input vectors

Figure 5.5 is abstract and matches for both X1 and X2 while LR is either Left or Right substitution matrix and TMP is a unique vector for each input vector. Storing each product of these multiplications in a different value, make it easier to accumulate all values in order to sum them and progress the other calculations. The next step is the summation of the 20 elements of each TMP, which produce the two values, ump_{x1} and ump_{x2} required for the computation of the initial likelihood x1px2. After the calculation of UMP_{X1} and UMP_{X2}, a multiplication between them follows to get x1px2.

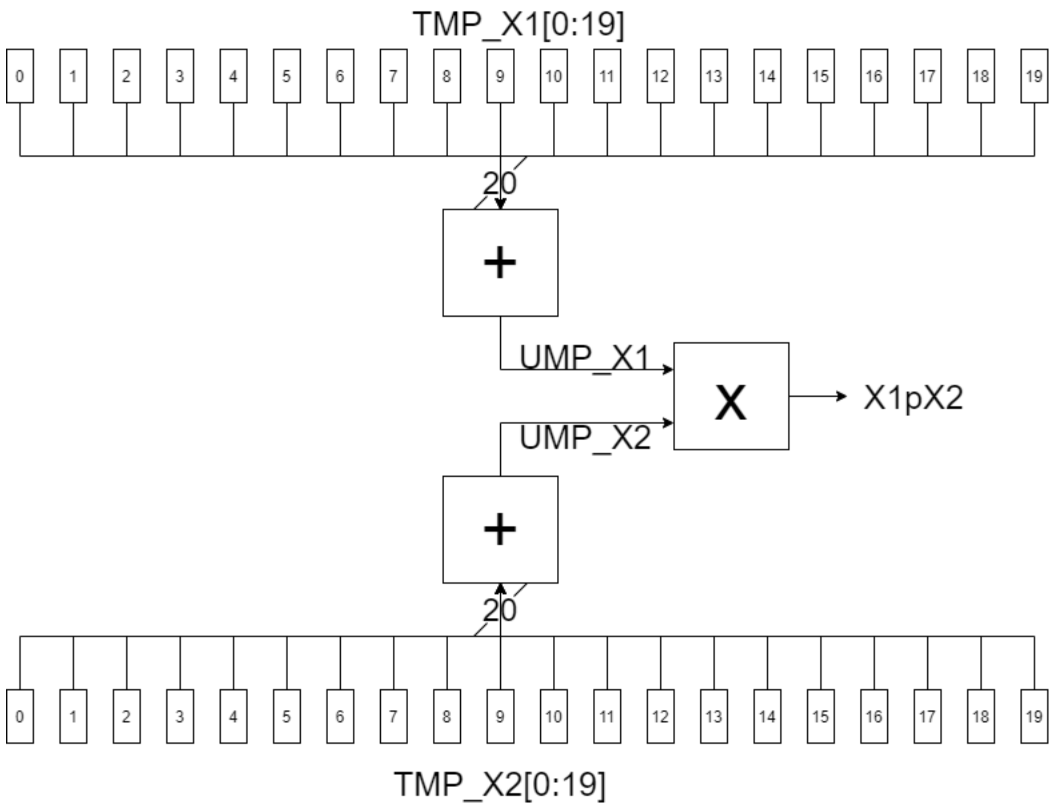


FIGURE 5.6: Summation of TMP vector to produce UMP_{X1} and UMP_{X2} and their multiplication for X1pX2

The result of the above multiplication is not the likelihood for a whole site j but a likelihood of an element of the 20 of each site. To completely calculate the likelihood for a site j, we need to set all above circuits as a single one and iterate it k=[0:19] times. This iterations springs from the middle loop of algorithm 1 and its purpose is to insert for every k, 20 new elements of the Substitutions matrices to the 5.5 circuit and calculate 20 new values in TMP which result in another x1px2 value. As a reminder, the Γ model provides us two new substitution matrices with $4 \times 20 \times 20$ dimensions. Comparing these dimensions with the above figures, it is easy to understand that the dimension 4 corresponds to the four different sites of a vector, the first dimension 20 corresponds to the 20 multiplications on 5.5 and the last dimension

20 corresponds to the k times that new twenty values will proceed for the multiplications. Ending all k values, the 20 likelihoods of an entire site are ready to proceed to the next operations.

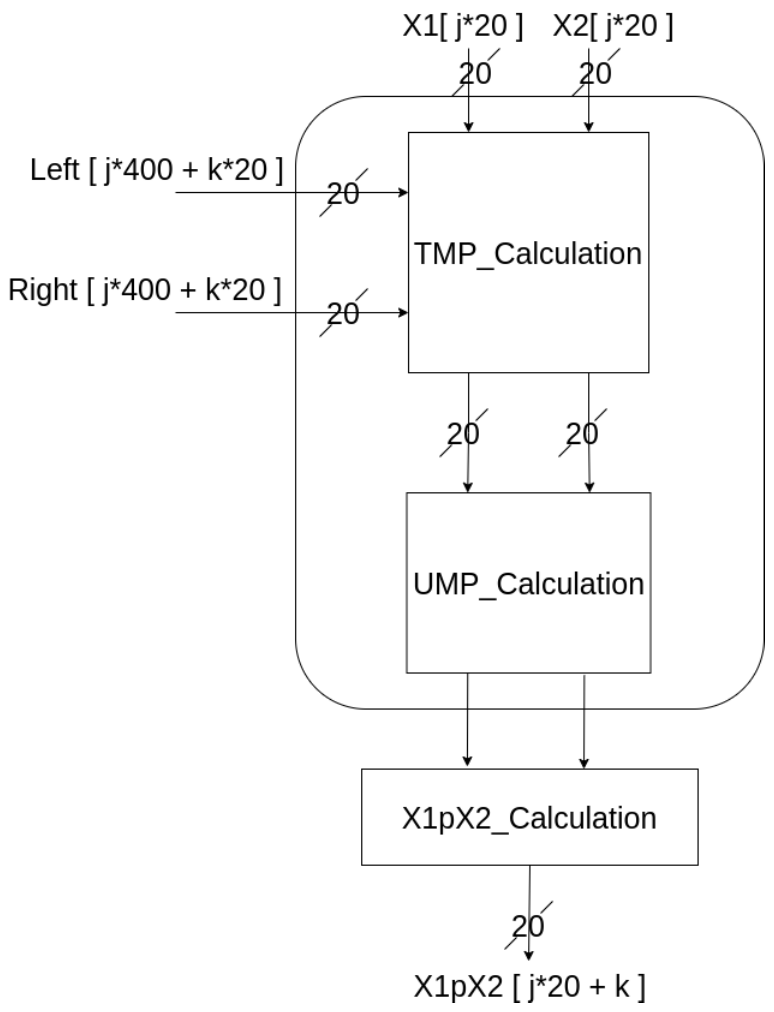


FIGURE 5.7: Calculation of likelihoods for a j site

Looking again on the algorithm 1, it is distinguished that, provided that 20 likelihoods of a *j* site have been produced then they proceed to the next phase of the final calculation to export a completely calculated site. Into our accelerators, there is a variance. Instead of completing all required calculations for a whole site per j time, we first take out x1px2 values for the whole vector that is 4×20 values. That is also the reason why on figure 5.7 we depict x1px2 as a vector with $j * 20$ length. So, the processing unit of Likelihood Calculation is completed and a top-view flow of data is shown below:

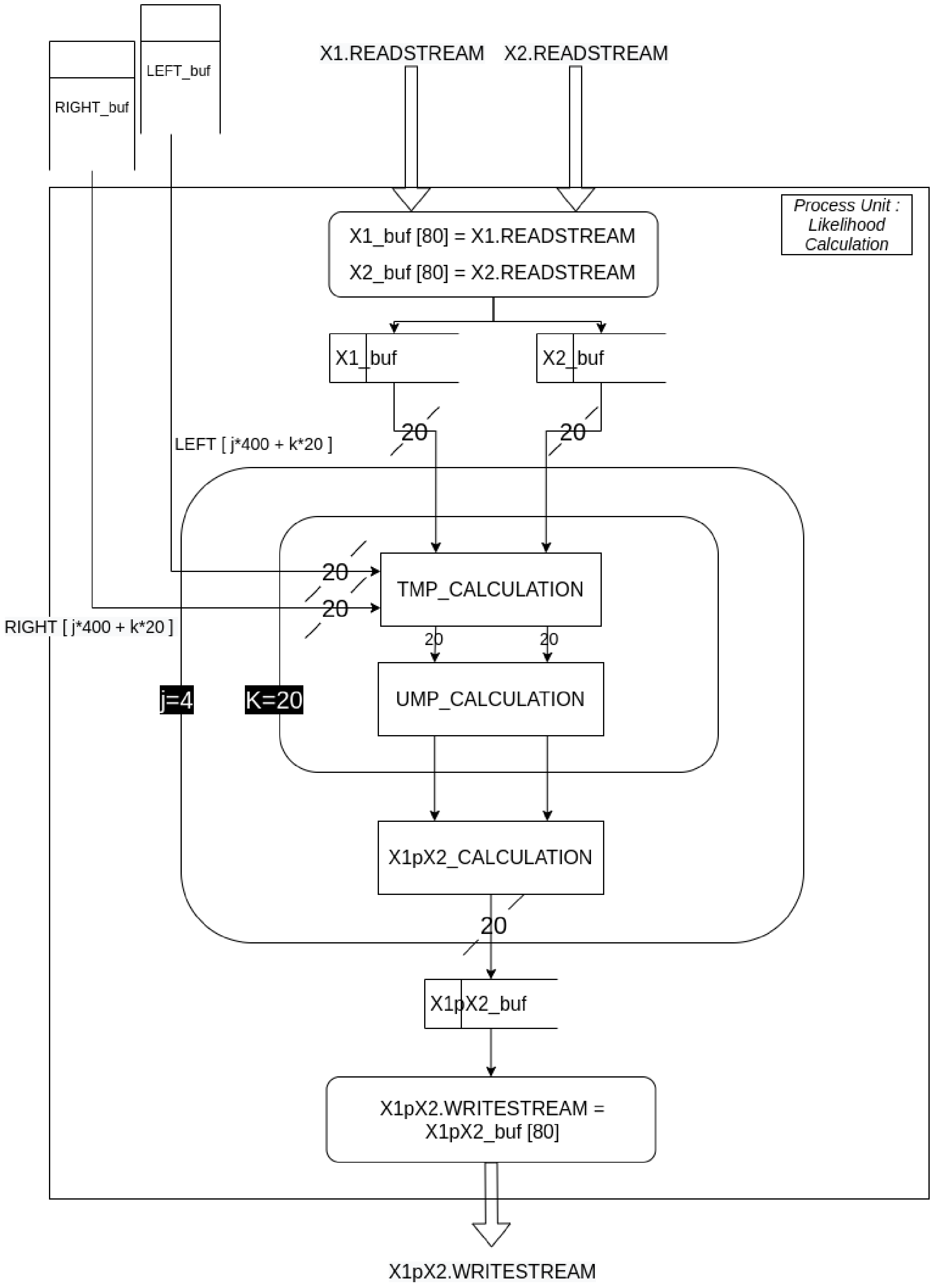


FIGURE 5.8: Data Flow of Processing Unit : Likelihood Calculation

The second processing unit of PLF is the **EV Final Calculation** of vectors likelihood. Here is the spot that eigenvectors are useful to make the required optimizations. Following the equation 4.8, this process consists of multiplications and summations and there is the usage of EV matrix and X1pX2, produced by the previous processing unit. Moreover, the algorithm 1 shows that every element of a j site of X3 vector is equal with a sum of twenty multiplications between each element of X1pX2 vector with twenty elements of EV matrix. It is recalled that through the first processing unit we produce twenty values for each j site of the input vector, which is in overall 80 values for a whole vector. Next diagram shows the vectors' likelihood is produced:

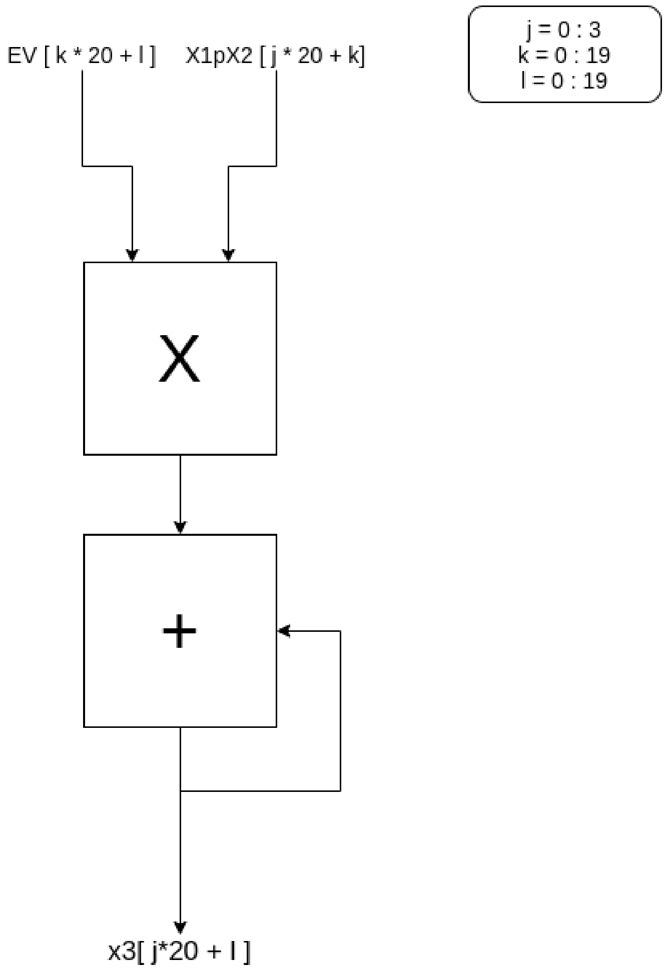


FIGURE 5.9: Multiplication Of EV with X1pX2 and a recursive addition for X3

As we can see in the above figure, every element of a j site must be put through this process and be multiplied with certain values from the EV matrix. Following as an example the previous processing unit, we can also execute 20 units of the circuit 5.9 for all elements of a j site concurrently. There is a graphic representation below:

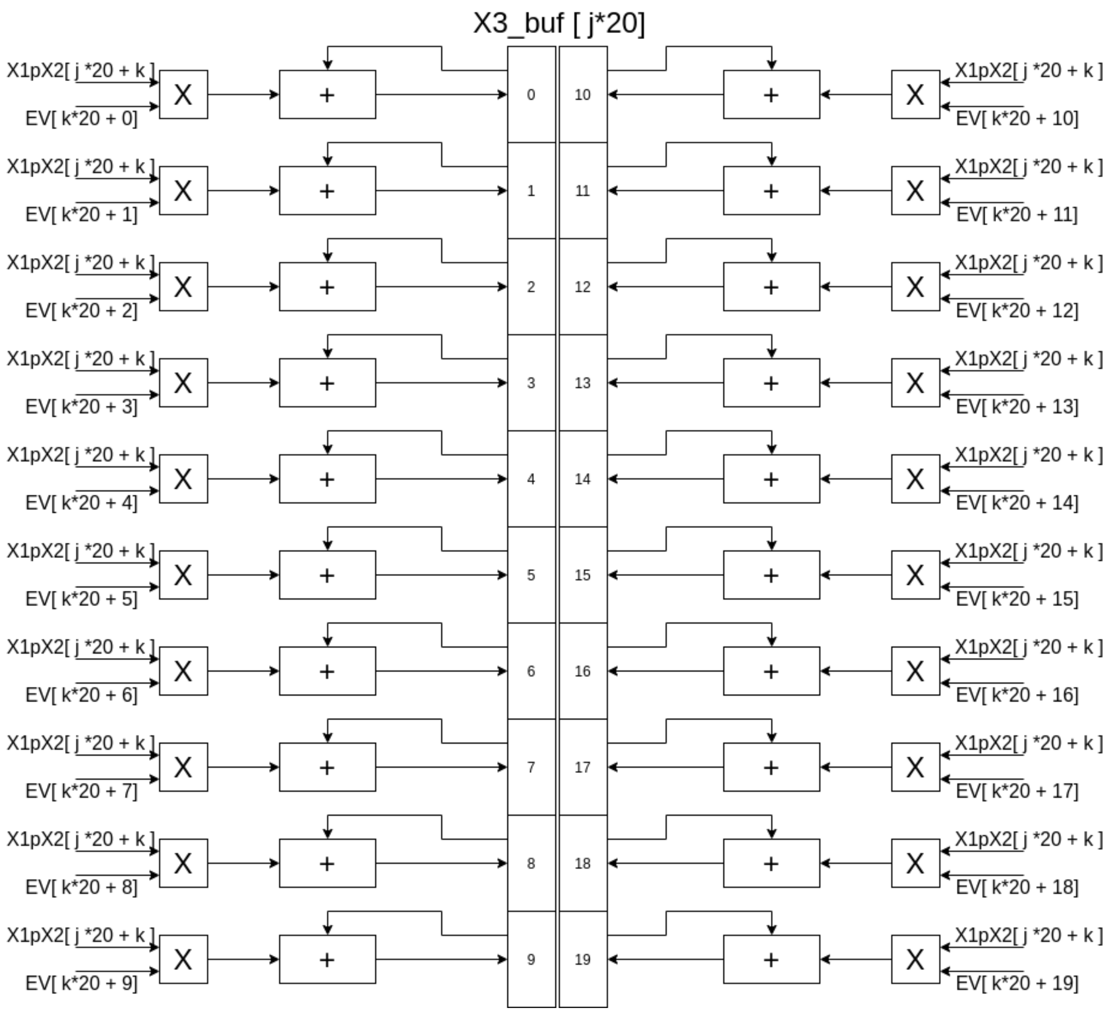


FIGURE 5.10: Final Calculation of a j site of X3 vector

It should be clarified that the above circuit calculates the final likelihood of a site provided that the k loop runs out all its 20 iterations, which means all required values from EV and X1pX2 matrices are going to be accessed. In this way, the calculation of the final likelihood of all N vectors is coming to an end. The only left part of the whole process is the scaling process.

Into the scaling process, a whole vector is going under comparison with a constant value of RAxML which corresponds to a minimum likelihood. For all 80 elements of a vector, if the absolute value of an element is less than the minimum likelihood then a scale counter is increased. At the end of this comparison of all elements of a vector, if the counter is zero means that all previous calculations are accurate and acceptable. On the contrary, if the counter is not zero means that some elements of the vector can't reach the requirements to go forward and need some extra process. There, a value named addscale is kept by adding, the corresponding to the i_{th} vector wgt value, which is flowed out by the wgt input stream. Finally, if there is a hit on the scale counter, the whole vector is multiplied by a constant factor and then proceeds to the exit; differently, the vectors' data remain without any interference and go forward to exit. In addition, ending the access of all N vectors, there is another output scalerIncrement which is equal to the addscale value.

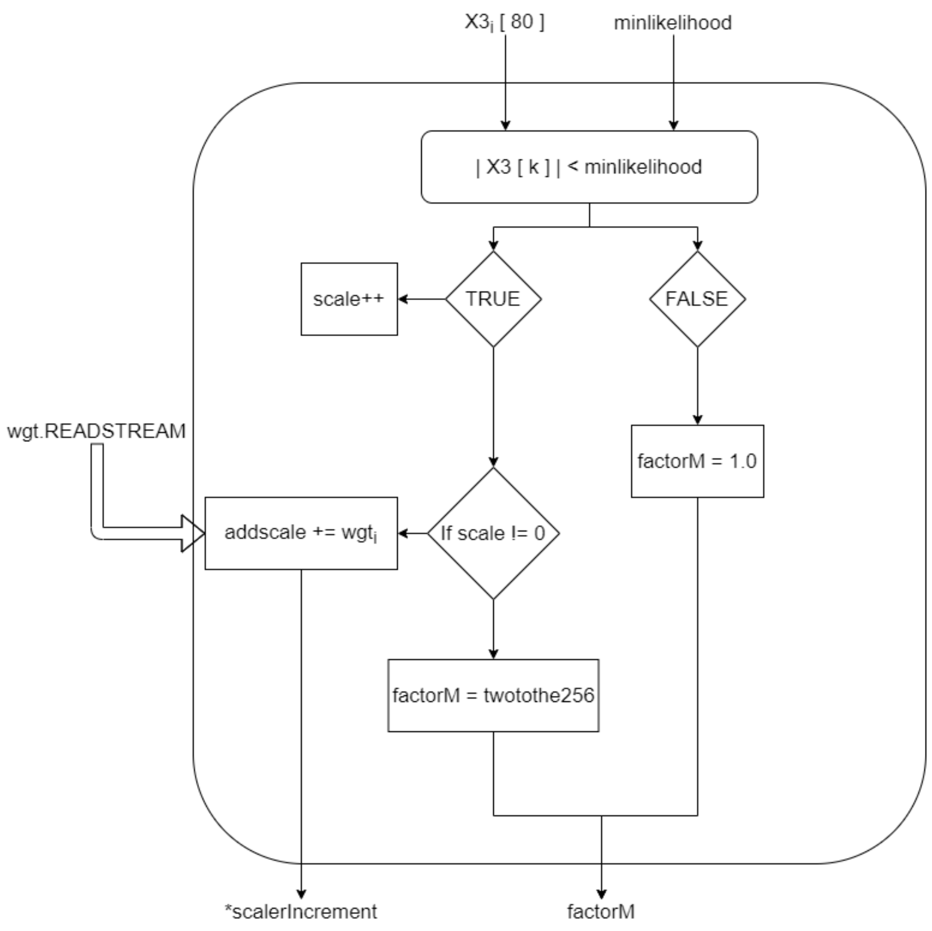


FIGURE 5.11: Scaling process of X3 vector

Completing all sub-processes, the final connection among them is figured below:

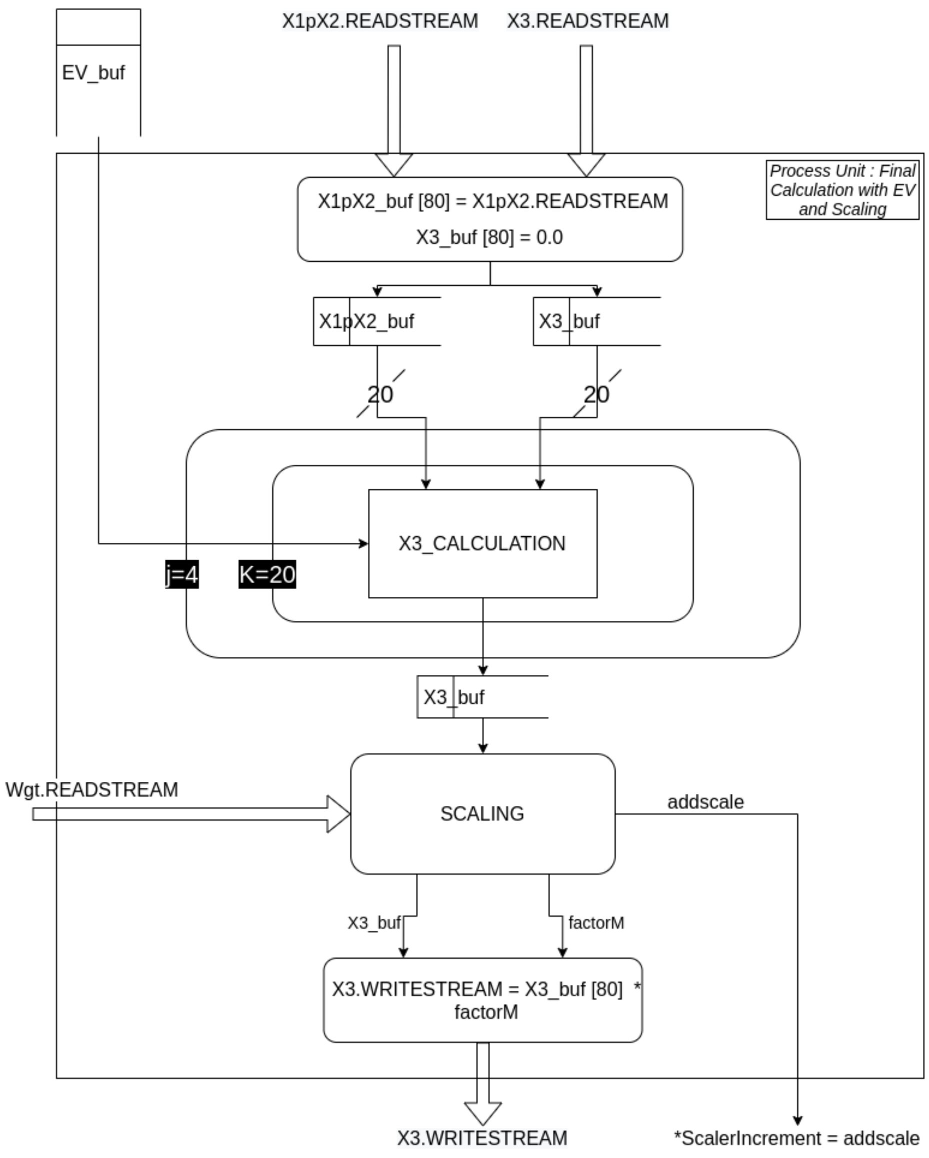


FIGURE 5.12: Data Flow of Processing Unit : EV Final Calculation

Finally, having complete the structure of all basic and required functions, the data flow of the main core can be engraved. Having as a reference the figures 5.3 and 5.4 next diagram shows in detail the data flow. It starts when fetch units retrieve data from global memory, then forwards to the main processes and calculations and finally ends to transfer of data form fetch unit back to global memory. It is mentioned that the connection between the different processes is designed with streams except for these which transfer constant values and buffers. Moreover, the set of processes is separated into different steps-phases of execution and this fact will be analyzed in the next Chapter. Finally, all values out of the main core are meant to be stored in global memory and they are the objects of transfers among the main core and memory.

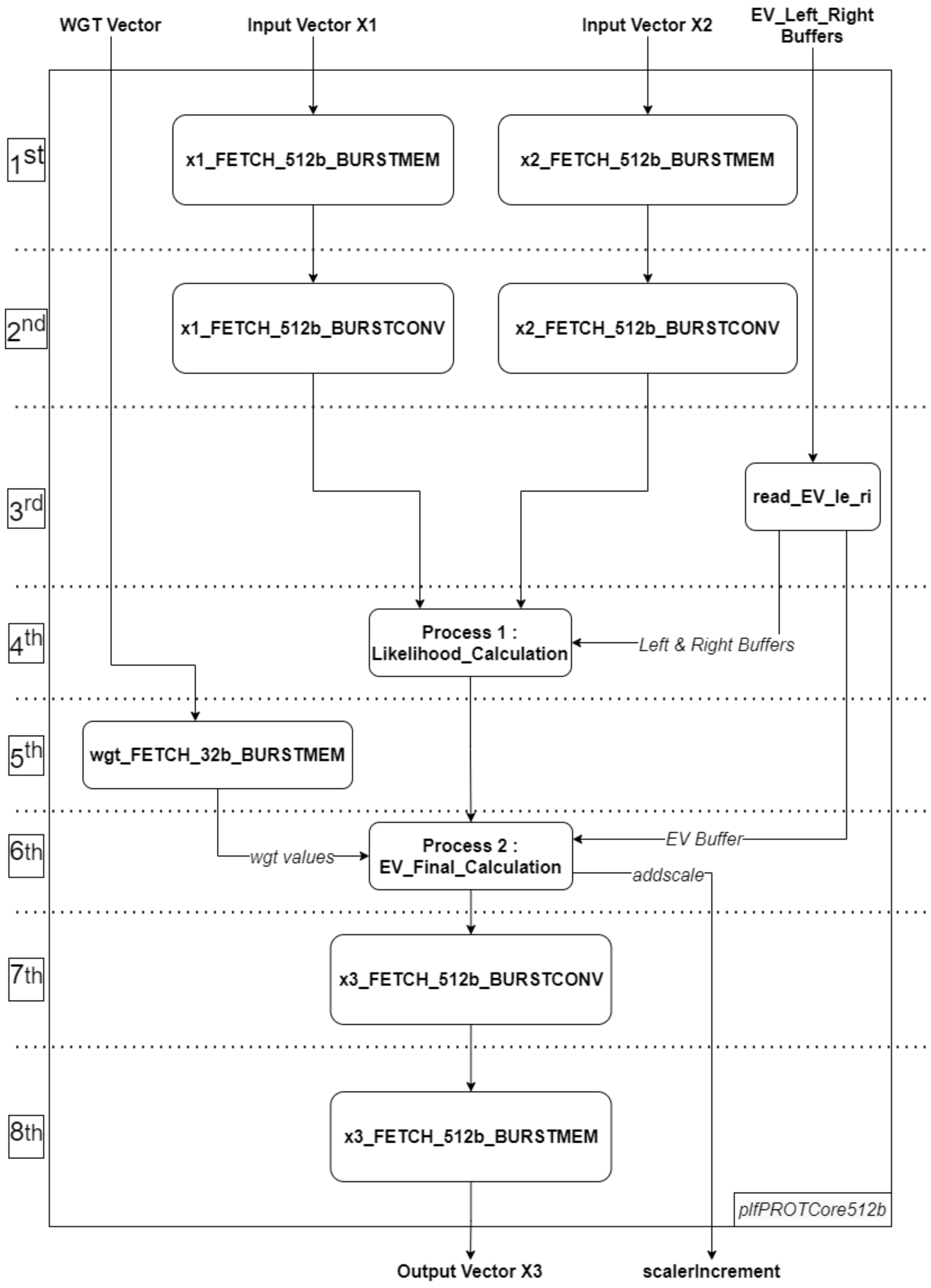


FIGURE 5.13: Data Flow of Main Core

B) SumGammaPROT

To understand the data flow of this function and build the architecture, we took as reference the pseudo-code of the function SumGammaPROT:

Algorithm 2 SumGamma Calculation

```
for $i = 0 : N$ do
    $X_1[80] \leftarrow Left(i)vector$
    $X_2[80] \leftarrow Right(i)vector$
    $X_3[80] \leftarrow Sum(i)vector$
    for $j = 0 : 3$ do
        for $k = 0 : 19$ do
            //Calculation of Sum
            $X_3[j * 4 + k] \leftarrow X_1[j * 4 + k] * X_2[j * 4 + k]$
```

The processing unit of this function consists of a main multiplication of 2 vectors, the left one and the right one. In detail, into the outer loop of this process unit which is executed N times, there are 3 inner loops, be implemented concurrently under the assignment of loop pipeline. The first one (addressed to input data) and the last one (addressed to output data) help in the transfer of the data from fetch units to the processing unit. The middle and the innermost one is the multiplication of the left and right inputs which produce the output result. In the next figure, the data flow of the processing unit is clearly figured.

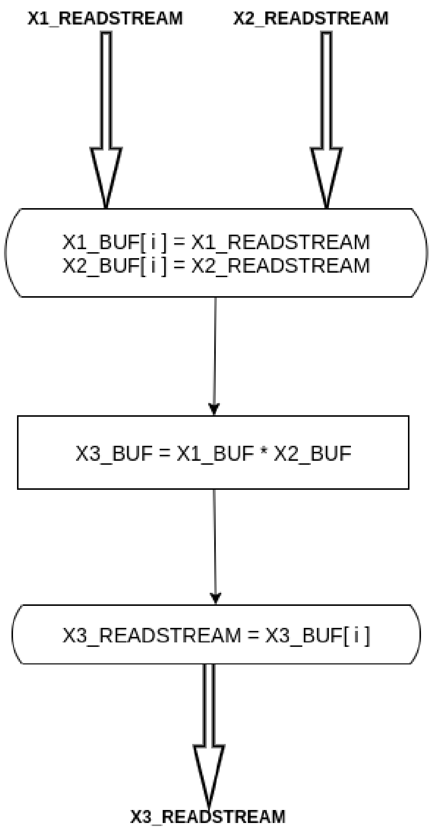

X1_READSTREAM
X2_READSTREAM
X1_BUF[i] = X1_READSTREAM
X2_BUF[i] = X2_READSTREAM
X3_BUF = X1_BUF * X2_BUF
X3_READSTREAM = X3_BUF[i]
X3_READSTREAM

FIGURE 5.14: SumGAMMA - Flow Chart

Stepping into the multiplication process, it is obvious that some operations can be executed in parallel. In detail, 80 (4×20) values of x1 and x2 input vectors insert this process and they are multiplied among them. So, these *20* multiplications can be executed concurrently, and consecutively all 80 values in *4* iterations. The final procedure can be described as :

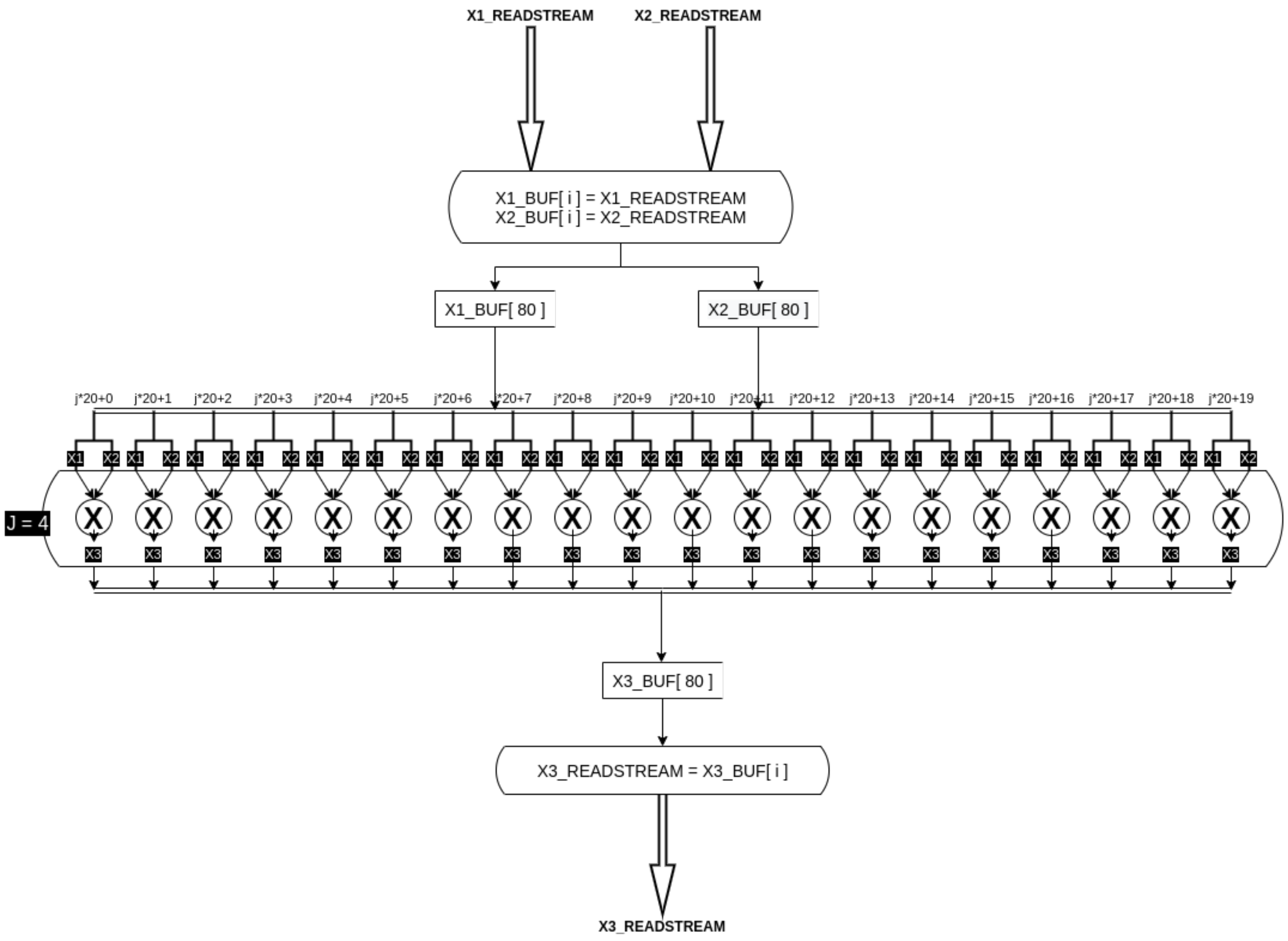


FIGURE 5.15: SumGAMMA - Detailed Data Flow

Completing the above process then the data flow of the main core is ready to be designed in the same way as PLF and is shown below:

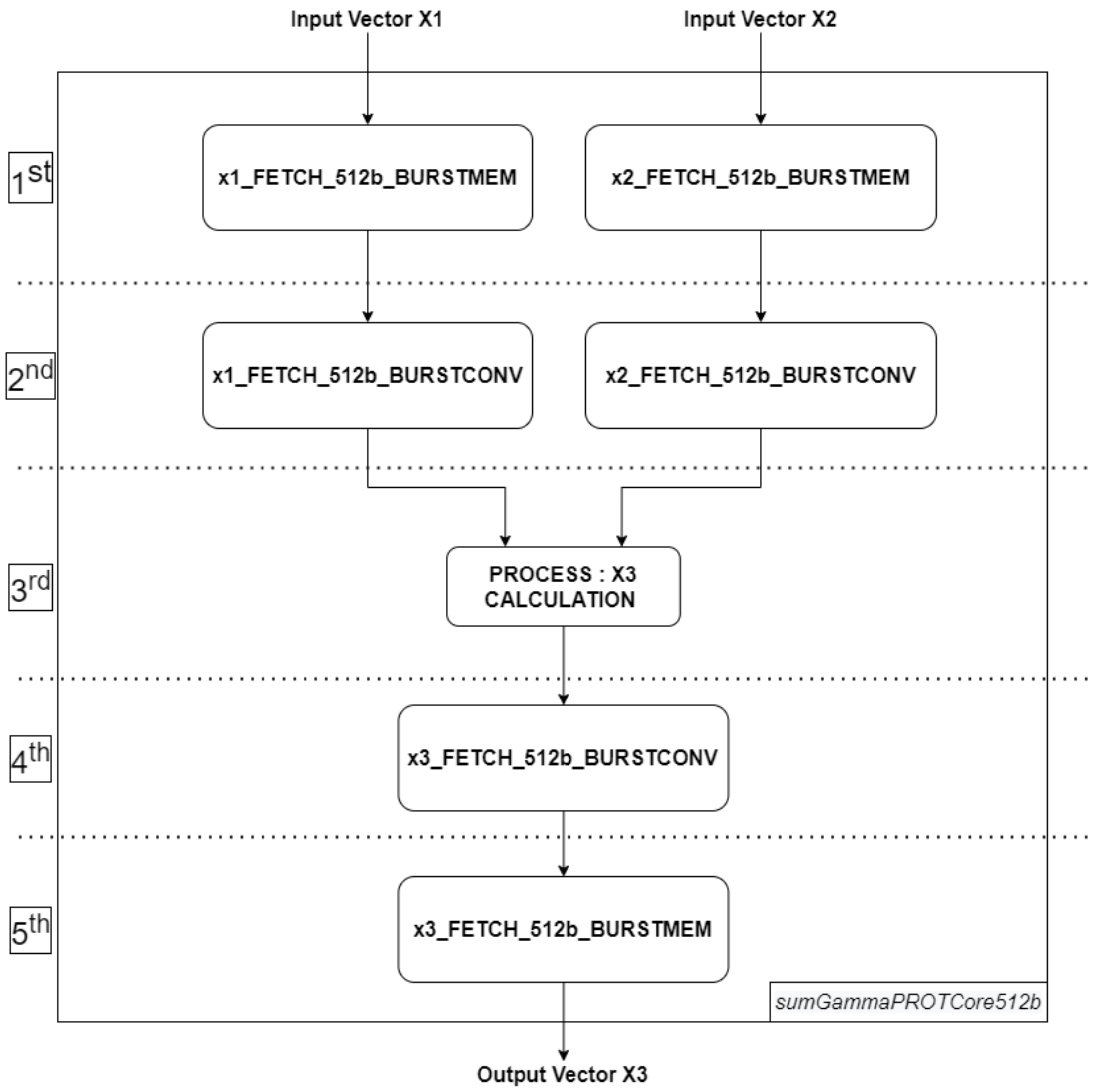


FIGURE 5.16: Data Flow of Main Core

Chapter 6

FPGA Implementation

6.1 Tools Used

The two accelerators, described in the previous chapter, were implemented and optimized for FGPA platforms using Xilinx Vivado Design Suite - HL System and Xilinx SDx Environment [44] [45], both on edition 2018.3. Vivado Design Suite is a software suite developed by Xilinx for its FPGA devices for analysis and synthesis of Hardware Description Language (HDL) designs, written in VHDL or Verilog. SDx Environment consists of SDAccel, SDSoc, and SDNet, and offers GPU-like and familiar embedded application development and runtime experiences for C, C++, and/or OpenCL development, targeting FGPA platforms and FPGA based acceleration cards. The basic tools used in this work are Xilinx SDAccel and SDSoc while Xilinx Vivado HLS and Xilinx Vivado IDE are prerequisite tools to handle the first ones, making them able to be used separately to interfere in some synthesis phases of SDAccel and SDSoc.

6.1.1 Vivado High Level Synthesis (HLS)

Xilinx Vivado High-Level Synthesis (HLS) [46], is a tool included in the Xilinx Vivado Design Suite, allowing for a higher level of abstraction design of HDL systems. Vivado 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 Vivado 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 instruct the synthesis procedure to implement a specific behavior, optimization, and resource management. Directives are optional and do not affect the input code's behavior. They are also not specialized into some certain programming language while they generalized for all acceptable ones. Their correct usage can improve the IP's performance while the wrong one can hurt it. 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.

Having complete the appropriate interference a C/C++ testbench should be used to debug the code's behavior prior to synthesis. In testbench, all input data are fed, and then the output data are checked for their correctness. Verification of the exported IP block is done using the C/RTL Cosimulation functionality, which uses the same C/C++ testbench, but replaces the function's call with the exported IP block call.

Synthesis Report

A synthesis report is generated whenever HLS successfully synthesizes an IP Block, showing some useful performance and resource utilization metrics. This report is generated for both total IP Block but also every individual instance of it. So, using this package of synthesis reports, a designer can easily find whether their goal was achieved and optimized, or else target the bottleneck to further optimize their design in terms of both performance and resources. Some of the metrics are presented and explained below:

- **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.
- **Area:** The utilization of hardware resources required of instances or whole IP's implementation into the target FPGA. The hardware resource

types are Block RAM (BRAM) and Ultra RAM (URAM), Digital Signal Processing (DSP) units, Flip Flops (FF), and Lookup Tables (LUT). There is also a detailed report, showing the number of hardware resources required for every hardware component type, which include DSPs, Expressions, First-In-First-Out (FIFO) queues, Instances, Memories, Multiplexers, and Registers.

Optimization Directives

As mentioned above, HLS provides a set of directives allowing us to optimize the designs and subsequently to instruct its behavior. These directives can be added directly into the input code in form pragmas which is an identifier for the preprocessor. Alternatively, we can select them into the GUI of HLS. The directives that are used in this work are presented below:

- **Interface:** Maps the top-level function's arguments to RTL ports to configure the IP block's functionality.The interface directive specifies each argument's port type. Here two port types are used s_axilite and m_axi.
- **Dataflow:** Enables parallel execution of functions and loops, increasing throughput and latency. It is used on our top level function.
- **Pipeline:** Reduces the number of clock cycles a function or loop can accept new inputs, by allowing the concurrent execution of operations. Moreover, it accepts as parameter an *Initiation Interval (II)* value, trying to target this number in clock cycles by unrolling all functions on the selected area (in some cases it substitutes the Unroll directive). In this project, having calculated the optimal Initiation Interval, we passed *II = 10* as target.
- **Inline:** Removes a function as a separate entity in the hierarchy. After inlining, the function is dissolved into the calling function and in some times achieves to reduce latency and initiation interval due to lower function call overhead. Into this work this directive is used for some functions that transform I/O data.
- **Array Partition:** Partitions an array into multiple smaller arrays or assigns each array's element to its register. This partitioning increases the read and write ports of the array on the hardware level, allowing for parallel I/O and computations. There are three different types of partitioning an array, *Complete*, *Block* and *Cyclic*. Complete partitioning, decomposes the array into individual registers which means that there

is an instant access to all of them. Block partitioning, creates smaller arrays from consecutive blocks of the original array. This effectively splits the array into N equal blocks where N is the parameter *factor* and should be an perfect divided integer with the number of elements. Cyclic partitioning creates smaller arrays by interleaving elements from the original array. The array is partitioned cyclically by putting one element into each new array before coming back to the first array to repeat the cycle until the array is fully partitioned. In other words, array is split into given *factor* sub-arrays and elements are stored sequentially into each one new sub-arrays.

- **Resource:** Specifies the resource (core) is used to implement a variable. It is used in this project to assign some arrays as BRAMs or URAMs.

6.1.2 Vivado SDx

As it was referred above, SDx Development Environment consists of three environments and they are used to accelerate applications. For this study, only SDSoc and SDAccel is used. SDSoC Development Environment is used for Zynq Ultrascale+ MPSoC and Zynq-7000 SoC families, while the SDAccel Development Environment for Data Center and PCI-e based accelerator systems. Both environments were required because there are two different implementations on different platforms.

SDSoc

Using SDSoc [47]the designer should use some directives (sds as prefix) in order to create and synchronize its host with the kernel. They are shown below:

- **Data access_pattern:** Specifies the data access pattern in the hardware function so as to determine the hardware interface to synthesize. Using the Sequential pattern a streaming interface is generated, otherwise, using a random pattern, a RAM interface is generated.
- **Data copy:** Means that data are explicitly copied between the host processor memory and the hardware function.
- **Data zero_copy:** Means that the hardware function accesses the data directly from shared memory through an AXI master bus interface.

- **Data mem_attribute:** Tells the compiler whether the arguments have been allocated in physically contiguous memory. Using the pattern *PHYSICAL_CONTIGUOUS* means that all memory corresponding to the associated array is allocated using sds_alloc. Otherwise, the default value is passed which is *NON_PHYSICAL_CONTIGUOUS* and means that all memory corresponding to the associated ArrayName is allocated using malloc or as a free variable on the stack.
- **Data sys_port:** Used in order to assign the arguments to specific memory ports.

SDAccel

On the contrary to SDSoc [48], handling the SDAccel environment there is no need to use the sds directives. The main optimized kernel code remains the same and the only change is on the host code. OpenCL instructions are used in order to pass/pull up the input/output arguments and data to the kernel and global memory and after that to synchronize the appropriate tasks. In the previous chapter 5 on section 5.3, it is described how OpenCL is used in this work and some design attributes we can handle with OpenCL.

6.1.3 Vivado IDE

Xilinx Vivado Integrated Design Environment (IDE) [49] is the basis for all Xilinx tools. It can compile, synthesize, implement, place and route FPGA hardware designs written in high-level languages such as C/C++, and HDLs such as VHDL and Verilog. Moreover, using the IP Integrator tool, hardware systems can be designed by graphically connecting IP blocks and configure them through their GUI with no coding involved.

The way that SDx Develepment Environment and Vivado IDE connect is that after optimizing and synthesizing the kernel and the host code on SDx, then this code passes to Vivado IDE as Rtl system and following the steps of synthesizing, implementation and place & route, it finally generates the bitstream of application, ready to be installed on the target platform. During the phase of implementation and place & route, Vivado IDE follows some design strategies, either default by system or strategies by designers' choice. These strategies consist of some steps such as OPT_DESIGN, PLACE_DESIGN, PHYS_OPT_DESIGN, ROUTE_DESIGN, and POST_ROUTE_DESIGN. Each step focuses on a different implementation or logic optimization process and there are many alternate directives and options for them [50]. To interfere and change these steps through the SDx environment, it is needed to assign these instructs on the xocc (Xilinx OpenCL Compiler) with the parameter *-xp* which enables the passing of parameters and properties of Vivado IDE to the running project. Below, the directives used for each step are referred:

- **OPT_DESIGN = ExploreArea:** Runs multiple passes of optimization with an emphasis on reducing combinational logic.
- **PLACE_DESIGN = SSI_BalanceSLRs:** Partitions across Super Logic Regions (SLRs) while attempting to balance Super Logic Lines (SLLs) between SLRs. In other words, trying to balance resources on all SLRs of the platform.
- **PHYS_OPT_DESIGN = AggresiveFanoutOpt:** Uses different algorithms for fanout-related optimizations with more aggressive goals.
- **ROUTE_DESIGN = AlternateCLBRouting:** Chooses alternate routing algorithms that require extra runtime but may help resolve routing congestion.
- **POST_ROUTE_DESIGN = AggressiveExplore:** Higher and aggressive placer effort in detail placement and post-placement optimization goals.

6.2 FPGA Platforms

The accelerators of this work were implemented on two FPGA platforms. The first one is the **ZCU102** evaluation board and the second one is **F1 instance** on Amazon Web Services (AWS) Elastic Compute Cloud (EC2), which was the main target of this thesis. Both two accelerators passed the synthesis phase and functional correctness on these two platforms and were evaluated according to their resource usage.

6.2.1 ZCU102

ZCU102 evaluation board [51] is targeted by *xczu9eg-ffvb1156-2-e* FPGA and includes 4GB of DDR4 for the Processing System, 512MB of DDR4 for the Programmable Logic, $2 \times 64MB$ Quad-SPI Flash, and an SDIO card interface. Its CPU frequency is 1200(MHz) while feeds six clock domains with 75, 100, 150, 200, 300, 400, 600 MHz.

TABLE 6.1: Platform's Software Components

System Configuration	Domain Details
a53_linux	linux on cortex-a53
a53_standalone	standalone on psu_cortexa53_0
r5_standalone	standalone on psu_cortexr5_0

TABLE 6.2: Used Resources on ZCU102.

	II	Frequency	BRAMs	DSPs	LUTs	REGs
TOTAL			4,320	6,840	2,364,480	1,182,240
SumGAMMAPROT	80	200 MHz	1.81%	3.49%	6.86%	5.91%
PLF	160	100 MHz	10.64%	17.54%	61.59%	25.67%

6.2.2 AWS EC2 F1 Instance

Amazon Elastic Compute Cloud (Amazon EC2) [52] is a web service that provides secure, resizable compute capacity in the cloud. It is designed to make web-scale cloud computing easier for developers. It contains a lot of features and platforms to fulfill all designers' demands. Some of them are FPGA instances, GPU compute instances, GPU graphics instances, high I/O instances, and more. It is also offered to customers private data storage, the capability of pausing, and resuming their instances and also unlimited times of reconfigure.

For this project, the f1.2xlarge instance was chosen to implement the designs. This platform targets the Virtex UltraScale+ AWS VU9P F1 Acceleration Development Board with VU13P. This high-performance acceleration platform features four channels of DDR4-2400 DIMMs (64GB), the expanded partial reconfiguration flow for high fabric resource availability, and Xilinx DMA Subsystem for PCI Express with PCIe Gen3 x16 connectivity. Its runtime is OpenCL. Moreover, its CPU frequency is 2.3GHz on basic mode and it can reach the peak of 2.7GHz on turbo mode. It also feeds two main clock domains with 250 and 500 MHz while there are more substitutes.

Both accelerators passed the synthesis phase and their usage of total resources according to their initiation interval and clock frequency are presented in the table 6.3.

TABLE 6.3: Used Resources on AWS F1 instance.

	II	Frequency	BRAMs	DSPs	LUTs	REGs
TOTAL			4,320	6,840	2,364,480	1,182,240
SumGAMMAPROT	10	350 MHz	1.26%	1.17%	1.78%	1.71%
PLF	20	100 MHz	19.25%	50.48%	85,35%	53.05%
	40	100 MHz	14.79%	25.24%	49.57%	36.28%
	80	150 MHz	8.09%	12.62%	39.79%	25.62%
	160	150 MHz	4.74%	6.47%	19.31%	8.87%

6.3 OpenCL Host

OpenCL (Open Computing Language) is a framework for writing programs that execute across heterogeneous platforms consisting of central processing units (CPUs), graphics processing units (GPUs), digital signal processors (DSPs), field-programmable gate arrays (FPGAs), and other processors or hardware accelerators. OpenCL specifies programming languages (based on C99 and C++11) for programming these devices and application programming interfaces (APIs) to control the platform and execute programs on the compute devices. OpenCL provides a standard interface for parallel computing using task- and data-based parallelism.

The OpenCL API defines the memory model to be used by all applications that comply with the standard. This hierarchical representation of memory is common across all vendors and can be applied to any OpenCL application. The vendor is responsible for defining how the OpenCL memory model maps to specific hardware. The OpenCL memory model is shown overlaid onto the OpenCL device model in the following figure :

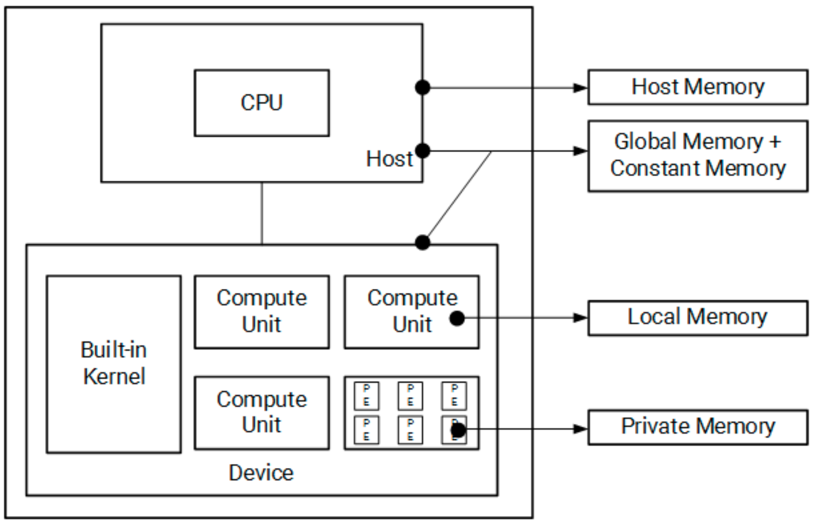


FIGURE 6.1: OpenCL Memory Model

For devices using an FPGA, the physical mapping of the OpenCL memory model is the following:

1. Host memory is any memory connected to the host processor only.
2. Global and constant memories are any memory that is connected to the FPGA. These are usually memory chips that are physically connected to the FPGA. The host processor has access to these memory banks through infrastructure in the FPGA base device.
3. Local memory is memory inside of the FPGA. This memory is typically implemented using block RAM elements in the FPGA fabric.
4. Private memory is memory inside of the FPGA. This memory is typically implemented using registers in the FPGA fabric to minimize latency to the compute data path in the processing element.

Subsequently, the connected Host with the above Kernels was implemented by the following steps. First of all, the host is responsible for the allocation and deallocation of buffers in **Global Memory** and there is a handshake between host and device over control of the data stored in this memory. So it's mandatory to seek and get information successfully about the used device. Then, it's easy to create the program and the kernel by finding the appropriate binary file and kernel core. Passing successfully the above steps, it is time to create Buffers to import our data. There must be a correct allocation of the data and their data sizes, otherwise, it might cause a data loss or misuse of memory. Moreover, if multi-memory controllers are used, it is also necessary to assign buffers' flags with DDR4 banks. After the correct allocation of Buffers then it's time to **enqueueMigratMemObjects** from the host to global memory. The next step is the set up of the arguments of the kernel core followed by kernel's **enqueueTask**. By the time the kernel is launched to process the data, the host loses access rights to the buffer in global memory, the device takes over and is capable of reading and writing data from the global memory until the kernels' execution is completed. Upon completion of the operations associated with the kernel, the device conveys the control of the global memory buffer back to the host processor. Finally, as host obtains again the control of the buffer, it can read and write data from/to the buffer, transfer data back to the host memory, and deallocate the used buffer.

Chapter 7

Results

In this chapter, the results that came from the profiling and the final implementation are presented. There is also a reference on how the datasets were generated and their role in the different executions of RAxML.

7.1 Datasets

As it was mentioned in a previous Chapter, RAxML uses mostly Phylip(.phy) files as input datasets. These datasets can be found on different surveys and scientific databases if users need real data. Otherwise, there are many programs that generate such files. In our case, in order to generate the inputs datasets, **MS**, a program for generating samples under neutral models by Richard R. Hudson [53] was used. The reason for choosing it was that, ms offers some extra parameters, such as mutation rate, that can adapt the random data in a way that some taxa can have similarities between them. This helps RAxML to evaluate and compute some data that are not completely foreign.

The datasets that were generated for the profiling of the software execution times, included 10, 25, 50, and 100 taxa each one with different alignment patterns equal to powers of 2 (1024,2048,....,262144). This choice was made as we wanted to test random big values and also to evaluate the performance of RAxML with the use of a uniform range of data sets. Moreover, the mutation rate for all these datasets was 1 and 10. We chose these values for the mutation rate because, as this value gets bigger, the taxa become less foreign between them. Low values of mutation rate make taxa quite similar and can also reduce the alignment patterns by a factor of 1 to 40 percent of the initial number. This fact is met when the alignment factor is low and the taxa are few.

7.2 Software Performance

Our implementation was compared with the timings obtained by running the software (RAxML) on a server with the specifications shown on table 7.1:

CPU :	Intel(R) Xeon(R) CPU E5-2630 v4 @ 2.20 GHz
Architecture :	x86_64
Thread(s) / Core :	2
Core(s) / Socket :	10
Socket(s) :	2
RAM :	251GB

TABLE 7.1: Server's hardware specifications

On the following results, the total execution time of RAxML and the execution time of functions SumGAMMAPROT and NewViewGTRGAMMAPROT are shown. It must be mentioned that the timings below are the accumulation of the total calls of these functions during the whole execution of RAxML. The titles of the figures below 7.1 - 7.9, state the length of a taxon used for each execution. In addition, all presented timings correspond to a mutation rate equal to 10 because there is no coherence of similar amino acids of the different taxa, a fact that makes all alignment patterns available and there are no cutbacks that bring an actual accuracy.

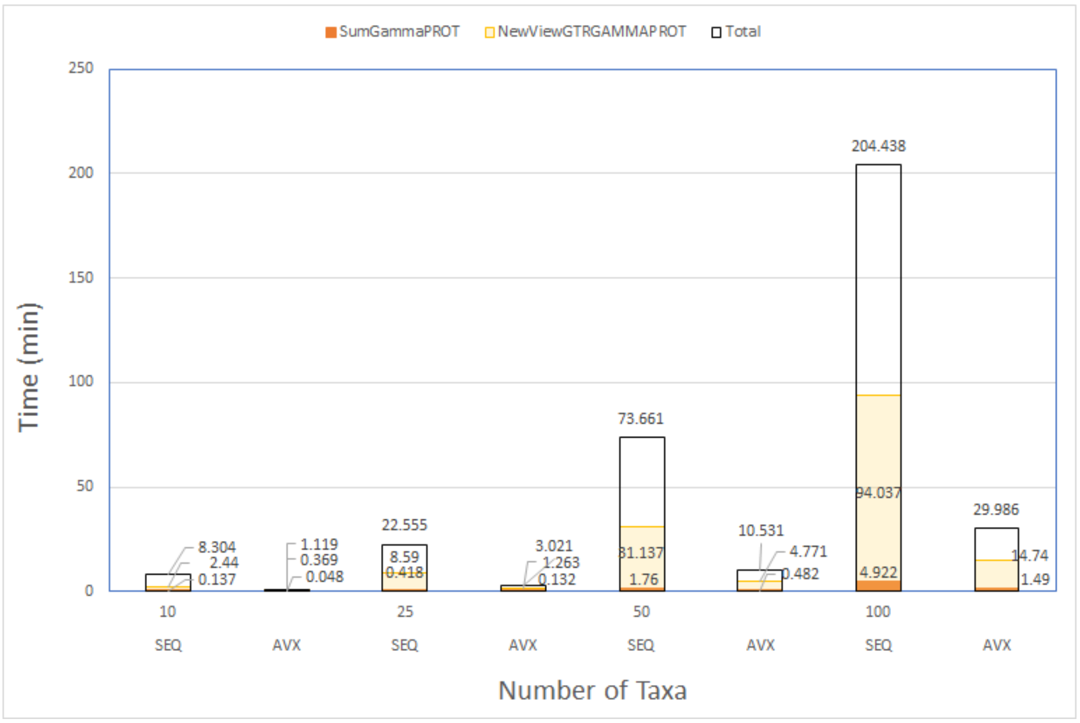


FIGURE 7.1: 1.024 Alignment Patterns-SW

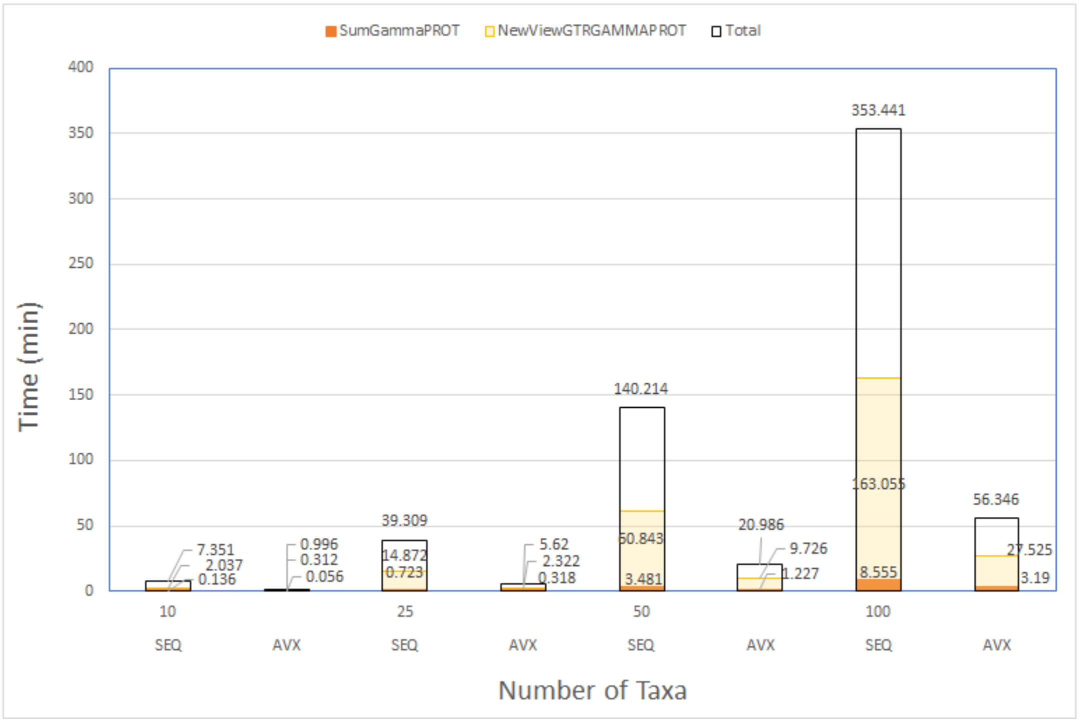


FIGURE 7.2: 2.048 Alignment Patterns-SW

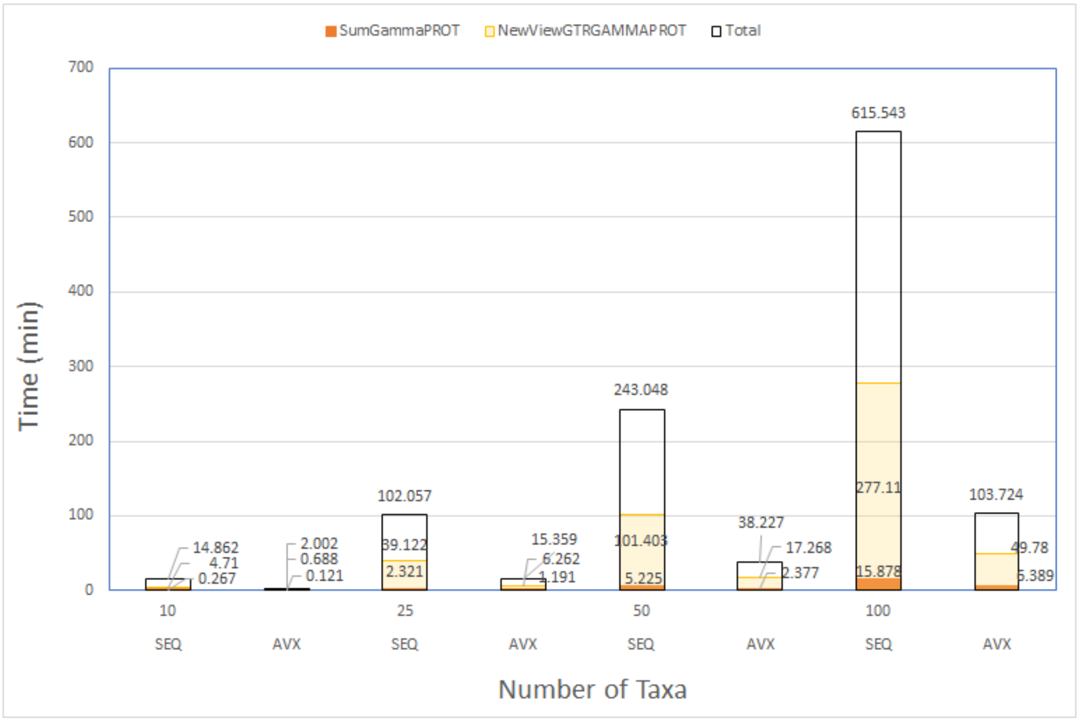


FIGURE 7.3: 4.096 Alignment Patterns-SW

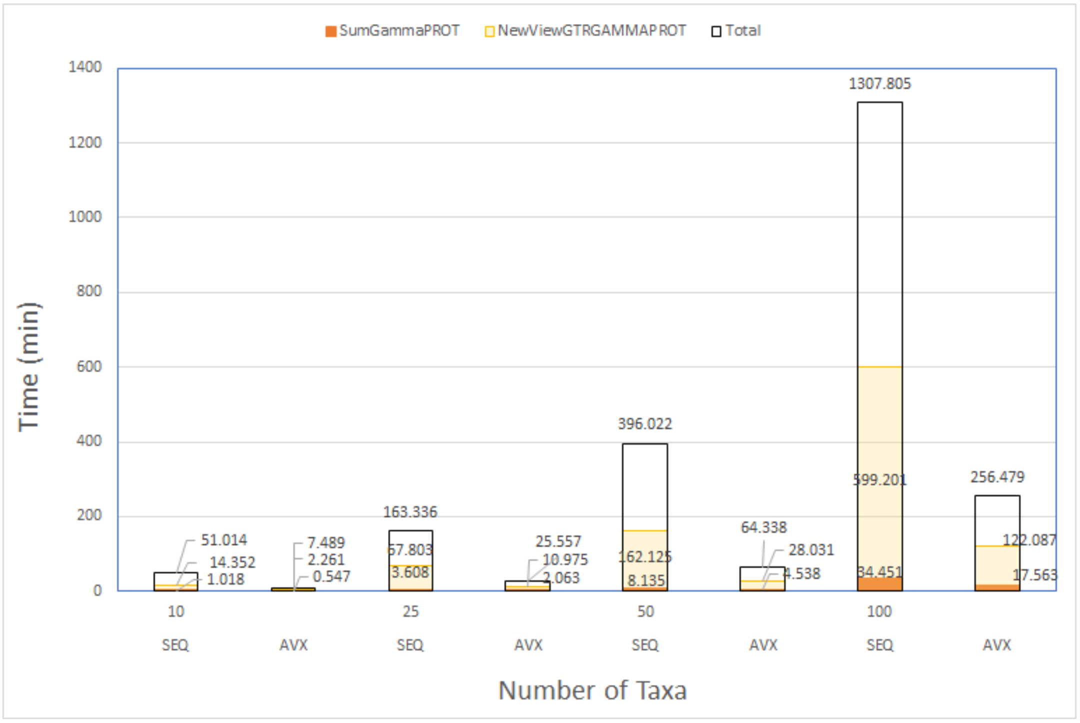


FIGURE 7.4: 8.192 Alignment Patterns-SW

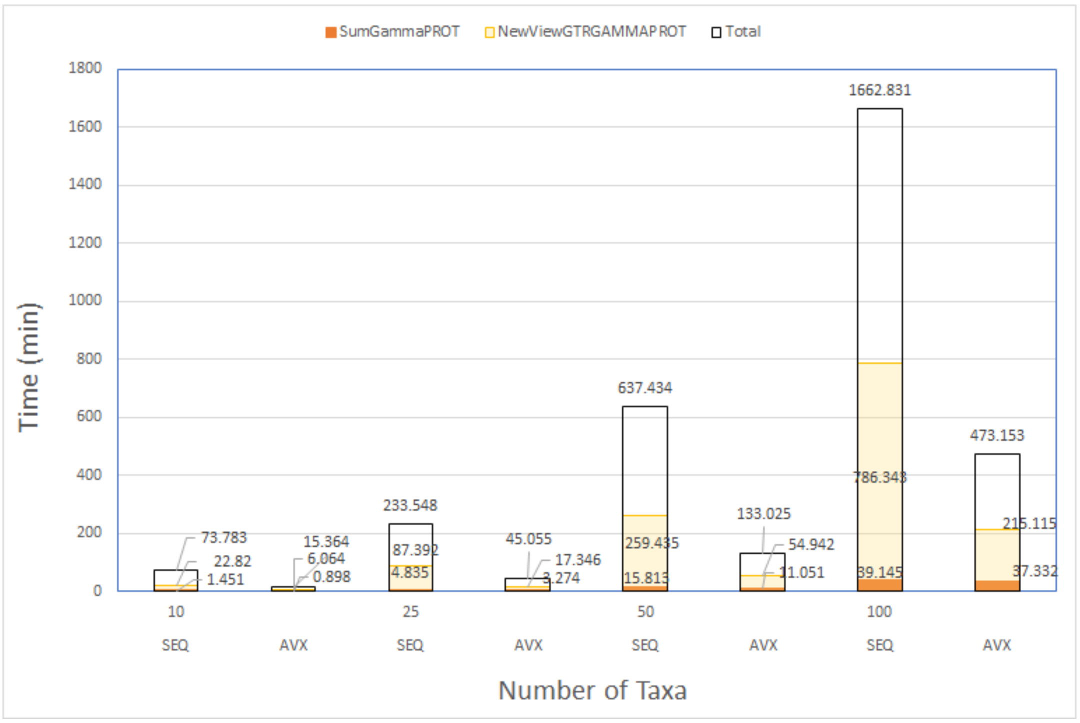


FIGURE 7.5: 16.384 Alignment Patterns-SW

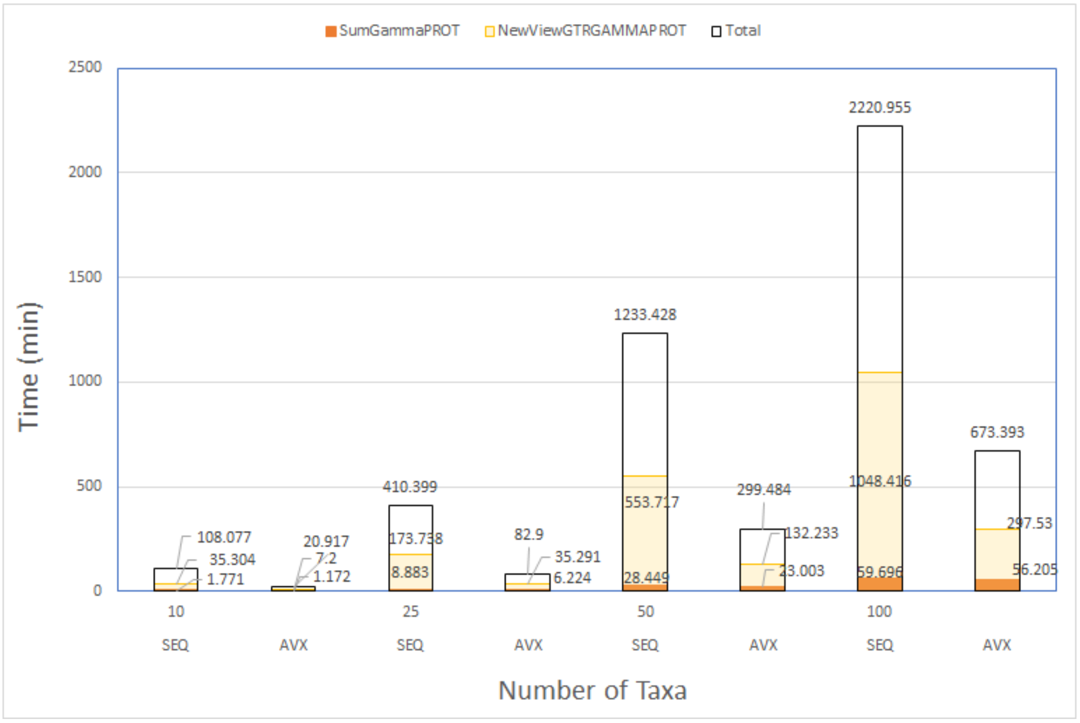


FIGURE 7.6: 32.768 Alignment Patterns-SW

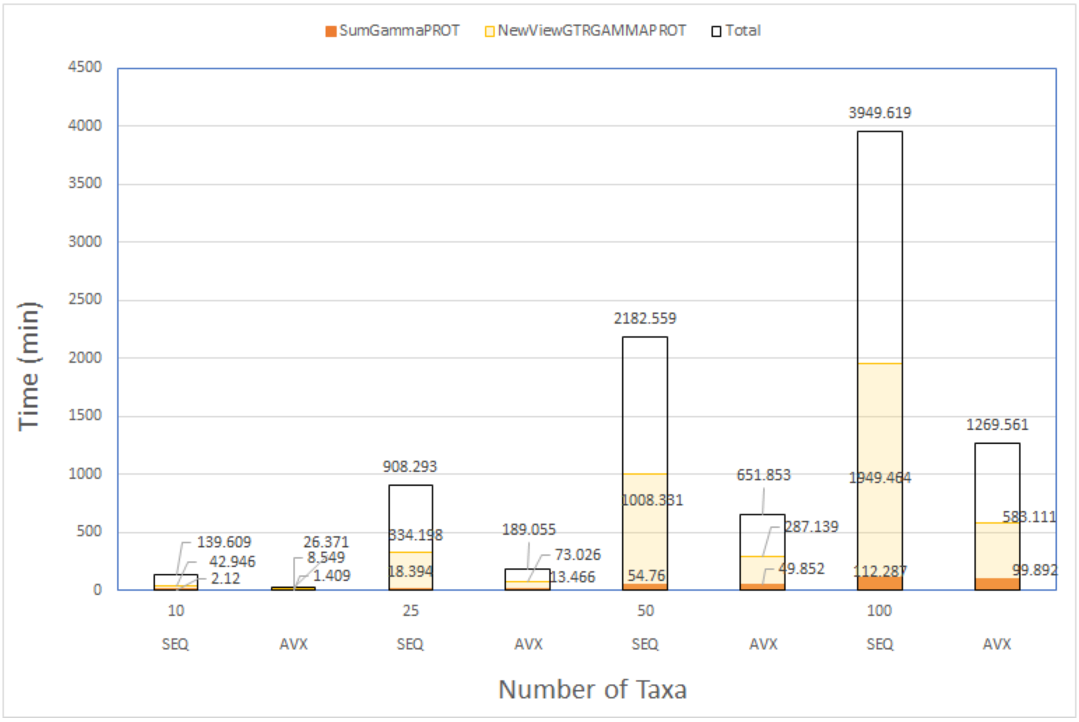


FIGURE 7.7: 65.536 Alignment Patterns-SW

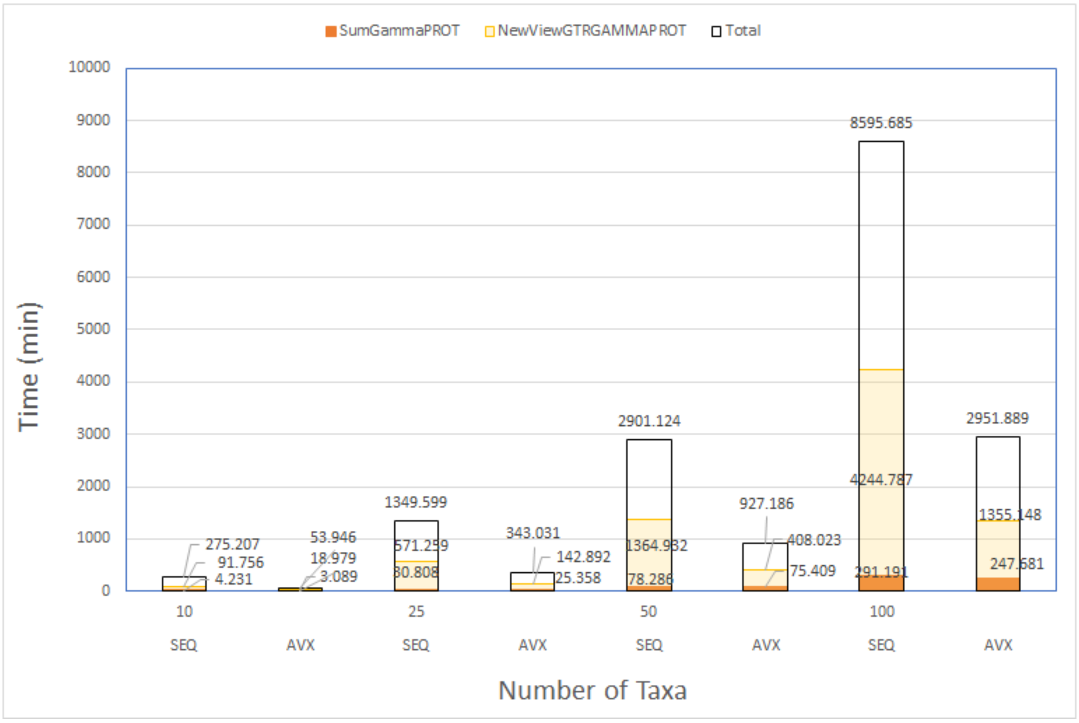


FIGURE 7.8: 131.072 Alignment Patterns-SW

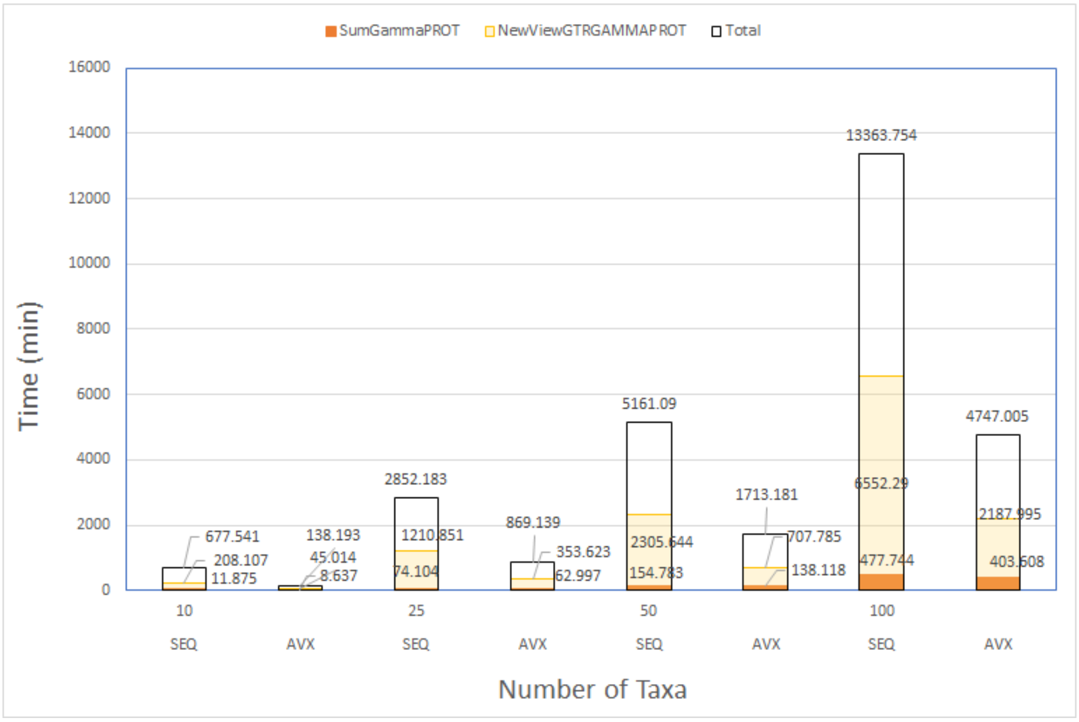


FIGURE 7.9: 262.144 Alignment Patterns-SW

Comparing the above results, it can be observed that both the total execution time and the functions' time increase proportionally (approximately 2x) with the increase of the number of alignment patterns. In addition, a similar reaction is observed when the number of taxa increases and that happens with a bigger rate (>2x). However, the audit findings of the above runs are that, as the already implemented AVX version of RAxML offers a significant acceleration, it is still consuming too much time when surpassing a threshold of 32K alignment patterns and a number of taxa 100. This fact leads us to have as a threshold the execution time of the AVX implementation and present the performance of hardware according to it.

7.3 Hardware Performance

7.3.1 ZCU102

The hardware kernel of function SumGAMMAPROT was the first one tested on targeted platform ZCU102 with II equal to 80 and a clock frequency of 200 MHz.

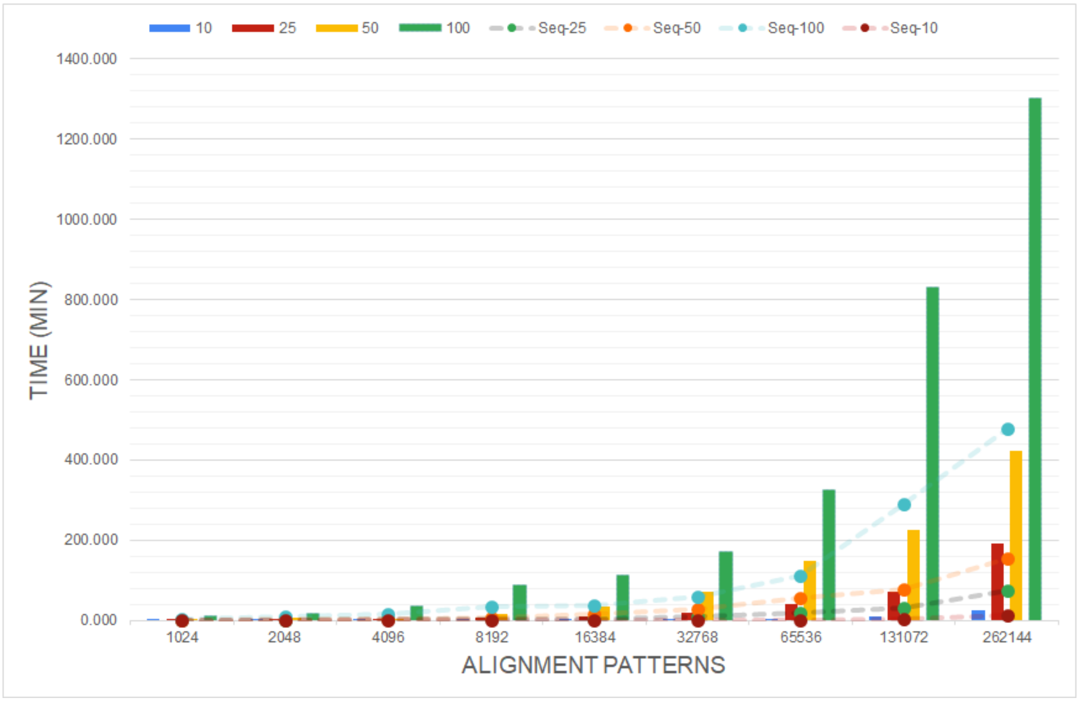


FIGURE 7.10: Execution Times Of SumGAMMAPROT Compared with Sequential Version

	1024	2048	4096	8192	16384	32768	65536	131072	262144
10	x0.49	x0.56	x0.63	x0.59	x0.58	x0.54	x0.54	x0.49	x0.50
25	x0.52	x0.53	x0.54	x0.52	x0.52	x0.49	x0.46	x0.43	x0.39
50	x0.50	x0.50	x0.49	x0.47	x0.47	x0.40	x0.37	x0.35	x0.37
100	x0.46	x0.45	x0.43	x0.38	x0.35	x0.35	x0.35	x0.35	x0.37

TABLE 7.2: Performance of SumGAMMAPROT Hardware Kernel Compared with Sequential on ZCU102

Analyzing the above results and performance, we distinguish that this kernel cannot accelerate the initial function. The performance of this kernel reaches half of the execution time on the CPU. That is caused due to the time-consuming setup of the DMA controller and the transfer of data, times that reach 40% of the total execution time. Theoretically using the equation

$$Time = N \times II \times ClkPeriod \tag{7.1}$$

, we calculated that achieving an ideal transfer of data we could achieve an approximate 2x speedup of the initial function in case that the number of Alignment Patterns (N) is bigger than 500K, a value N that is unusual compared with real data sets. So the conclusion is that this function is better to be used by the CPU while the processing load is low and can be handled by CPU's resources and secondly there is no time waste during the transfer of data.

The hardware kernel of function NewViewGTRGAMMAPROT with II equal to 160, clock frequency 100 MHz ran on ZCU102 and gave the following results on figures 7.11,7.12 and tables 7.3, 7.4:

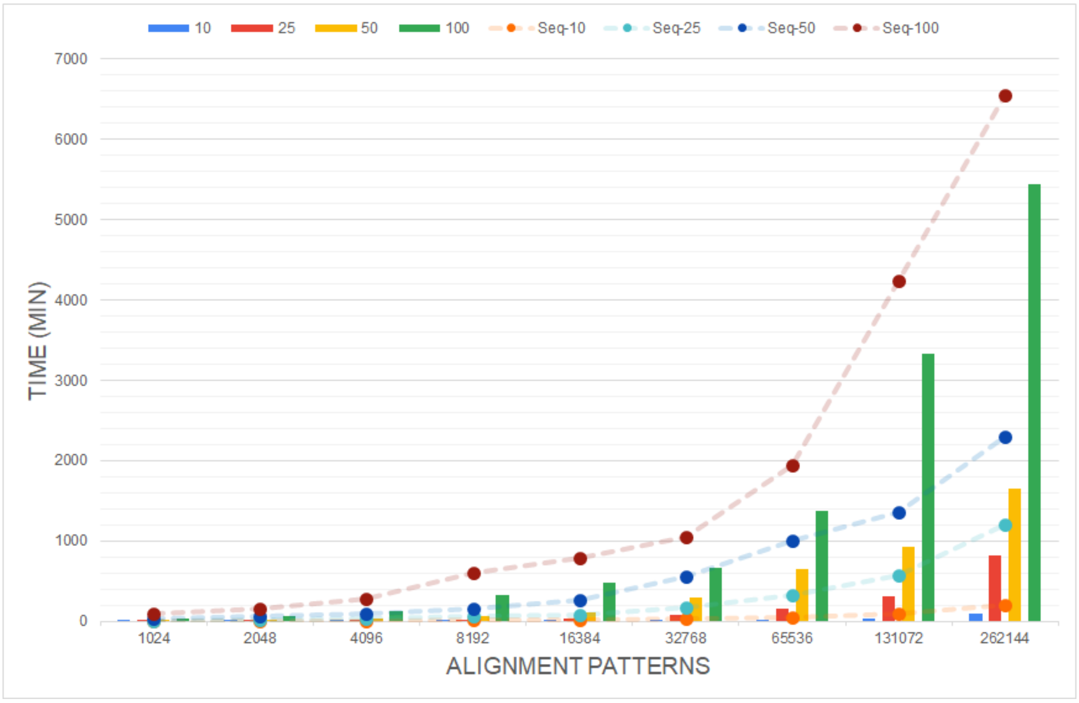


FIGURE 7.11: Execution Times Of NewViewGTRGAMMAPROT Compared with Sequential Version

	1024	2048	4096	8192	16384	32768	65536	131072	262144
10	x2.83	x2.84	x3.01	x2.80	x2.80	x2.68	x2.68	x2.42	x2.17
25	x2.91	x2.78	x2.71	x2.63	x2.51	x2.33	x2.11	x1.83	x1.49
50	x2.78	x2.65	x2.46	x2.36	x2.22	x1.90	x1.57	x1.48	x1.39
100	x2.58	x2.37	x2.18	x1.83	x1.66	x1.57	x1.42	x1.27	x1.20

TABLE 7.3: Performance of NewViewGTRGAMMAPROT Hardware Kernel Compared with Sequential on ZCU102

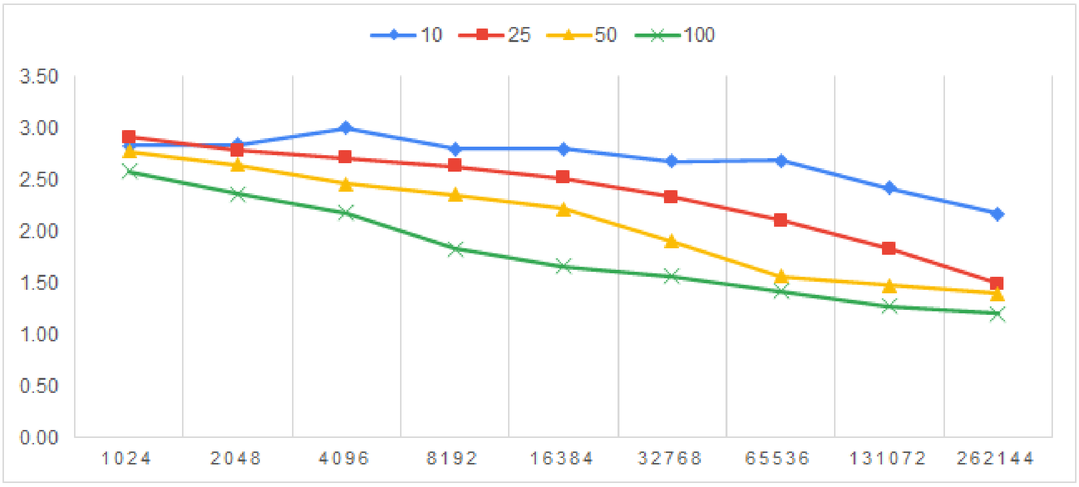


FIGURE 7.12: Performance of Speedup for NewViewGTRGAMMAPROT Compared With Sequential Version-ZCU102

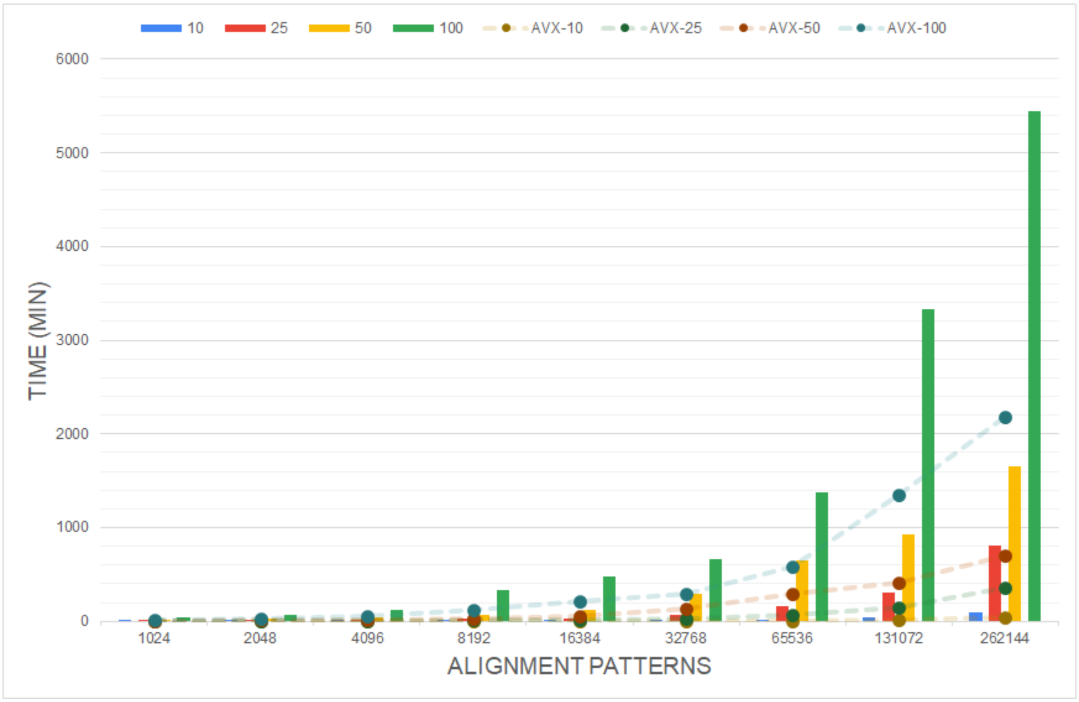


FIGURE 7.13: Execution Times Of NewViewGTRGAMMAPROT Compared with AVX Version

	1024	**2048**	**4096**	**8192**	**16384**	**32768**	**65536**	**131072**	**262144**
10	x0.43	x0.44	x0.44	x0.44	x0.74	x0.55	x0.53	x0.50	x0.47
25	x0.43	x0.43	x0.43	x0.43	x0.50	x0.47	x0.46	x0.46	x0.43
50	x0.43	x0.42	x0.42	x0.41	x0.47	x0.45	x0.45	x0.44	x0.43
100	x0.40	x0.40	x0.39	x0.37	x0.45	x0.44	x0.42	x0.41	x0.40

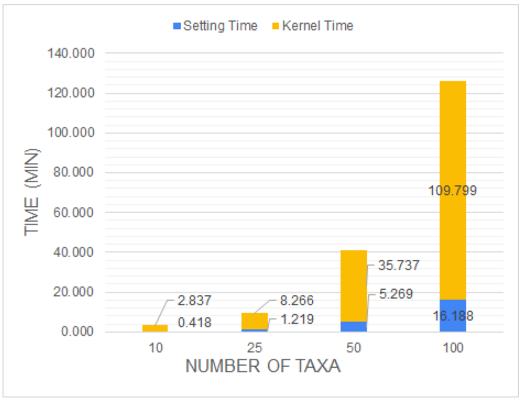
TABLE 7.4: Performance of NewViewGTRGAMMAPROT Hardware Kernel Compared with AVX on ZCU102

Analyzing the above results and performance, we note that this kernel brings an acceleration of the initial sequential function but it cannot even reach its AVX version. Regarding the acceleration that is achieved, there is a 2.5-3x speedup on 1K alignment patterns, and as they increase the speedup decreases. When there is a low value of alignment patterns, there is a good trade between DMA setup and processing time. This means that the load of the initial function with these parameters can be distributed to our design and bring acceleration. Another observation is that, while the number of taxa increases, the total speedup decreases. This is caused because there is a loss of time during the supernumerary recurrent calls of the DMA and kernel through the whole execution of RAxML. Finally, we calculated theoretically, like in the previous kernel, this performance provided we have an ideal DMA, and it showed that there is a similar and equal performance with the experimental one.

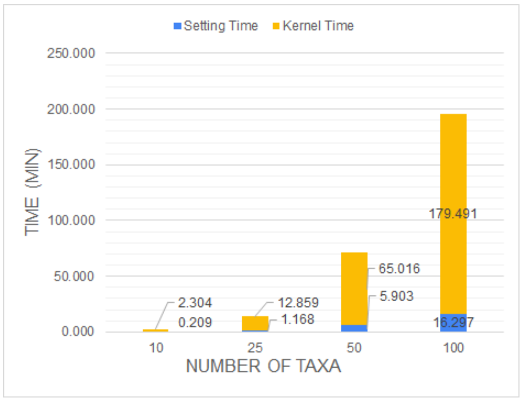
It must be noted that it was impossible to predict and calculate the performance of the designs under the usage of bigger data because RAxML has uncertainty on how many times it can call these functions in order to produce the results and the total execution time was tremendous.

7.3.2 AWS F1 Instance

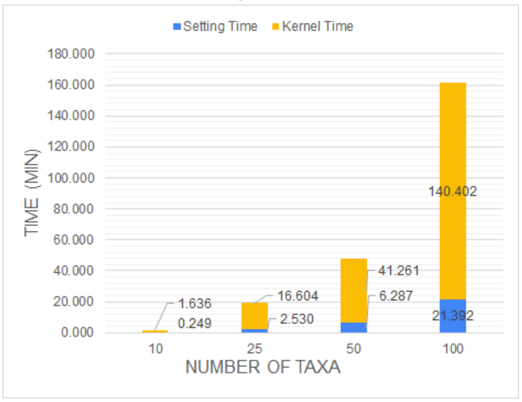
The first kernel that tested on AWS F1 Instance was SumGAMMAPROT. Achieving a clock with 350 MHz and the optimal II equal to 10, the results of execution times of the kernel and the setup times of data (Transfer Device to Host and via versa, Setting Arguments) are shown in the below figures 7.14, 7.15:



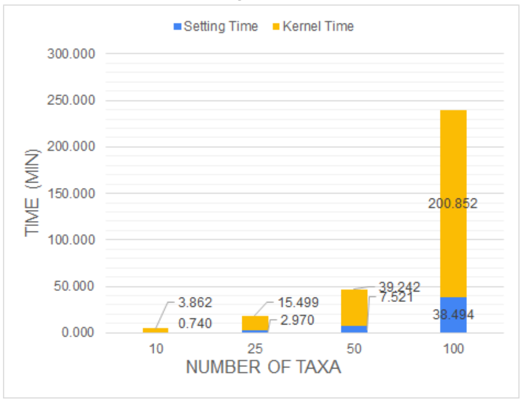
(A) 1024 Alignment Patterns



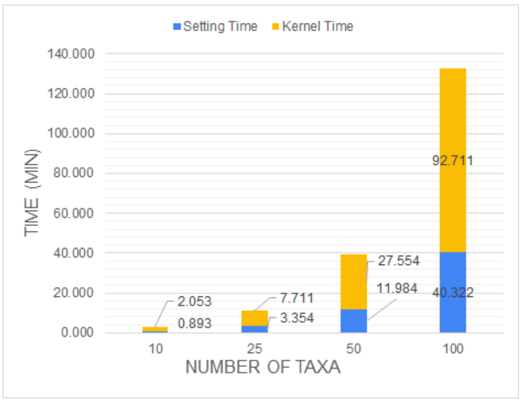
(B) 2048 Alignment Patterns



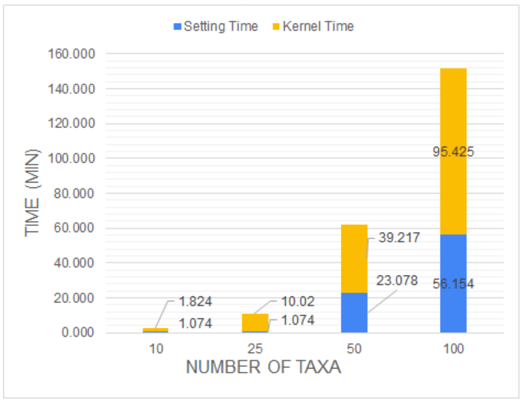
(C) 4096 Alignment Patterns



(D) 8192 Alignment Patterns

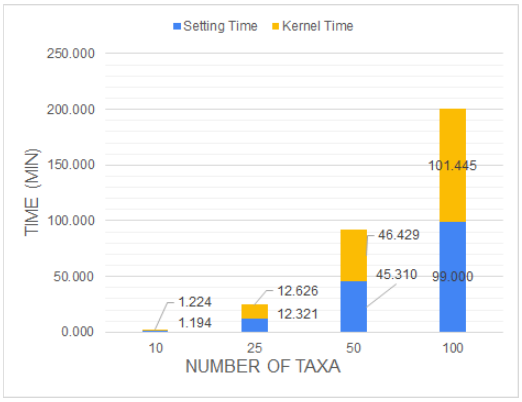


(E) 16384 Alignment Patterns

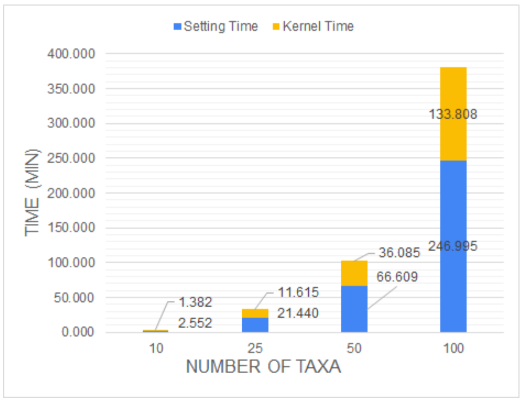


(F) 32768 Alignment Patterns

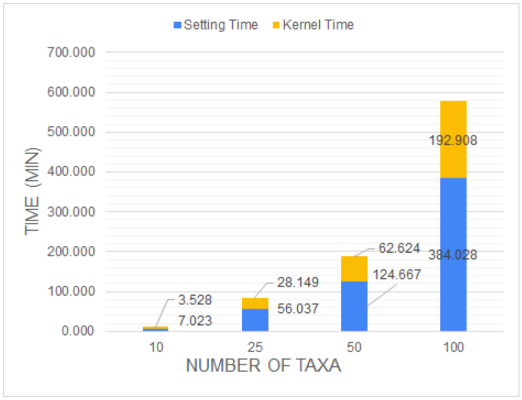
FIGURE 7.14: Execution Times Of SumGAMMAPROT kernel On AWS F1 (1)



(A) 65536 Alignment Patterns



(B) 131072 Alignment Patterns



(C) 262144 Alignment Patterns

FIGURE 7.15: Execution Times Of SumGAMMAPROT kernel On AWS F1 (2)

In figure 7.16 we present the total execution times(Set up Time + Kernel Time) on the targeted platform for all these alignment patterns on the x-axis and with a different number of taxa on each data set. The dotted lines represent the sequential execution times of these data sets on the selected server. There is no comparison with AVX running times because there is not an AVX version of this function. On the table 7.5 the total performance of hardware is also shown.

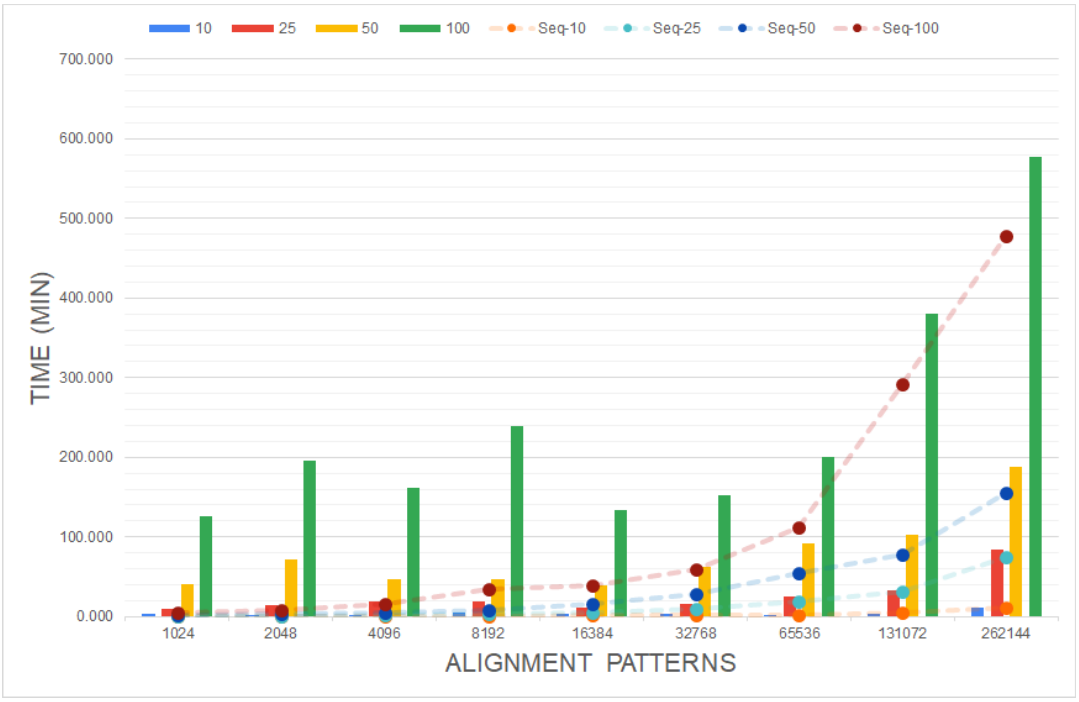


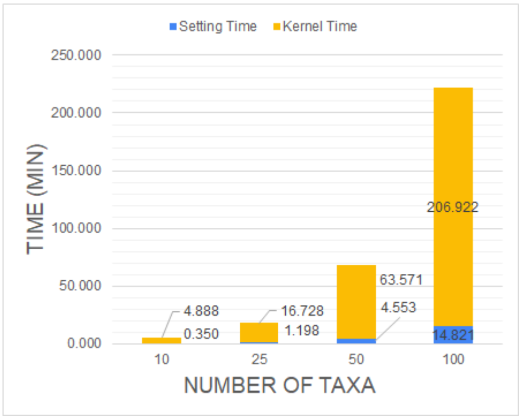
FIGURE 7.16: Total Execution Time Of SumGAMMAPROT Kernel on AWS F1

	1024	**2048**	**4096**	**8192**	**16384**	**32768**	**65536**	**131072**	**262144**
10	x0.04	x0.05	x0.14	x0.22	x0.49	x0.61	x0.88	x1.08	x1.13
25	x0.04	x0.05	x0.12	x0.20	x0.44	x0.56	x0.74	x0.93	x0.88
50	x0.04	x0.05	x0.11	x0.17	x0.40	x0.46	x0.60	x0.76	x0.83
100	x0.04	x0.04	x0.10	x0.14	x0.29	x0.39	x0.56	x0.76	x0.83

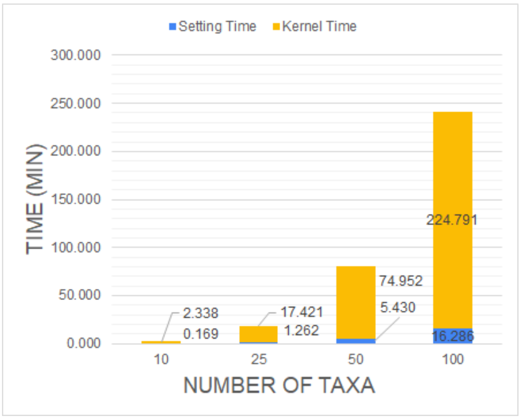
TABLE 7.5: Performance of SumGAMMAPROT Hardware Kernel On AWS F1

It is observed that although we achieved the optimal requirements of the kernel of function SumGAMMAPROT, we could not achieve an actual speedup of it. As it is shown on 7.16, the executing times on hardware are approaching the software ones without surpassing them. The only case that it can exceed the software executing time is when the data sets contain sequences with more than 500K alignment patterns, a value that is too big according to real data used by scientists. The fact that there is not a actual acceleration is revealed on figures 7.14, 7.15 and 7.16. Although the execution time of the Kernel decreases with the growth of the number of alignment patterns, the setup timings increase. This fact means that there is a slow transfer of the data to the device while the kernel itself processes the data more quickly.

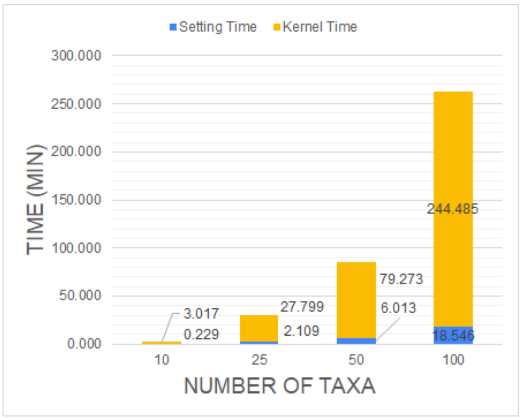
The second kernel that tested on AWS F1 Instance was the one that corresponds to the function of NewViewGTRGAMMAPROT. Achieving a clock of 150 MHz and the II equal to 160, the results of execution times of the kernel and the setup times of data (Transfer Device to Host and via versa, Setting Arguments) are figured below 7.17, 7.18:



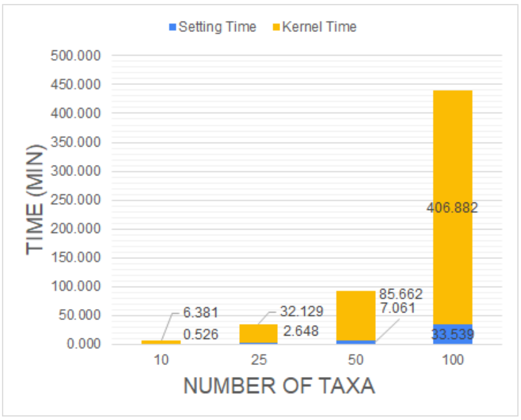
(A) 1024 Alignment Patterns



(B) 2048 Alignment Patterns

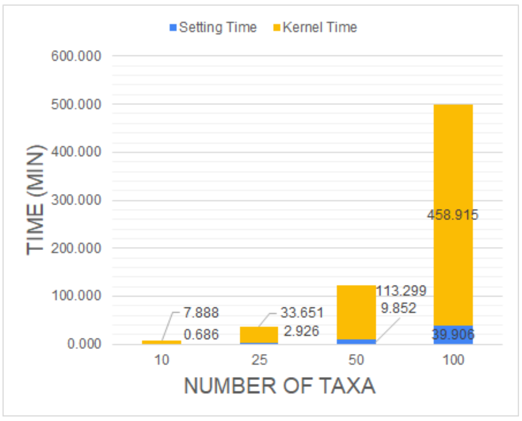


(C) 4096 Alignment Patterns

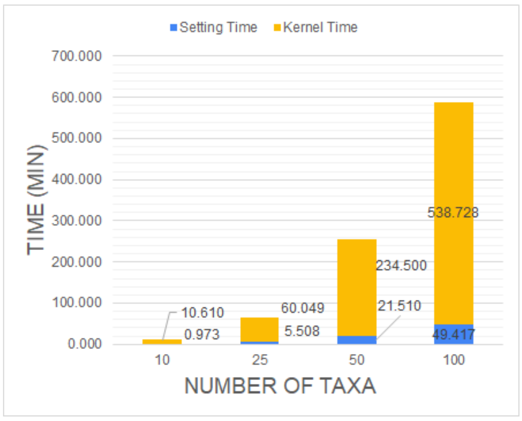


(D) 8192 Alignment Patterns

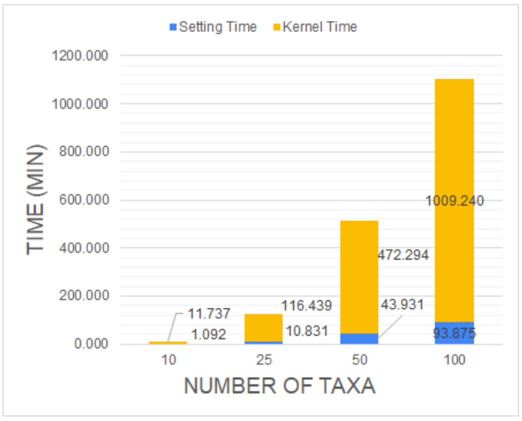
FIGURE 7.17: Execution Times Of NewViewGTRGAMMAPROT kernel On AWS F1 (1)



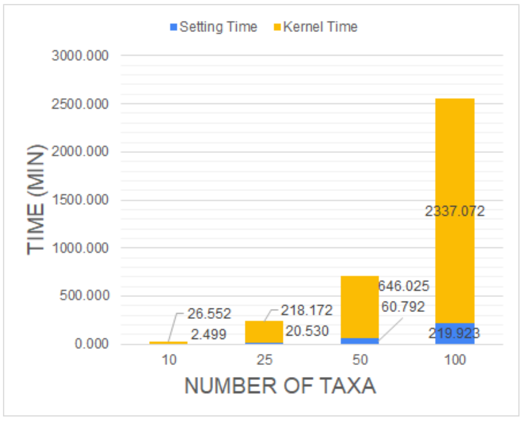
(A) 16384 Alignment Patterns



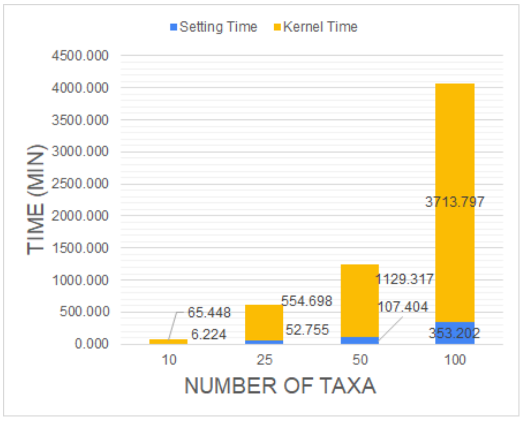
(B) 32768 Alignment Patterns



(C) 65536 Alignment Patterns



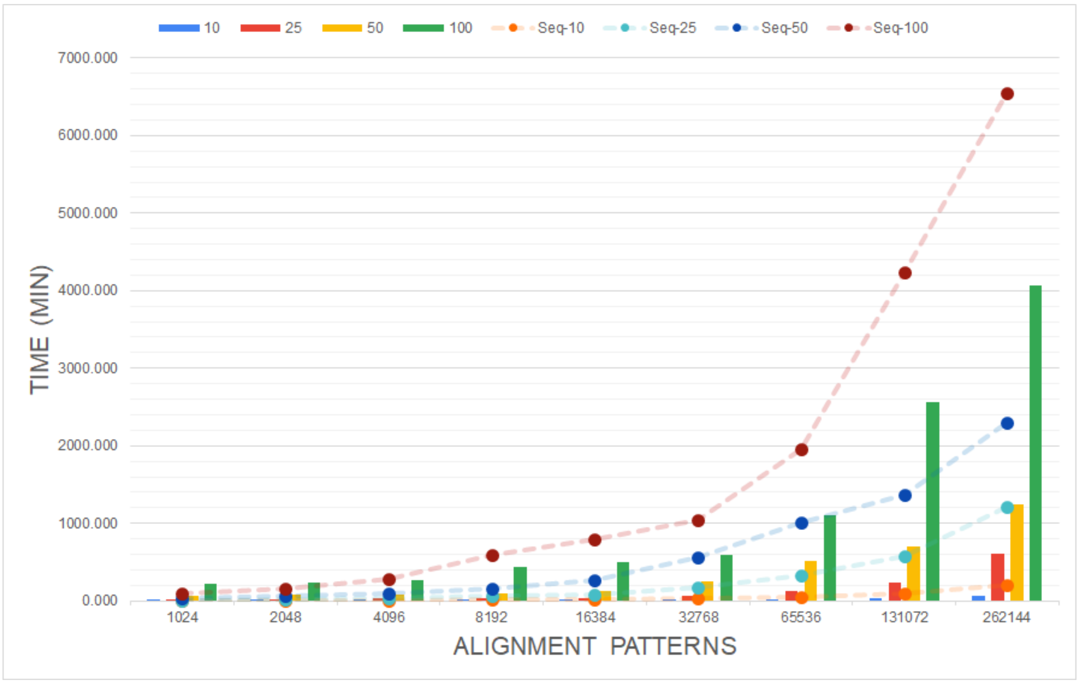
(D) 131072 Alignment Patterns



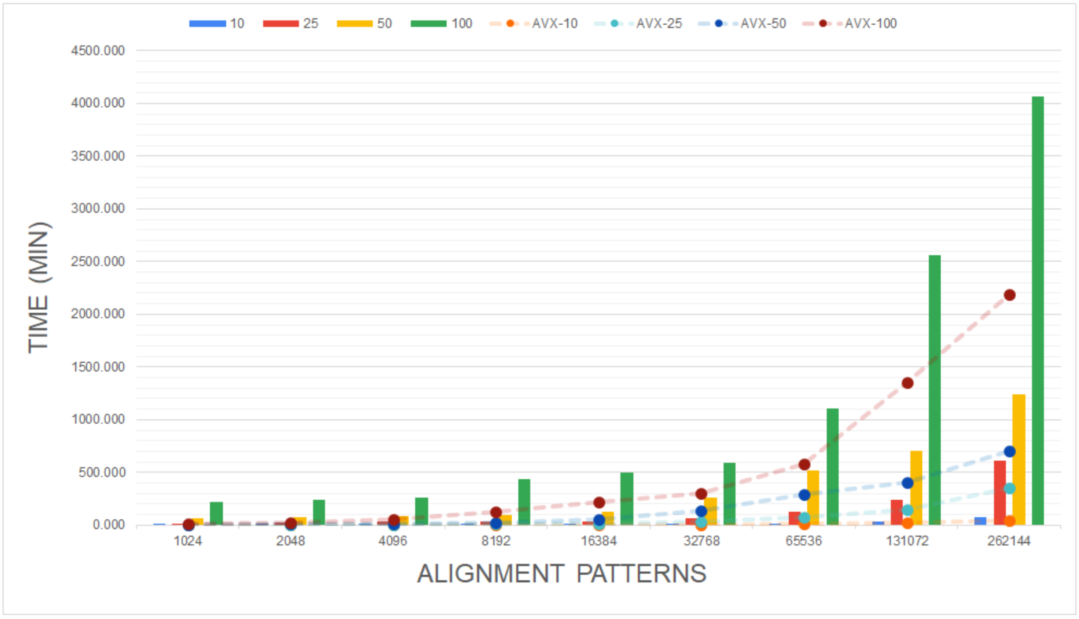
(E) 262144 Alignment Patterns

FIGURE 7.18: Execution Times Of NewViewGTRGAMMAPROT kernel On AWS F1 (2)

In figure 7.19a we present the total execution times(Set up Time + Kernel Time) on the targeted platform for all these alignment patterns on the x-axis and with a different number of taxa on each data set. The dotted lines represent the sequential execution times of these data sets on the selected server. In figure 7.19b the dotted lines represent the AVX version execution times of these data sets on the selected server. Moreover on tables 7.6, 7.7 the detailed performance of hardware kernel are presented.



(A) Hardware - Sequential Version



(B) Hardware - AVX Version

FIGURE 7.19: Execution Times Of NewViewGTRGAMMAPROT Kernel

	1024	**2048**	**4096**	**8192**	**16384**	**32768**	**65536**	**131072**	**262144**
10	x0.47	x0.81	x1.45	x2.08	x2.66	x3.05	x3.35	x3.16	x2.90
25	x0.48	x0.80	x1.31	x1.95	x2.39	x2.65	x2.63	x2.39	x1.99
50	x0.46	x0.76	x1.19	x1.75	x2.11	x2.16	x1.95	x1.93	x1.86
100	x0.42	x0.68	x1.05	x1.36	x1.58	x1.78	x1.77	x1.66	x1.61

TABLE 7.6: Performance of NewViewGTRGAMMAPROT Hardware Kernel Compared With Sequential Version

	1024	**2048**	**4096**	**8192**	**16384**	**32768**	**65536**	**131072**	**262144**
10	x0.07	x0.12	x0.21	x0.33	x0.71	x0.62	x0.67	x0.65	x0.63
25	x0.07	x0.12	x0.21	x0.32	x0.47	x0.54	x0.57	x0.60	x0.58
50	x0.07	x0.12	x0.20	x0.30	x0.45	x0.52	x0.56	x0.58	x0.57
100	x0.07	x0.11	x0.19	x0.28	x0.43	x0.51	x0.53	x0.53	x0.54

TABLE 7.7: Performance of NewViewGTRGAMMAPROT Hardware Kernel Compared With AVX Version

While we analyze the above results, we observe that with the current Kernel, clock frequency 150 MHz and II equal to 160, we achieve an acceleration comparing with the sequential version of NewViewGTRGAMMAPROT. This is obvious when the alignment patterns surpass the 4K of alignment patterns. Starting with an acceleration factor of x1.13 on 4.096 alignment patterns and 100 taxa, we then brought off a factor of x3.18 on 262.144 alignment patterns and 10 taxa. As an observation, our kernel can bring significant acceleration using a range between 4K and 262K of alignment patterns. Out of this range, there is a minor or no acceleration, due to the big time of transfer of data and we can not benefit from it. In detail under the boundary of 4K, it fails to accelerate due to the slow process of data. On the other hand, up to the boundary of 262K, the transfer of data tends to consumes too much time on the processing part. The following figure 7.20 shows the graduated acceleration of the kernel function to the number of alignment patterns.

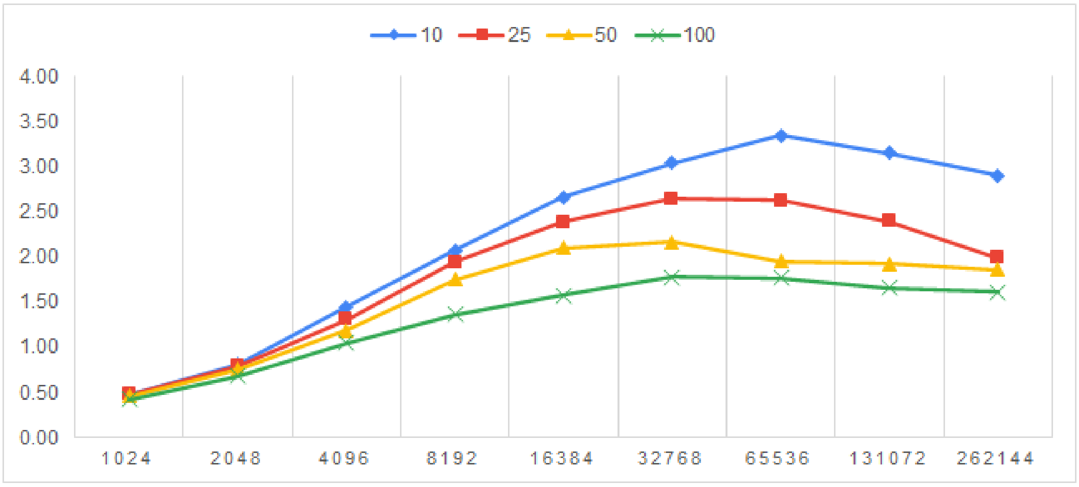


FIGURE 7.20: Performance of Speedup for NewViewGTRGAMMAPROT Compared With Sequential Version-AWS F1

Going into the comparison of the current kernel and AVX version, 7.19b and 7.7, it is observed that there is no acceleration. This fact happens because the architecture of AVX intrinsics is better than our kernel with the existing II and clock frequency.

7.4 Performance Model

As it is obvious through the study of the above results, our kernel cannot accelerate the function SumGAMMAPROT when it is standalone, because the transfer of the data to the kernel, requires more time than to process them.

On the other hand, the implemented kernel for NewViewGTRGAMMAPROT has successfully achieved an acceleration of the sequential version of the initial software function both on the ZCU102 and AWS F1. However, as we set as threshold the performance of the AVX version, there is a need to make some optimizations.

As we mentioned in previous chapters, the optimal II of our kernel on targeted platform AWS F1 is 10 while on ZCU102 is 40. That comes from computations when we factor the accumulated data size we transfer and the maximum or used bandwidth of the platform.

$$II = \frac{\#OfValuesPerDataPack \times SizeOfValue \times \#OfValuesPerSite}{MaximumBandwidth} \tag{7.2}$$

However, the initial target of this thesis was AWS F1 so the following results correspond to this platform and not on ZCU102. Moreover, the AWS f1 platform offers us more hardware resources and better specifications than ZCU102. Going forward and step by step, we achieved to reach the complete level of synthesis of the kernel for NewViewGTRGAMMAPROT, with II equal to 80, 40, 20 and with clock frequency 100 MHz each one. Following these timing features, we studied and designed a timing model that could simulate an approximate performance framework. The results are figured below:

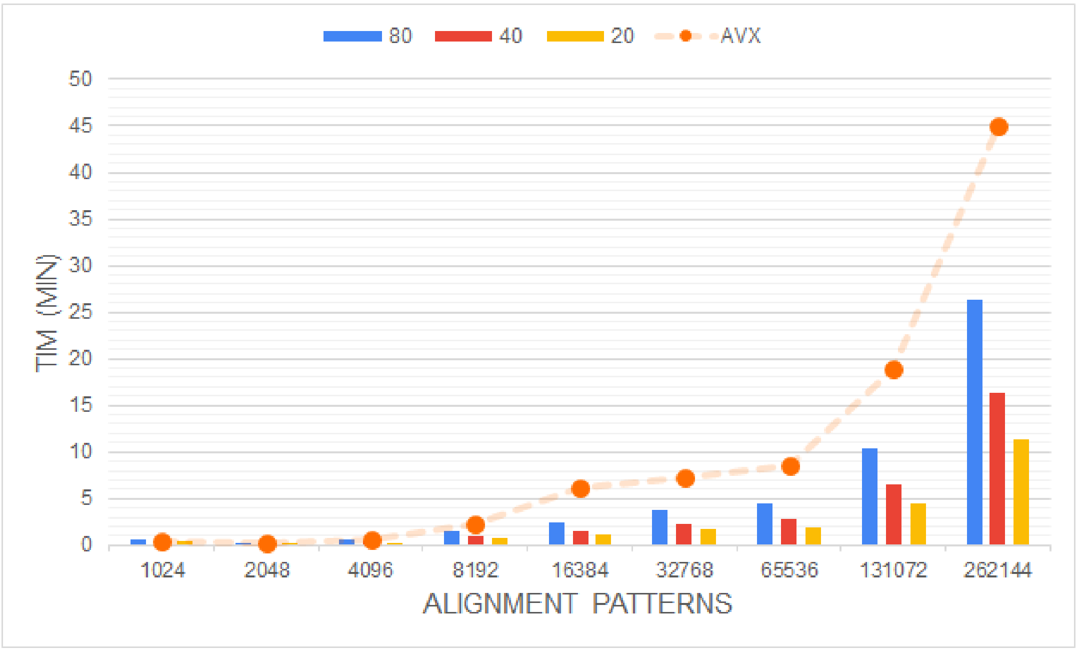


FIGURE 7.21: Model of the Kernels Performance using 10 taxa

	1024	2048	4096	8192	16384	32768	65536	131072	262144
80	x0.70	x0.98	x1.24	x1.41	x2.54	x1.93	x1.92	x1.82	x1.71
40	x0.84	x1.28	x1.75	x2.13	x3.94	x3.06	x3.09	x2.93	x2.77
20	x0.93	x1.51	x2.21	x2.84	x5.45	x4.33	x4.43	x4.23	x4.00

TABLE 7.8: Performance of Model of NewViewGTRGAMMAPROT Hardware Kernel Compared With AVX Version (10 Taxa)

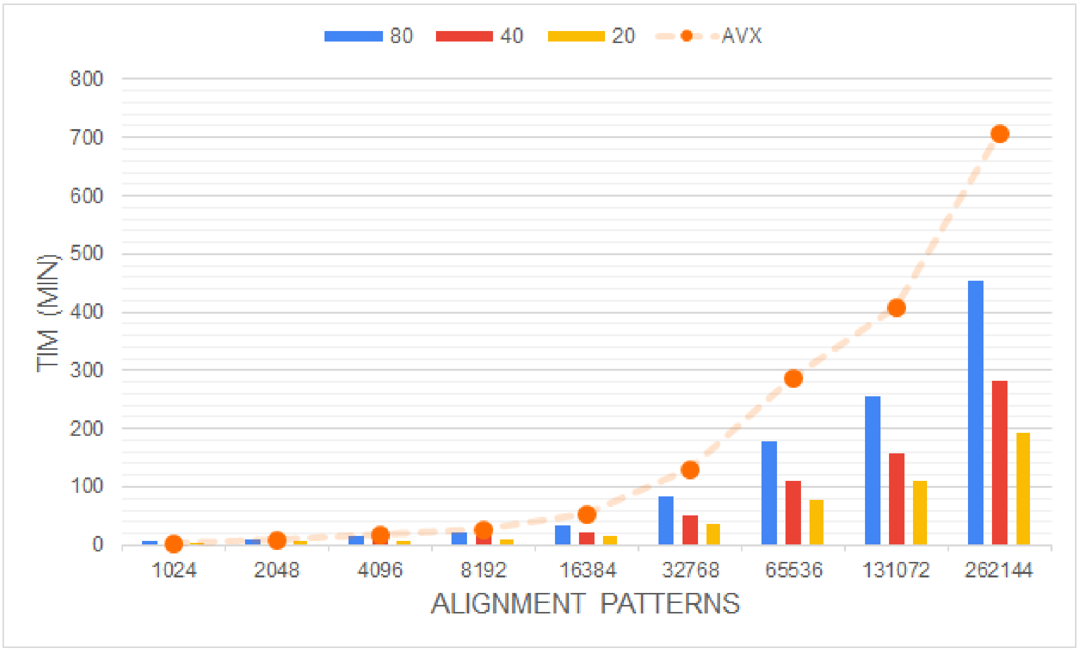


FIGURE 7.22: Model of the Kernels Performance using 25 taxa

	1024	2048	4096	8192	16384	32768	65536	131072	262144
80	x0.70	x0.98	x1.22	x1.36	x1.70	x1.67	x1.66	x1.67	x1.59
40	x0.84	x1.28	x1.73	x2.05	x2.64	x2.65	x2.66	x2.69	x2.56
20	x0.93	x1.51	x2.18	x2.74	x3.65	x3.75	x3.81	x3.88	x3.71

TABLE 7.9: Performance of Model of NewViewGTRGAMMAPROT Hardware Kernel Compared With AVX Version (25 Taxa)

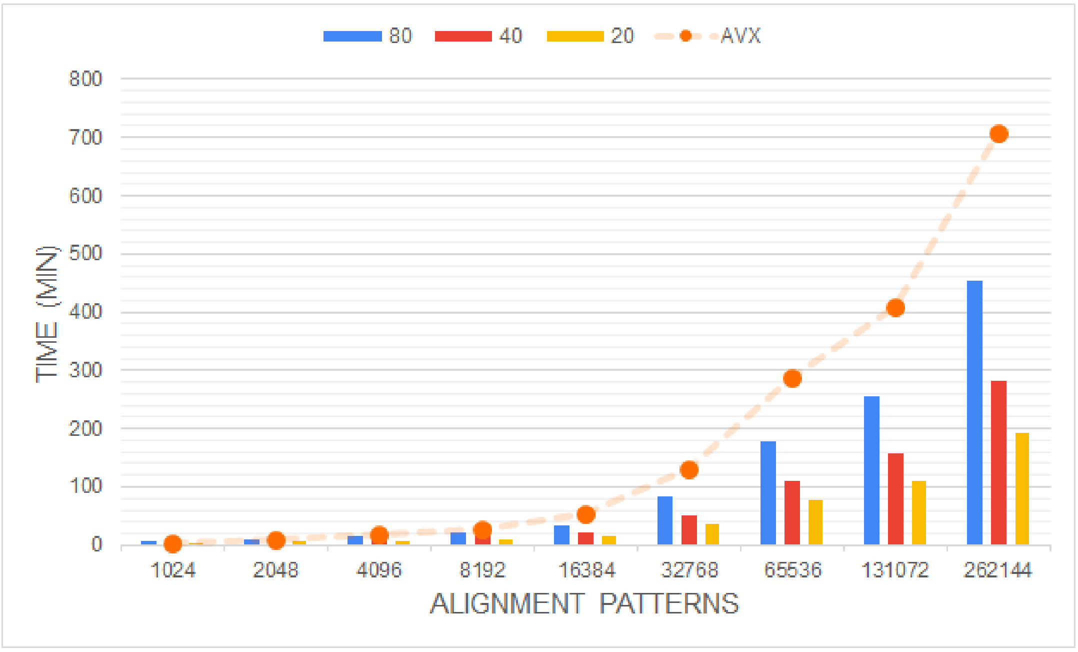


FIGURE 7.23: Model of the Kernels Performance using 50 taxa

	1024	**2048**	**4096**	**8192**	**16384**	**32768**	**65536**	**131072**	**262144**
80	x0.69	x0.95	x1.18	x1.31	x1.60	x1.60	x1.61	x1.61	x1.56
40	x0.83	x1.24	x1.67	x1.96	x2.49	x2.54	x2.58	x2.59	x2.52
20	x0.93	x1.47	x2.11	x2.63	x3.44	x3.60	x3.70	x3.74	x3.65

TABLE 7.10: Performance of Model of NewViewGTRGAMMAPROT Hardware Kernel Compared With AVX Version (50 Taxa)

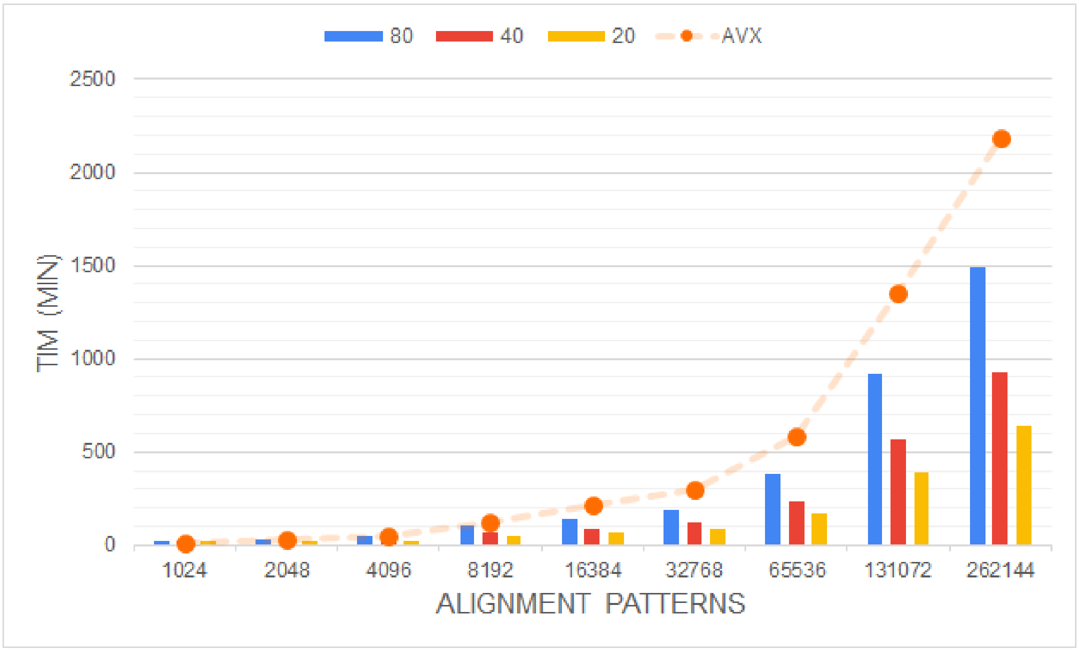
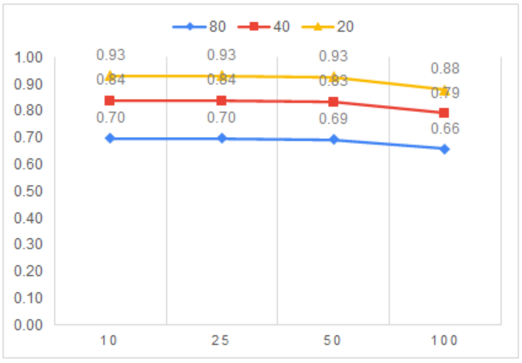


FIGURE 7.24: Model of the Kernels Performance using 100 taxa

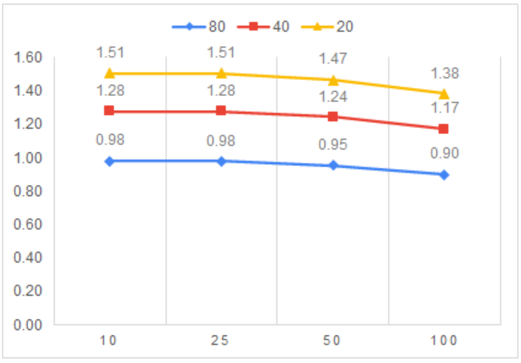
	1024	**2048**	**4096**	**8192**	**16384**	**32768**	**65536**	**131072**	**262144**
80	x0.66	x0.90	x1.10	x1.20	x1.55	x1.57	x1.53	x1.47	x1.47
40	x0.79	x1.17	x1.56	x1.80	x2.40	x2.49	x2.45	x2.38	x2.37
20	x0.88	x1.38	x1.97	x2.41	x3.32	x3.52	x3.51	x3.43	x3.43

TABLE 7.11: Performance of Model of NewViewGTRGAMMAPROT Hardware Kernel Compared With AVX Version (100 Taxa)

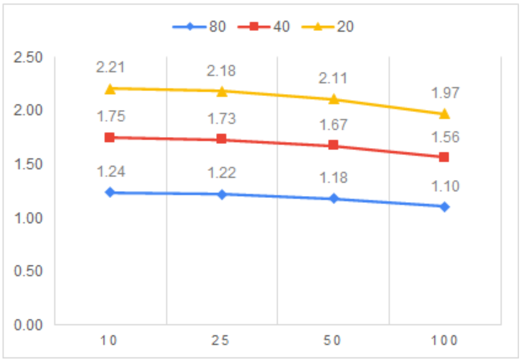
As we can see on the above figures 7.21-7.24 and the corresponding tables 7.8-7.11, optimizing and reaching a lower II, a significant acceleration could be achieved. The above comparison is between the execution times on hardware and the AVX version of the function. It is obvious that a number of taxa, lower than 100, have approximately the same performance. Achieving the II equal to 80 the acceleration factor comes approximately up to x1.5 when the alignment patterns are more than 4K and reaches x4-5 while they are enlarging. On the other hand, when the number of taxa is bigger than 100, the acceleration factor starts to decrease by some decimals. This small reduction is caused due to the time-consuming transfer of data to the device and backward to the host. Moreover, it is not alarming because the acceleration factor still remains on the same level as the previous values with a smaller number of taxa. In the meantime, the performance of the process of the data into the kernel still brings a speed up. It must be highlighted that all these results derive from our simulation model corresponds to the total execution time of NewViewGTRGAMMAPROT (Set up time + Transfer Data + Kernel process time). Finally, on the next figures 7.25 and 7.26, the final and graded performance of our model are presented, according to the achieved II, the number of taxa and number of alignment patterns.



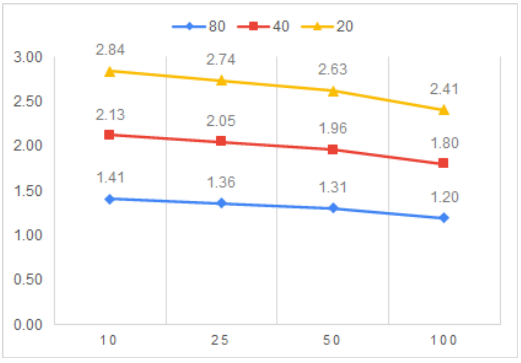
(A) 1024 Alignment Patterns



(B) 2048 Alignment Patterns

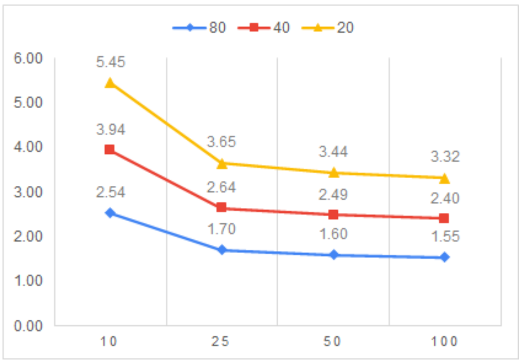


(C) 4096 Alignment Patterns

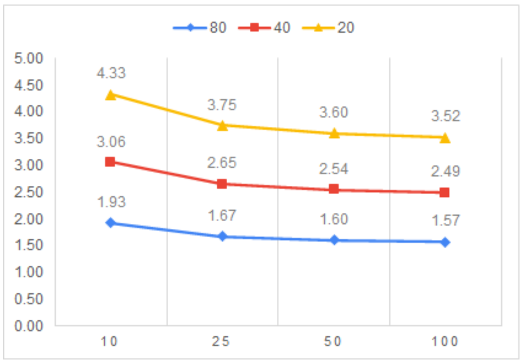


(D) 8192 Alignment Patterns

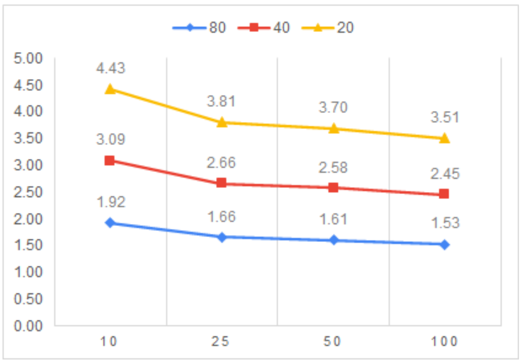
FIGURE 7.25: Final Performance Of Speedup of NewViewGTRGAMMAPROT kernel (1)



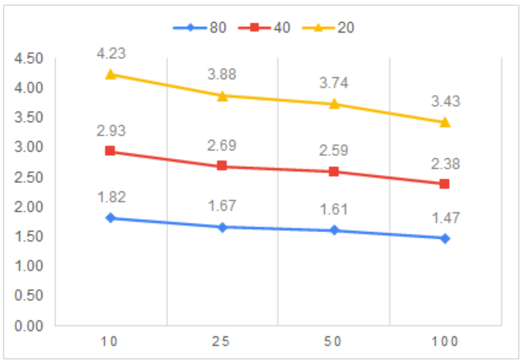
(A) 16384 Alignment Patterns



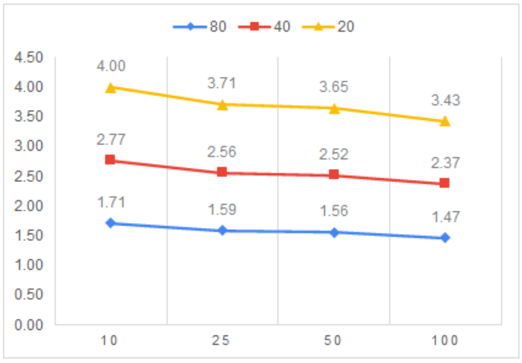
(B) 32768 Alignment Patterns



(C) 65536 Alignment Patterns



(D) 131072 Alignment Patterns



(E) 262144 Alignment Patterns

FIGURE 7.26: Final Performance Of Speedup of NewViewGTRGAMMAPROT kernel (2)

7.5 Analysis and Final Performance

Summarizing all the above results, the final performance of RAxML can be estimated using Amdahl's law. Capturing the best speedup factor from the above cases and platforms, the feasible speedup of RAxML is calculated:

- On ZCU102, the accelerator for the function SumGAMMA(3rd case) could not achieve a speedup so it cannot bring a speedup on RAxML.

- On ZCU102, the accelerator for the PLF(3^{rd} case), brought a maximum speedup factor x3.01, compared with the sequential version of the code. Although our goal is not to accelerate the sequential version, it is useful to know how this speedup affects the whole performance. With this speedup factor, PLF can be accelerated by x1.43, and consecutively RAxML can be accelerated by x1.36.
- On AWS F1 instance, the accelerator for the function SumGAMMA(3^{rd} case) achieved a maximum speedup factor x1.13, using big data sets $\geq 131K$. The whole top function can benefit from it and be accelerated by x1.06. Then RAxML is only affected by an x1.003 factor which is too little to bring a significant acceleration.
- On AWS F1 instance, the accelerator for the PLF(3^{rd} case), brought a maximum speedup factor x3.35, compared with the sequential version of the code. The whole PLF benefits from it and is accelerated by x1.46 while RAxML is accelerated by x1.38.

As we can see all these acceleration factors can not surpass the optimal ones analyzed and mentioned in chapter 5. That was expected while on the initial theoretical computations we did not count on the communication parameters. With regards to the final results that we obtained and whether we achieved our goal, the following paragraphs are going to analyze them.

When we estimated theoretically the optimal performance of RAxML with an accelerated function of SumGAMMAPROT, the result was that the overall speedup of RAxML could be maximum x1.02. Our accelerator used on ZCU102 could not bring an optimization while itself on AWS F1 gave an x1.003. These values are too low which shows us that functions that occupy a small percentage, like 5% of total time, can not bring an intended outcome on these two FPGAs. Moreover, we can assume that functions with a small number of operations, are better to be executed by CPUs.

As regards the PLF, the theoretical maximum estimation was x1.63 acceleration of RAxML. Our accelerator mapped on ZCU102 brought an acceleration x1.36 on RAxML while itself mapped on AWS F1 brought an acceleration x1.38. These factors regard the comparison with the sequential version of RAxML. On the contrary, our goal was to surpass the performance of the AVX version of PLF. These two platforms seem to be inadequate enough to support the optimal implementation of our accelerators and that is due to the number of their resources. Moreover, ZCU102 has an ARM processor which is very slow enough to support such systems. In the meanwhile, on the AWS F1 instance, we did not meet any problems with its processor. The common problem that we met on both platforms was the transfer of data. While the data sets and their number of alignment patterns increase, it is necessary to find other solutions, such as double or triple buffering, to transfer and export the data from/to the kernel.

Assuming that we could map, place & route our accelerator on a bigger platform, then we could achieve a better II on our accelerator for PLF, a fact that could bring an improved performance compared with our threshold which is the AVX version. Following the theoretical model in the previous section, these acceleration factors could be achieved, installing the accelerators on a bigger platform. Moreover, we can hit the following acceleration factors when the number of alignment patterns in datasets is greater than 4K. So, in the case that we achieve an II=80, the maximum speedup would be x2.54 which brings an x1.31 speedup on RAxML. With II=40 the accelerator brings a maximum speedup x3.94 and accordingly RAxML is accelerated by x1.41. Finally, with II=20 the accelerator brings a maximum speedup of x5.45, and correspondingly RAxML is being accelerated by x1.47.

Chapter 8

Conclusions and Future Work

In this chapter, this thesis's workload is being summed up and evaluated. Also, directions for future work, possible extensions, and optimizations are being given.

8.1 Conclusions

Over the last years, scientists who work in the field of bioinformatics, give their best to solve the significant problems that arise from the excessive increase of data. On this effort, there is a contribution by technology innovations and services, such as GPUs, FPGAs, and clouds, which help scientists to find new guidelines on their way to the solutions.

In general, our thesis's goal was to provide a solution for the above problem, trying to reduce the total required time for the processing data of an algorithm. Specifically, we managed to accelerate the total execution time of some time-consuming functions of an algorithm that use as input data amino acid sequences, trying to process them and develop an evolution-relation tree between them.

In this thesis we achieved to :

- Examine and analyze the optimal performances of our systems, taking advantage of the specifications of the given platforms such as the clock frequencies, memory access patterns, and bandwidths.
- Design hardware kernels with as far as possible the minimum provided resources of the targeted platforms.
- Accelerate the initial functions (both their sequential and AVX versions), using these kernels, by a significant factor.

- Propose one of the least implemented accelerators of an algorithm that targets the usage and processing of proteins or amino acids data. That fact is high of importance because most of the related previous works focused on DNA data. Although the processing of amino acid data has more constraints we achieved to deal with it.

8.2 Future Work

This thesis' designs offered a reliable acceleration on the target functions of RAxML and the platform on AWS F1 instance is easily accessible by everyone. However, all this work opens a frame with proposals for the accelerators' optimization. Some of them are presented below:

- A Better and sufficient technique for the transfer of the data from Host to Device and vice versa on all accelerators.
- Execute the current kernel of NewViewGTRGAMMAPROT on a larger platform to achieve the optimal II of 10 and export the conclusion for the accelerator.
- Although there were sufficient data sets for the executions of RAxML and the accelerators, it could be a good idea to gather real data sets from scientists to run them and show the performance of the accelerators.
- It would be a significant optimization if a better way of recalling the kernel and transfer data without time losses during the total run of RAxML was designed.
- Integration of the other two cases (tip-tip & tip-inner) of these functions into the kernels and with a larger targeted platform. Alternatively, they might be used as standalone kernels.
- Dealing with difficulties due to the worldwide pandemic this year, the access on the first platform was minimum. On other conditions, more data sets could run to present a better and more gradient performance of our accelerators. This could be an apropos remark.

References

[1] Nikolaos Alachiotis et al. “Accelerating Phylogenetics Using FPGAs in the Cloud”. In: *IEEE Micro* 41.4 (2021), pp. 24–30. DOI: 10.1109/MM.2021.3075848.

[2] A. Stamatakis. “RAxML-VI-HPC: maximum likelihood-based phylogenetic analyses with thousands of taxa and mixed models”. In: *Bioinformatics* 22.21 (Aug. 2006), pp. 2688–2690. ISSN: 1367-4803. DOI: 10.1093/bioinformatics/btl446. eprint: https://academic.oup.com/bioinformatics/article-pdf/22/21/2688/16851699/btl446.pdf. URL: https://doi.org/10.1093/bioinformatics/btl446.

[4] F. Izquierdo-Carrasco et al. “A Generic Vectorization Scheme and a GPU Kernel for the Phylogenetic Likelihood Library”. In: *2013 IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW)*. Los Alamitos, CA, USA: IEEE Computer Society, 2013, pp. 530–538. DOI: 10.1109/IPDPSW.2013.103. URL: https://doi.ieeecomputersociety.org/10.1109/IPDPSW.2013.103.

[5] A. Stamatakis et al. “Exploring new Search Algorithms and Hardware for Phylogenetics: RAxML meets the IBM Cell”. In: *The Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology* 48 (Sept. 2007), pp. 271–286. DOI: 10.1007/s11265-007-0067-4.

[7] A. Dollas E. Sotiriades C. Kozanitis. “FPGA based Architecture of DNA Sequence Comparison and Database Search”. In: *Proceedings 20th International Parallel and Distributed Processing Symposium, IPDPS 2006* at the 13th Reconfigurable Architectures Workshop Rhodes, Greece (Apr. 2006), Page: 193.

[8] A. Dollas E. Sotiriades C. Kozanitis. “Some Initial Results on Hardware BLAST Acceleration with a Reconfigurable Architecture”. In: *Proceedings 20th International Parallel and Distributed Processing Symposium, IPDPS 2006* at the 5th IEEE International Workshop on High Performance Computational Biology (HiCOMB2006) (Apr. 2006), Page: 251.

[9] A. Dollas E. Sotiriades. “A General Reconfigurable Architecture for the BLAST algorithm”. In: *The Journal of VLSI Signal Processing Systems for*

Signal, Image, and Video Technology, Special Issue on Computing Architectures and Acceleration for Bioinformatics Algorithms Volume 48, Issue 3 (Sept. 2007), Pages: 189–208.

[10] D. Maskel T. Oliver B. Schmidt. "Reconfigurable Architectures for Bio-sequence Database Scanning on FPGAs". In: *IEEE Transactions on Circuits and Systems II* Volume 52, No 12 (2005), Pages: 851–855.

[11] K. Underwood K. Muriki and R. Sass. "RC-BLAST: Towards an open source hardware implementation". In: *Proceedings of the International Workshop on High Performance Computational Biology* (2005).

[12] Y. Gu B. Sukhwani T. VanCourt M. Herbordt J. Model. "Single Pass, BLAST-Like, Approximate String Matching on FPGAs". In: *14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines(FCCM'06)* (2006), Pages: 217–226.

[13] B. Sukhwani Y. Gu M. Herbordt J. Model and T. VanCourt. "Single pass streaming BLAST on FPGAs". In: *Parallel Computing* Volume 33, issue 10-11 (2007), Pages: 741–756.

[14] V. Promponas et al. "CAST: an iterative algorithm for the complexity analysis of sequence tracts. Complexity analysis of sequence tracts". In: *Bioinformatics (Oxford, England)* 16.10 (2000), 915—922. ISSN: 1367-4803. DOI: `10.1093/bioinformatics/16.10.915`. URL: `https://doi.org/10.1093/bioinformatics/16.10.915`.

[15] A. Papadopoulos, V. Promponas, and T. Theocharides. "Towards systolic hardware acceleration for local complexity analysis of massive genomic data". In: May 2012. DOI: `10.1145/2206781.2206864`.

[16] K. Katoh et al. "Katoh K, Kuma K, Toh H, Miyata TMAFFT version 5: Improvement in accuracy of multiple sequence alignment. Nucleic Acids Res 33:511-518". In: *Nucleic acids research* 33 (Feb. 2005), pp. 511–8. DOI: `10.1093/nar/gki198`.

[17] C. Notredame, D. G. Higgins, and J. Heringa. "T-coffee: a novel method for fast and accurate multiple sequence alignment11Edited by J. Thornton". In: *Journal of Molecular Biology* 302.1 (2000), pp. 205 –217. ISSN: 0022-2836. DOI: `https://doi.org/10.1006/jmbi.2000.4042`. URL: `http://www.sciencedirect.com/science/article/pii/S0022283600940427`.

[18] G. Chrysos E. Sotiriades I. Papaefstathiou A. Dollas M. Lakka A. Desarti. "Reconfigurable Computing IP Cores for Multiple Sequence Alignment". In: *Proceedings of BIOINFORMATICS* (2011), pp. 216–221.

[19] N. R Markham and M. Zuker. "UNAFold: software for nucleic acid folding and hybridization". In: *Methods in molecular biology (Clifton, N.J.)*

453 (2008), 3—31. ISSN: 1064-3745. DOI: 10.1007/978-1-60327-429-6_1. URL: https://doi.org/10.1007/978-1-60327-429-6_1.

[20] D. Frishman and P. Argos. “Incorporation of long-distance interactions into a secondary structure prediction algorithm”. In: *Protein Engineering* vol 9 (1996), pp.133–142.

[21] G. Chrysos E. Sotirades A. Dollas M. Smerdis P. Dagritzikos. “Reconfigurable Systems for the Zuker and the Predator Algorithms for Secondary Structure Prediction of Genetic Data”. In: *In the proccedings of Field Programmable Logic (FPL)* (2010).

[22] Arpith C. Jacob, J. Buhler, and R. Chamberlain. “Rapid RNA Folding: Analysis and Acceleration of the Zuker Recurrence”. In: *2010 18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines* (2010), pp. 87–94.

[23] S. L. Salzberg et al. “Microbial gene identification using interpolated Markov models”. In: *Nucleic Acids Research* 26.2 (Jan. 1998), pp. 544–548. ISSN: 0305-1048. DOI: 10.1093/nar/26.2.544. eprint: https://academic.oup.com/nar/article-pdf/26/2/544/3995426/26-2-544.pdf. URL: https://doi.org/10.1093/nar/26.2.544.

[24] I. Papaefstathiou G. Chrysos E. Sotiriades and A. Dollas. “A FPGA based coprocessor for gene finding using Interpolated Markov Model (IMM)”. In: *Proceedings of the 19th International Conference on Field Programmable Logic and Applications (FPL '09)* (Aug. 2009), pp. 683–686.

[29] N. Alachiotis et al. “A reconfigurable architecture for the Phylogenetic Likelihood Function”. In: *2009 International Conference on Field Programmable Logic and Applications*. 2009, pp. 674–678. DOI: 10.1109/FPL.2009.5272341.

[30] S. A. Berger, N. Alachiotis, and A. Stamatakis. “An Optimized Reconfigurable System for Computing the Phylogenetic Likelihood Function on DNA Data”. In: *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops PhD Forum*. 2012, pp. 352–359. DOI: 10.1109/IPDPSW.2012.43.

[31] N. Alachiotis et al. “Exploring FPGAs for accelerating the phylogenetic likelihood function”. In: *2009 IEEE International Symposium on Parallel Distributed Processing*. 2009, pp. 1–8. DOI: 10.1109/IPDPS.2009.5160929.

[32] S. Zierke and J.D. Bakos. “FPGA Acceleration of the phylogenetic likelihood function for Bayesian MCMC inference methods”. In: *BMC bioinformatics* 11 (Apr. 2010), p. 184. DOI: 10.1186/1471-2105-11-184.

[33] A. M. Kozlov, C. Goll, and A. Stamatakis. “Efficient Computation of the Phylogenetic Likelihood Function on the Intel MIC Architecture”. In: *2014 IEEE International Parallel Distributed Processing Symposium Workshops*. 2014, pp. 518–527. DOI: 10.1109/IPDPSW.2014.198.

[34] A. Stamatakis et al. “RAxML-Light”. In: *Bioinformatics* 28.15 (Aug. 2012), 2064–2066. ISSN: 1367-4803. DOI: 10.1093/bioinformatics/bts309. URL: https://doi.org/10.1093/bioinformatics/bts309.

[35] A. Stamatakis A. M. Kozlov A. J. Aberer. “ExaML Version 3: A Tool for Phylogenomic Analyses on Supercomputers”. In: (). DOI: 10.1093/bioinformatics/btv184.

[36] W. R. Gilks. “M arkov Chain M onte C arlo”. In: *Encycl. Biostat.* vol. 4 (2005).

[37] S. Whelan and N. Goldman. “A General Empirical Model of Protein Evolution Derived from Multiple Protein Families Using a Maximum-Likelihood Approach”. In: *Molecular Biology and Evolution* 18.5 (May 2001), pp. 691–699. ISSN: 0737-4038. DOI: 10.1093/oxfordjournals.molbev.a003851. eprint: https://academic.oup.com/mbe/article-pdf/18/5/691/23447821/i0737-4038-018-05-0691.pdf. URL: https://doi.org/10.1093/oxfordjournals.molbev.a003851.

[38] J. Felsenstein. “EVOLUTIONARY TREES FROM GENE FREQUENCIES AND QUANTITATIVE CHARACTERS: FINDING MAXIMUM LIKELIHOOD ESTIMATES”. In: *Evolution* 35.6 (1981), pp. 1229–1242. DOI: https://doi.org/10.1111/j.1558-5646.1981.tb04991.x. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/j.1558-5646.1981.tb04991.x. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1558-5646.1981.tb04991.x.

[39] M. Ott. “Inference of Large Phylogenetic Trees on Parallel Architectures”. In: (2010).

[40] P. J Waddell and M.A Steel. “General Time-Reversible Distances with Unequal Rates across Sites: Mixing Γ and Inverse Gaussian Distributions with Invariant Sites”. In: *Molecular Phylogenetics and Evolution* 8.3 (1997), pp. 398 –414. ISSN: 1055-7903. DOI: https://doi.org/10.1006/mpev.1997.0452. URL: http://www.sciencedirect.com/science/article/pii/S1055790397904528.

[41] Z. Yang. “Maximum Likelihood Phylogenetic Estimation from DNA Sequences with Variable Rates over Sites: Approximate Methods”. In: *Journal of molecular evolution* 39 (Oct. 1994), pp. 306–14. DOI: 10.1007/BF00160154.

[42] A. Stamatakis and Michael Ott. “Exploiting Fine-Grained Parallelism in the Phylogenetic Likelihood Function with MPI, Pthreads, and OpenMP: A Performance Study”. In: *Pattern Recognition in Bioinformatics*. Ed. by Madhu Chetty, Alioune Ngom, and Shandar Ahmad. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 424–435. ISBN: 978-3-540-88436-1.

[43] G. Charitopoulos et al. “A decoupled access-execute architecture for reconfigurable accelerators”. In: May 2018, pp. 244–247. DOI: 10.1145/3203217.3203267.

External Links

[3] *PHYLIP type of data*. URL: URL:https://evolution.genetics.washington.edu/phylip.html.

[6] *BLAST Algorithm*. URL: URL:https://blast.ncbi.nlm.nih.gov/Blast.cgi?CMD\%20=\%20Web&PAGE_TYPE\%20=\%20BlastDocs\&DOC_TYPE\%20=\%20DeveloperInfo.

[25] *GARLI Algorithm*. URL: URL:http://www.bio.utexas.edu/faculty/antisense/garli/Garli.html.

[26] *Mr Bayes Algorithm*. URL: URL:http://nbisweden.github.io/MrBayes/.

[27] *PAML Algorithm*. URL: URL:http://web.mit.edu/6.891/www/lab/paml.html.

[28] *PAUP Algorithm*. URL: URL:https://paup.phylosolutions.com/.

[44] *Vivado Design Suite - HLx Editions*. URL: URL:https://www.xilinx.com/products/design-tools/vivado.html.

[45] *SDx Development Environments and Embedded Computing*. URL: URL:https://www.xilinx.com/products/design-tools/software-zone.html.

[46] *Vivado Design Suite User Guide - Using High-Level Synthesis*. URL: URL:https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_3/ug902-vivado-high-level-synthesis.pdf.

[47] *SDSoC Environment User Guide*. URL: URL:https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_3/ug1027-sdsoc-user-guide.pdf.

[48] *SDAccel Environment User Guide*. URL: URL:https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_3/ug1023-sdaccel-user-guide.pdf.

[49] *Vivado Design Suite User Guide - Using the Vivado IDE*. URL: URL:https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_3/ug893-vivado-ide.pdf.

[50] *UltraFast Design Methodology Timing Closure Quick Reference Guide*. URL: URL:https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_1/ug1292-ultrafast-timing-closure-quick-reference.pdf.

[51] *ZCU102 Evaluation Board User Guide*. URL: URL:https://www.xilinx.com/support/documentation/boards_and_kits/zcu102/ug1182-zcu102-eval-bd.pdf.

[52] *Amazon EC2 F1 Instances*. URL: URL:https://aws.amazon.com/ec2/instance-types/f1/.

[53] *ms - a program for generating samples under neutral models by Richard R. Hudson*. URL: URL:http://home.uchicago.edu/~rhudson1/.