UNIVERSITY OF THESSALY

SCHOOL OF ENGINEERING

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

FPGA Acceleration of Bioinformatics Algorithms

Diploma Thesis

Antonia Sakellariou

Supervisor: Christos Antonopoulos

Volos 2021

UNIVERSITY OF THESSALY

SCHOOL OF ENGINEERING

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

FPGA Acceleration of Bioinformatics Algorithms

Diploma Thesis

Antonia Sakellariou

Supervisor: Christos Antonopoulos

Volos 2021

ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ

ΠΟΛΥΤΕΧΝΙΚΗ ΣΧΟΛΗ

ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Επιτάχυνση Αλγορίθμων Βιοπληροφορικής σε FPGA

Διπλωματική Εργασία

Αντωνία Σακελλαρίου

Επιβλέπων: Χρήστος Αντωνόπουλος

Βόλος 2021

Approved by the Examination Committee:

Supervisor **Christos Antonopoulos**
Associate professor, Department of Electrical and Computer Engineering, University of Thessaly

Member **Spyros Lalis**
Professor, Department of Electrical and Computer Engineering, University of Thessaly

Member **Dimitrios Katsaros**
Associate professor, Department of Electrical and Computer Engineering, University of Thessaly

Date of approval: 16-7-2021

Acknowledgements

First and foremost, I would like to thank my thesis supervisor Prof. Nikolaos Bellas, for his guidance, and support throughout the duration of my studies. Most importantly, I would like to thank him for always being available and, for his great advice any time I needed it. I would also like to thank him for thank him for our great collaboration all those years.

I would also like to thank PhD Candidate Alexandros Patras, for always being available and for answering my endless questions. His help was detrimental for the outcome of this thesis.

I would also like to thank my parents and my sisters for always being my biggest supporters. Thank you for always being there for me and for your encouragement. Lastly, I would like to thank my friends, that played a huge part on this journey. Thank you for all your support for those five years.

DISCLAIMER ON ACADEMIC ETHICS AND INTELLECTUAL PROPERTY RIGHTS

«Being fully aware of the implications of copyright laws, I expressly state that this diploma thesis, as well as the electronic files and source codes developed or modified in the course of this thesis, are solely the product of my personal work and do not infringe any rights of intellectual property, personality and personal data of third parties, do not contain work / contributions of third parties for which the permission of the authors / beneficiaries is required and are not a product of partial or complete plagiarism, while the sources used are limited to the bibliographic references only and meet the rules of scientific citing. The points where I have used ideas, text, files and / or sources of other authors are clearly mentioned in the text with the appropriate citation and the relevant complete reference is included in the bibliographic references section. I fully, individually and personally undertake all legal and administrative consequences that may arise in the event that it is proven, in the course of time, that this thesis or part of it does not belong to me because it is a product of plagiarism».

The declarant

Antonia Sakellariou
16-7-2021

Abstract

Bioinformatics is one of the most growing fields in the technology industry. Bioinformatics combines biology and computer science in order to analyze biological data, mainly genomes. Due to high computational complexity, algorithms used in bioinformatics such as Gibbs Sampler for motif finding, are extremely slow on commercial devices, and there is a need of accelerating them using state-of-the-art technology. Gibbs Sampler is one of the most known algorithms for solving the motif finding problem, which searches for motifs, that are a set of nucleotides where a transcription factor binds to a DNA sequence.

This thesis presents an implementation of the Gibbs Sampler algorithm first in software running on a multicore CPU, and, then, in hardware on a high performance FPGA device. Our work introduces various optimizations, and we show that Gibbs Sampler algorithm can be highly accelerated. The execution time for Gibbs Sampler on the ARM processor was 3600 ms for an input of 10 DNA sequences that consist of 1974 nucleotides each, and returns the best motifs found in these sequences of size 15. Our FPGA implementation, using a Xilinx Ultrascale+ ZCU102 MPSoC, enables high performance total time execution at 53 ms, which is 67x times faster than the ARM processor implementation.

The source code of the thesis: github.com/ansakellariou/GibbsSamplerFPGA

Περίληψη

Η βιοπληροφορική είναι ένας από τους πιο αναπτυσσόμενους τομείς στον κλάδο της τεχνολογίας. Η βιοπληροφορική συνδυάζει τη βιολογία και την επιστήμη των υπολογιστών προκειμένου να αναλύσει βιολογικά δεδομένα, κυρίως γονιδιώματα. Λόγω της υψηλής υπολογιστικής πολυπλοκότητας, οι αλγόριθμοι που χρησιμοποιούνται στη βιοπληροφορική, όπως το Gibbs Sampler για εύρεση μοτίβων, είναι εξαιρετικά αργοί σε εμπορικές συσκευές και υπάρχει ανάγκη επιτάχυνσης τους χρησιμοποιώντας τεχνολογία αιχμής. Το Gibbs Sampler είναι ένας από τους πιο γνωστούς αλγόριθμους για την επίλυση του προβλήματος εύρεσης μοτίβου, ο οποίος αναζητά μοτίβα, που είναι ένα σύνολο νουκλεοτιδίων όπου ένας παράγοντας μεταγραφής συνδέεται με μια αλληλουχία DNA.

Αυτή η διατριβή παρουσιάζει μια εφαρμογή του αλγορίθμου Gibbs Sampler πρώτα σε λογισμικό που εκτελείται σε CPU πολλαπλών πυρήνων και, στη συνέχεια, σε υλικό σε μια συσκευή FPGA υψηλής απόδοσης. Η δουλειά μας εισάγει διάφορες βελτιστοποιήσεις και δείχνουμε ότι ο αλγόριθμος Gibbs Sampler μπορεί να επιταχυνθεί σημαντικά. Ο χρόνος εκτέλεσης του Gibbs Sampler στον επεξεργαστή ARM ήταν 3600 ms για είσοδο 10 ακολουθιών DNA που αποτελούνται από 1974 νουκλεοτίδια το καθένα και επιστρέφει τα καλύτερα μοτίβα που βρέθηκαν σε αυτές τις ακολουθίες μεγέθους 15. Η υλοποίηση και οι βελτιστοποιήσεις που εφαρμόσαμε στην FPGA, χρησιμοποιώντας την Xilinx Ultrascale + ZCU102 MPSoC, επιτρέπει την εκτέλεση υψηλής απόδοσης συνολικού χρόνου στα 53 ms, η οποία είναι 67 φορές πιο γρήγορη από την υλοποίηση στον επεξεργαστή ARM.

Ο πηγαίος κώδικας της διατριβής: github.com/ansakellariou/GibbsSamplerFPGA

Table of contents

List of figures

List of tables

Abbreviations

II	Initiation interval
2D	two dimension
1D	one dimension
K	motif length
T	number of DNA sequences
L	Dna sequence length
HLS	High-Level Synthesis
FPGA	Field Programmable Gate Array
BRAM	Block Random Access Memory
k-mer	motif of length k

Chapter 1

Introduction

The field of bioinformatics combines computer science and biology, and its purpose is the collection and interpretation of biological data by using computational methods. The main problem of this field is the vast amount of data that one has to process in order to analyze anything that has to do with a genome. If someone tries to analyze the data conventionally, for instance accessing data one by one and processing it with a brute force algorithm, it could take forever to take some results back. This is the reason why alternative methods are looked for by computer and bioinformatics scientists.

1.1 Subject of the thesis

Most of the algorithms that try to analyze data that come from a genome have to be accelerated in some way, in order to return the results on a realistic time frame. This thesis is focusing on the acceleration of Gibbs sampler algorithm for motif finding in DNA sequences. After we implemented the algorithm in C language and got its timing analysis, we tried to optimize its performance by using OpenMP. The results were slightly better, but there was still room for improvement. Then we made the necessary changes for the algorithm to be able to run on hardware, which were very important for its functionality, as some things that are supported by software are not supported by hardware. After that, we tried to optimize further the time that the algorithm needed to run on the FPGA by making precise and approximate optimizations. The last part of the thesis is the analysis of all those results, that came from implementing and optimizing Gibbs sampling algorithm both on software and hardware.

1.1.1 Contribution

This thesis contribution is summarized as follows:

1. Gibbs sampler was implemented in C and its performance was analyzed using Gprof tool and clock() function. We measured the time that the algorithm needed to run, the time each function needed to run, and the total calls of each function.

2. Gibbs sampler was optimized to run in a multithreaded way using OpenMP.

3. Gibbs sampler algorithm was implemented on an FPGA device, after being transformed accordingly in order to be supported by hardware.

4. Gibbs sampler was optimized on the FPGA precisely using HLS pragmas and approximately by reducing the number of its iterations.

5. The results of all of those implementations and optimizations were analyzed and compared.

1.2 Organization of the thesis

Below is a brief presentation of the contents of each chapter:

Chapter 2 presents background information related to the topics that will be covered by this thesis. After a brief introduction to the field of bioinformatics, we explain the Motif Finding algorithms, focusing on Gibbs Sampler algorithm. Then, we present the tools that were use for timing analysis of C code, also we provide the reader with basic information on OpenMP, Vitis Unified Software platform, OpenCL, Vivado HLS tool and lastly FPGA technology.

Chapter 3 presents the steps that we took for implementing the pseudocode of Gibbs sampling algorithm in C and the optimizations that we did in OpenMP. Also, the alterations that the algorithm needed to have in order to be implemented on the FPGA device. Lastly, the precise and approximate optimizations of the algorithm in order to accelerate it further.

Chapter 4 presents the results that we got after implementing Gibbs sampler algorithm on software and hardware and optimized it to work faster. Firstly, we analyze the results of running Gibbs sampler algorithm on x86 processor without optimizations, on ARM processor, on x86 processor with OpenMP optimizations, and we compare those results. Then, we show

the results of the implementation on the FPGA and compare it to all the previous results. Lastly, we present and analyze the results of the FPGA optimizations and of course compare them as well with all the previous results. Finally, Chapter 4 combines all the results of the thesis.

Chapter 5 presents a summary of this thesis and the conclusions that we can draw from the final results. It also presents some ideas for future development of the thesis.

Chapter 2

Background

This chapter describes the fundamentals needed to read and understand the content of this thesis. First, we describe what bioinformatics are, with a focus on motif finding algorithms, what is Gibbs Sampler and how it works. We also go through the OpenMP framework and FPGA technology, which we used to accelerate the Gibbs Sampler algorithm C version.

2.1 Bioinformatics

Bioinformatics [7] is a growing area of biology that plays an increasingly important role in new discoveries and developments in the field. It is actually a combining field that connects biology, computer science, physics, chemistry, mathematics and statistics, to analyze and process biological data, which are most of the time large and complex data sets. The most important goal of bioinformatics is developing efficient algorithm for solving trivial problems of biology and genetics. The most significant data that can be retrieved from a cell are DNA, RNA, protein sequences and micro array images, from which the first three are plain text data, which means that they can easily be processed by a computer. [8]

2.1.1 Motif finding Algorithms

Most of the information on how a cell works can be found on DNA. As a matter of fact, DNA contains the information required for gene expression, which produces proteins. Some proteins, though, are more important than others, as they are able to start and stop genes, these proteins are called transcription factors or master regulatory proteins. The way that they can control when a gene is expressed is by binding to a specific place on DNA, called a regulatory

motif or transcription binding site, and is located in the beginning of the gene. The problem of locating where these regulatory motifs are located is the **motif finding problem**. [1]

```
 1 "atgaccgggatactgatAAAAAAAAGGGGGGGggcgtacacattagataaacgtatgaagtacgttagactcggcgccgccg"
 2 "acccctatttttgagcagatttagtgacctggaaaaaaaatttgagtacaaaactttccgaataAAAAAAAAGGGGGGGa"
 3 "tgagtatccctgggatgacttAAAAAAAAGGGGGGGtgctctcccgatttttgaatatgtaggatcattcgccagggtccga"
 4 "gctgagaattggatgAAAAAAAAGGGGGGGtccacgcaatcgcgaaccaacgcggacccaaaggcaagaccgataaaggaga"
 5 "tccctttgcggtaatgtgccgggaggctggttacgtagggaagccctaacggacttaatAAAAAAAAGGGGGGGcttatag"
 6 "gtcaatcatgttcttgtgaatggatttAAAAAAAAGGGGGGGgaccgcttggcgcacccaaattcagtgtgggcgagcgcaa"
 7 "cggttttggcccttgttagaggcccccgtAAAAAAAAGGGGGGGcaattatgagagagctaatctatcgcgtgcgtgttcat"
 8 "aacttgagttAAAAAAAAGGGGGGGctggggcacatacaagaggagtcttccttatcagttaatgctgtatgacactatgta"
 9 "ttggcccattggctaaaagcccaacttgacaaatggaagatagaatccttgcatAAAAAAAAGGGGGGGaccgaaagggaag"
10 "ctggtgagcaacgacagattcttacgtgcattagctcgcttccggggatctaatagcacgaagcttAAAAAAAAGGGGGGGa"
```

Figure 2.1: An illustration of 10 DNA sequences and the regulatory motif AAAAAAAAGGGGGGG. [1]

Unfortunately, there is a possibility for this motif to be mutated, meaning that there could be slight differences between the motifs discovered in different DNA sequences and also from the original one.

```
 1 "atgaccgggatactgatAgAAgAAAGGttGGGggcgtacacattagataaacgtatgaagtacgttagactcggcgccgccg"
 2 "acccctatttttgagcagatttagtgacctggaaaaaaaatttgagtacaaaactttccgaatacAAtAAAAcGGcGGGa"
 3 "tgagtatccctgggatgacttAAAAtAAtGGaGtGGtgctctcccgatttttgaatatgtaggatcattcgccagggtccga"
 4 "gctgagaattggatgcAAAAAAAGGGattGtccacgcaatcgcgaaccaacgcggacccaaaggcaagaccgataaaggaga"
 5 "tccctttgcggtaatgtgccgggaggctggttacgtagggaagccctaacggacttaatAtAAtAAAGGaaGGGcttatag"
 6 "gtcaatcatgttcttgtgaatggatttAAcAAtAAGGGctGGgaccgcttggcgcacccaaattcagtgtgggcgagcgcaa"
 7 "cggttttggcccttgttagaggcccccgtAtAAAcAAGGaGGGccaattatgagagagctaatctatcgcgtgcgtgttcat"
 8 "aacttgagttAAAAAAtAGGGaGccctggggcacatacaagaggagtcttccttatcagttaatgctgtatgacactatgta"
 9 "ttggcccattggctaaaagcccaacttgacaaatggaagatagaatccttgcatActAAAAAGGaGcGGaccgaaagggaag"
10 "ctggtgagcaacgacagattcttacgtgcattagctcgcttccggggatctaatagcacgaagcttActAAAAAGGaGcGGa"
```

Figure 2.2: An illustration of 10 DNA sequences and the regulatory motif AAAAAAAAGGGGGGG mutated at four random positions. [1]

This problem arises the need for some algorithms for **Motif finding**. Such an algorithm is the Brute force algorithm, which explores all possible motifs and checks if it solves the problem, but it is extremely slow.

Terms that need to be explained prior to introducing more effective algorithms:

- **Score(Motifs)**: it is the number of unpopular(lower case) letters

- **Count(Motifs)**: it is a 4*k matrix, where k is the size of the motifs, that counts how many times each nucleotide appears in a column
- **Profile(Motifs)**: it is a 4*k matrix, where every cell contains the frequency of the same cell of the Count matrix
- **Consensus(Motifs)**: it is an ideal motif for the given regions of DNA

[1]

Motifs		T	C	G	G	G	G	g	T	T	T	t	t
		c	C	G	G	t	G	A	c	T	T	a	C
		a	C	G	G	G	G	A	T	T	T	t	C
		T	t	G	G	G	G	A	c	T	T	t	t
		a	a	G	G	G	G	A	c	T	T	C	C
		T	t	G	G	G	G	A	c	T	T	C	C
		T	C	G	G	G	G	A	T	T	c	a	t
		T	C	G	G	G	G	A	T	T	c	C	t
		T	a	G	G	G	G	A	a	c	T	a	C
		T	C	G	G	G	t	A	T	a	a	C	C
Score(Motifs)		3	+	4	+	0	+	0	+	1	+	1	+	1	+	5	+	2	+	3	+	6	+	4	=	30
Count(Motifs)	A:	2	2	0	0	0	0	9	1	1	1	3	0
	C:	1	6	0	0	0	0	0	4	1	2	4	6
	G:	0	0	10	10	9	9	1	0	0	0	0	0
	T:	7	2	0	0	1	1	0	5	8	7	3	4
Profile(Motifs)	A:	.2	.2	0	0	0	0	.9	.1	.1	.1	.3	0
	C:	.1	.6	0	0	0	0	0	.4	.1	.2	.4	.6
	G:	0	0	1	1	.9	.9	.1	0	0	0	0	0
	T:	.7	.2	0	0	.1	.1	0	.5	.8	.7	.3	.4
Consensus(Motifs)		T	C	G	G	G	G	A	T	T	T	C	C

Figure 2.3: An example of Score, Count Profile and Consensus for the motif TCGGGGATTTCC. [1]

2.1.2 Gibbs Sampler

Now that we have introduced the above terms, we can describe a randomized algorithm that by rolling a virtual dice is searching for motifs. Even though randomized algorithms do not offer the control over the results that greedy algorithms provide, they take significantly less time to find a solution, which is approximate. This gives us the opportunity to run the

algorithm multiple times to find the best approximation possible, which is most of the time close to the solution. **Gibbs sampler algorithm**, which is the main algorithm for this thesis, is a Monte Carlo algorithm. [9]

Its main idea is that by starting with a collection of stings DNA and a matrix Profile, which is the profile of some randomly selected Motifs from DNA, it uses this information to generate a new collection of motifs, which is again processed in the same way and so on, for as many iterations as the user defines. Each time only one motif, which is again randomly selected, is changed, only if it has a worse score than the one that is being examined in the same DNA line. The reason because only one motif is chosen to be changed is so that we can avoid missing a motif if it is in a selection that has a worse score than the others, but it can be in the solution by itself. [1]

```
1   Gibbs_Sampler(Dna, k, t, N)
2       randomly select k-mers Motifs = (Motif1, …, Motift) in each
            string from Dna
3       BestMotifs ← Motifs
4       for j ← 1 to N
5           i ← Random(t)
6           Profile ← profile matrix constructed from all strings in
                Motifs except for Motifi
7           Motifi ← Profile-randomly generated k-mer in the i-th
                sequence
8           if Score(Motifs) < Score(BestMotifs)
9               BestMotifs ← Motifs
10      return BestMotifs
```

Figure 2.4: Pseudocode for Gibbs Sampler [1]

Gibbs Sampler algorithm example

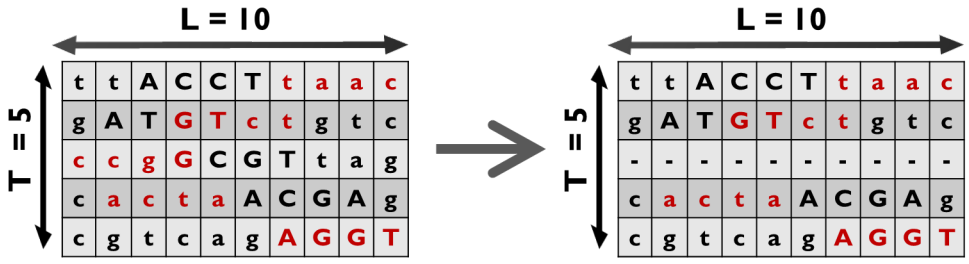

Figure 2.5: At first, the algorithm has chosen the following 4-mers (shown in red) and has randomly selected the third string for removal.

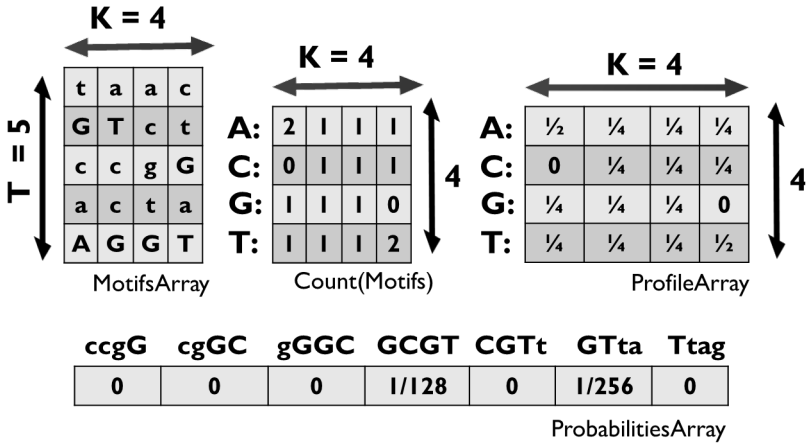

Figure 2.6: Create CountArray, ProfileArray, ProbabilitiesArray.

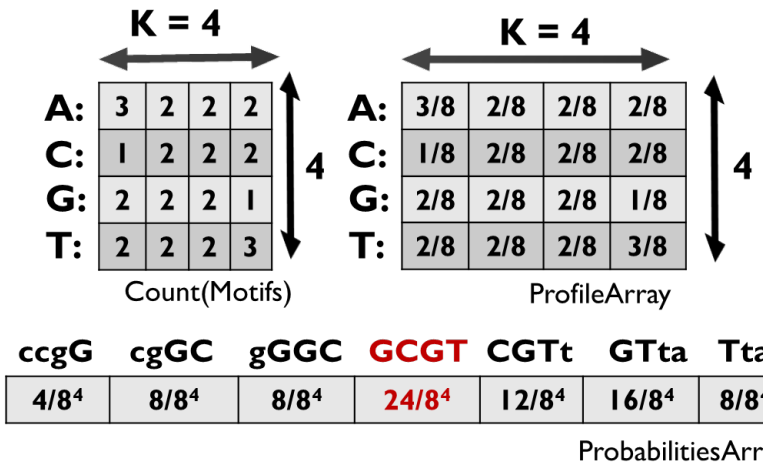

Figure 2.7: Update Count and ProfileArray with pseudocount, and use this ProfileArray to compute the ProbabilitiesArray of all 4-mers in the deleted string ccgGCGTtag.

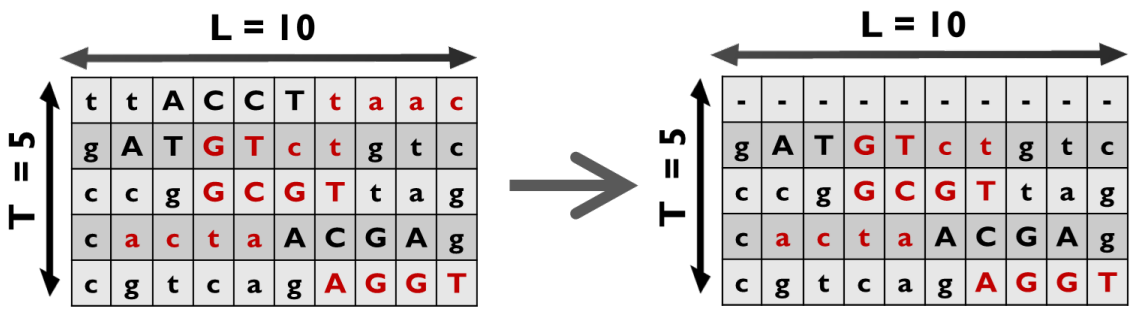

Figure 2.8: The deleted string ccgGCGTtag is now added back to the collection of motifs, and GCGT substitutes the previously chosen ccgG in the third string in Dna.

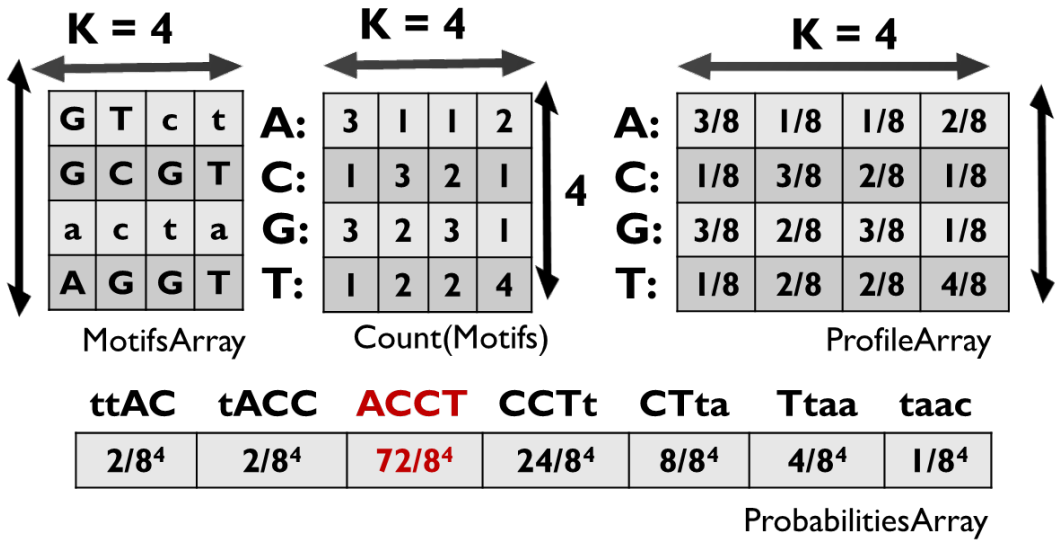

Figure 2.9: Create CountArray, ProfileArray, ProbabilitiesArray. Then update Count and ProfileArray with pseudocount, and use this ProfileArray to compute the ProbabilitiesArray of all 4-mers in the deleted string ttACCTtaac. The deleted string ttACCTtaac is now added back to the collection of motifs, and ACCT substitutes the previously chosen taac in the first string in Dna.

The capital letters in this example represent the BestMotifs that Gibbs Sampler will eventually return after as many iterations as it was set to do. Already in the example, we have found the two out of the four motifs that we are searching. After a few iterations if lines 2 and 4 are picked we will get BestMotifsArray. This is how Gibbs Sampler algorithm works. [1]

2.2 Tools used for timing analysis of C code

For better understanding which parts of the algorithm were taking the most time to execute, so that we could optimize it on the best possible way, we used Gprof tool and `clock()` function.

2.2.1 Gprof tool

Gprof [10] is a tool used for performance analysis of Unix applications. The way it works is by automatically inserting instrumentation code into the program code during compilation (specifically for the GCC compiler the user needs to use the '-pg' option), to gather caller-function data. Sampling data is saved in `gmon.out` file before the program exits, the user can analyze it with the 'gprof' command-line tool.

Gprof output consists of two parts: the flat profile and the call graph:

1. The flat profile provides the total execution time spent in each function and its percentage of the total running time. It also contains function call counts. The output is sorted in the order of the higher percentage.

2. The second part of the output is the textual call graph, which shows for each function which one called it (parent) and which it called (child subroutines).

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 97.14      0.63     0.63     2000     0.32     0.32  generate_probabilities_of_all_kmers
  1.54      0.64     0.01     4000     0.00     0.00  score
  1.54      0.65     0.01     2000     0.01     0.01  profile_with_pseudocounts_without_i
  0.00      0.65     0.00     2000     0.00     0.32  generate_random_kmer
  0.00      0.65     0.00        3     0.00     0.00  lenHelper
  0.00      0.65     0.00        1     0.00   651.47  gibbs_sampler
```

Figure 2.10: Example of a flat profile produced by Gprof tool.

```
                     Call graph (explanation follows)

granularity: each sample hit covers 2 byte(s) for 0.28% of 3.60 seconds

index % time    self  children    called     name
                0.00    3.60       1/1          main [2]
[1]    100.0    0.00    3.60       1          gibbs_sampler [1]
                0.08    3.45    2000/2000        generate_random_kmer [3]
                0.04    0.00    4000/4000        score [5]
                0.03    0.00    2000/2000        profile_with_pseudocounts_without_i [6]
                0.00    0.00    2213/4213        mystrncpy [7]
                0.00    0.00    2010/2010        myrand [8]
-----------------------------------------------|
                                                 <spontaneous>
[2]    100.0    0.00    3.60                 main [2]
                0.00    3.60       1/1          gibbs_sampler [1]
                0.00    0.00       3/3          lenHelper [9]
-----------------------------------------------
                0.08    3.45    2000/2000        gibbs_sampler [1]
[3]     98.1    0.08    3.45    2000         generate_random_kmer [3]
                3.45    0.00    2000/2000        generate_probabilities_of_all_kmers [4]
                0.00    0.00    2000/4213        mystrncpy [7]
-----------------------------------------------
                3.45    0.00    2000/2000        generate_random_kmer [3]
[4]     95.8    3.45    0.00    2000         generate_probabilities_of_all_kmers [4]
-----------------------------------------------
                0.04    0.00    4000/4000        gibbs_sampler [1]
[5]      1.1    0.04    0.00    4000         score [5]
-----------------------------------------------
                0.03    0.00    2000/2000        gibbs_sampler [1]
[6]      0.8    0.03    0.00    2000         profile_with_pseudocounts_without_i [6]
-----------------------------------------------
                0.00    0.00    2000/4213        generate_random_kmer [3]
                0.00    0.00    2213/4213        gibbs_sampler [1]
[7]      0.0    0.00    0.00    4213         mystrncpy [7]
-----------------------------------------------
                0.00    0.00    2010/2010        gibbs_sampler [1]
[8]      0.0    0.00    0.00    2010         myrand [8]
-----------------------------------------------
                0.00    0.00       3/3          main [2]
[9]      0.0    0.00    0.00       3         lenHelper [9]
-----------------------------------------------
```

Figure 2.11: Example of a call graph produced by Gprof tool.

2.2.2 Clock() function

The function `clock()` is included in the library `time.h` and it returns an approximation of processor time used by the program. In particular, the value that is returned is the CPU time used so far as a clock_t type of variable. In order to get the number of seconds used, the user needs to divide by `CLOCKS_PER_SEC`. If the processor time used is not available or its value cannot be represented, the function returns the value (clock_t) = -1.

2.3 OpenMP

OpenMP [11] (Open Multi-Processing) is an API that supports shared-memory multiprocessing in C. It is an extension of C (or C++) language, using a set of compiler directives, environment variables and library routines that influence the execution of the program and enabling the opportunity to run a program concurrently within a multi-core system.

OpenMP's directives are called `pragmas`. Using `pragmas` a user is able to create threads, run loops in parallel, create critical sections, insert barriers and many more functions. [12] This is what makes OpenMP an easy-to-use library, combining great functionality with user-friendly interface.

In OpenMP, the main thread (called `master`) is the primary thread of the execution. Whenever we create sub-threads and tasks, the framework forks the specified number of threads. All these threads, may run concurrently, taking advantage of all possible and available cores in the system. Each thread is associated with a single `ID`. The user can create parallel regions where needed, using as many threads as needed. The only limitations are the computer resources and algorithms restrictions, a program (or specified parts of a program) may run only by the master, or from any number of threads. A similar structure is shown in 2.15, where the master thread forks different number of threads to run parallel regions. [2]

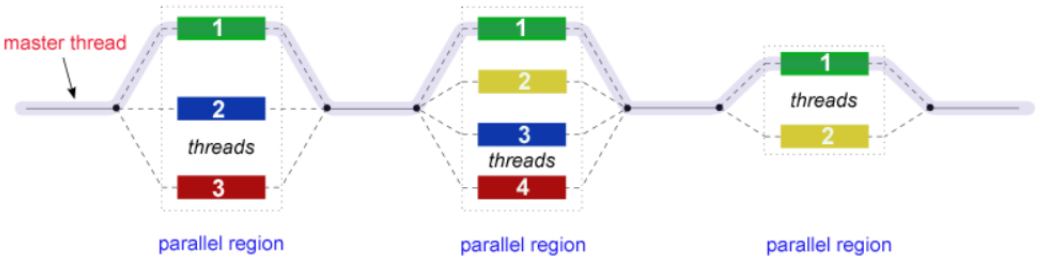

Figure 2.12: An illustration of multithreading where the primary thread forks off a number of threads which execute blocks of code in parallel. [2]

2.4 Vitis Unified Software Platform

The Vitis unified software platform makes possible the development of embedded software and accelerated applications on heterogeneous Xilinx platforms, including FPGAs and SoCs. When it comes to FPGA-based acceleration, the Vitis core development kit allows the

user to build a software application using an API, like the OpenCL API, to run hardware kernels on accelerator cards. The Vitis core development kit also supports running the software application on an embedded processor platform running Linux, such as on Zynq UltraScale+ MPSoC devices. As for the embedded processor platform, the Vitis core development kit execution model also uses the OpenCL API and the Linux-based Xilinx Runtime to schedule the HW kernels and control data movement. [3, 4, 13]

For this thesis, for the hardware development stage were used Vitis Unified Platform and Vivado HLS 2019.2. [3, 14]

Figure 2.13: Vitis unified software platform overview. [3, 4]

2.5 OpenCL

Applications take less time to run with OpenCL, because they are offloading their most computationally intensive code onto accelerator processors or devices. OpenCL's developers use C or C++ based kernel languages to code programs that are passed through a device compiler for parallel execution on accelerator devices.

An OpenCL application is split into host and device parts. The host code is written using a general programming language such as C or C++ and it is compiled by a conventional compiler for execution on a host CPU. The device compilation phase can be done online, for example during the execution of an application, using special API calls. It can alternatively be compiled, before executing the application, into the machine binary or a special representation. [5, 15]

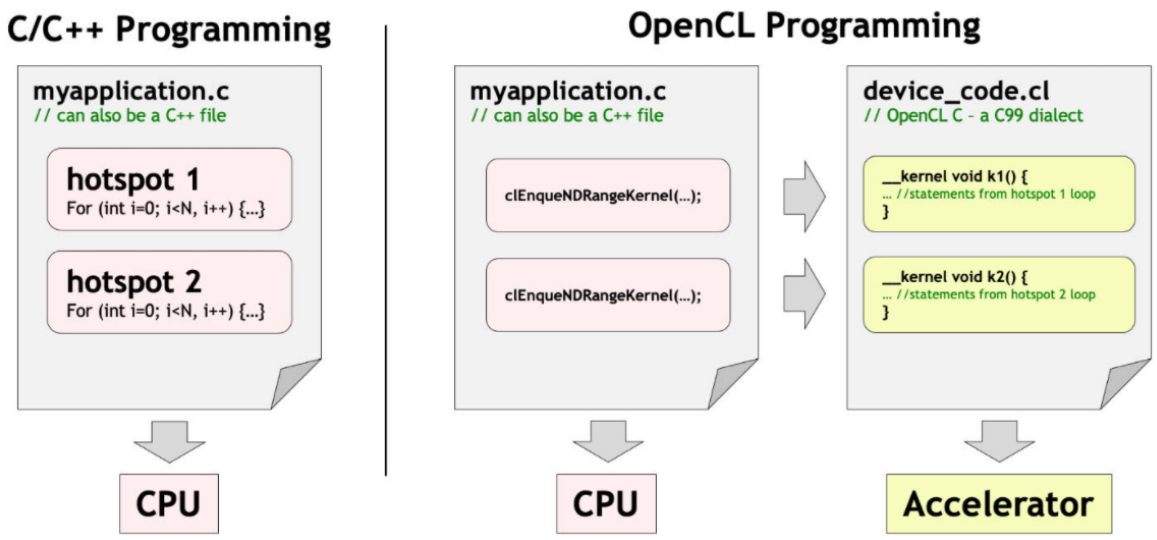

Figure 2.14: Traditional vs OpenCL programming example. [5]

2.6 Vivado HLS tool

The Xilinx Vivado High Level Synthesis (HLS) tool transforms a C specification into a register transfer level (RTL) implementation that you can be synthesized into a Xilinx field programmable gate array (FPGA).

The user can write C specifications in C, C++, or SystemC, and the FPGA provides a massively parallel architecture with benefits in performance, cost, and power over traditional processors.An optimized implementation is created by High-Level Synthesis based on default behavior, constraints, and any optimization directives the user specifies. Also, it is up to the user to use optimization directives to modify and control the default behavior of the internal logic and I/O ports. This allows the generation of variations of the hardware implementation from the same C code. [14]

2.7 FPGA Technology

Field-programmable gate array (FPGA) is an integrated circuit that is built around a uniform matrix of configure logic blocks, that are interconnected to each other on a grid. [16] A programmer, can enable different combinations using the grid and an FPGA can reconfigure to execute different applications. This one of the biggest advantages, if we compare them to CPUs or GPUs, as it is a semiconductor that is able to be reconfigured and run applications at the lowest level, compared to CPUs and GPUs that are built to run under certain

circumstances and all applications go through a different pipeline. Not only that, but FPGAs provide significant advantages compared to other semiconductor devices, such as low latency and energy efficiency. Due to their reconfigurable nature, FPGAs are now used in fields such as Aerospace, Automotive, Medical, Video Processing and more and more people are getting involved with their research. FPGAs have an extreme growth thought the last decade, and are expected to be used more and more in the upcoming years.

A Programmer is able to specify an FPGA configuration using (usually) a hardware description language (HDL), such as Verilog or VHDL. It is recently that High-Level Synthesis has taken over the place, and using state-of-the-art tools, users are able to write a high level language, such as C, C++, OpenCL, and be able to synthesize it using tools such as Vitis Platform [4].

2.7.1 MPSoC FPGA

MultiProcessor System-on-Chip FPGA, is a technology that combines a processor and a reconfigurable architecture into a single device. This provides high performance, low bandwidth communication through the processor and the FPGA. Ultrascale+ ZCU102 MPSoC [6] is used in this Thesis as the target platform.

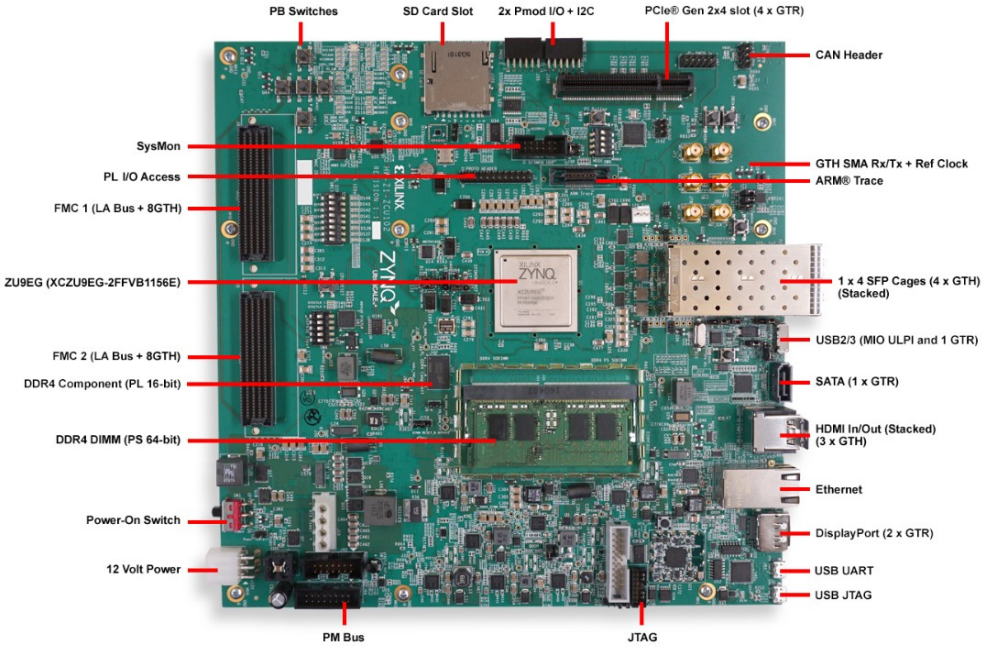

Figure 2.15: ZCU102 Evaluation Board [6]

Chapter 3

Implementation & Optimizations

This chapter describes how the Gibbs sampler algorithm for motif finding was implemented in C and what transformations of the C code needed to be made to implement it in an FPGA device. It also describes the optimizations that were made using OpenMP to run it with multithreading in an x86 processor, and the optimizations done on the FPGA device.

3.1 C Implementation

The pseudocode of Gibbs sampler had first to be implemented in C, in order to get results from x86. It consists of four important functions which are presented as pseudocode in Fig 3.1, Fig 3.2, Fig 3.3, Fig 3.4 and some functions less important, not included in this thesis.

Function `GibbsSampler` presented in Fig 3.1 is the main function of this program as it the one that calls all the other functions on its main loop of `N` iterations, where `N` is given by the user. It takes as an input `Dna`(which is a two-dimensional array of `T` DNA sequences of (`L` length), `T`(integer), `K`(integer which defines how long will the produced motifs be), `L`(for DnaLength). It returns the result in the form the two-dimensional(2D) array `BestMotifs`, that has a size `K * T`, where `K` is the length of the motifs and is given by the user and `T` is the number of DNA sequences that are given as input. Gibbs Sampler is a function that contains the element of randomness, as it uses the function `rand()` both to determine where in the DNA do the initial motifs and best motifs start, and to pick one of the motifs to be excluded in each iteration of the calculation of the profile matrix. The excluded motif, depending on the `ProbabilitiesArray` that is constructed, changes with the one that has the greatest probability. If the score of newly changed `MotifsArray` is less than the score

of BestMotifsArray, then BestMotifsArray takes the values of MotifsArray.

Function Profile presented in Fig 3.2 creates a two-dimensional(2D) array ProfileArray, of size 4 * K, where 4 stands for the number of nucleotides(adenine, cytosine, guanine and cytosine).

Function Score presented in Fig 3.3 calculates the score of the motifs array, which it takes as input. Score is the result of adding for each column the instances of the nucleotides that appears fewer times (for example, if in column there are more adenine than every other nucleotide, then score for this particular column equals the result of adding all cytosine, guanine and thymine of this column).

Function Probabilities presented in Fig 3.4 creates a one-dimensional array ProbabilitiesAr of size L - K + 1, where L is the size of Dna array line. ProbabilitiesArray contains the probability of each motif of length K (k-mer).

```
1  Gibbs_Sampler(Dna, k, t, N, L)
2      for i ← 1 to T
3          StartOfMotif = Random(DnaLength - K + 1);
4          Motifs(i, 0..K) = Dna(i, StartOfMotif .. StartOfMotif + K)
5          BestMotifs(i, 0..K) = Motifs(i, 0..K)
6      for j ← 1 to N
7          i ← Random(t)
8          ProfileArray = ProfileNot_i(Motifs, k, t, i, pseudocount)
9          ProbabilitiesArray = Probabilities(Dna(i), L, k, ProfileArray)
10         MostProbable_j = Get max from ProbabilitiesArray(max_j)
11         Motifs(i, 0..K) = Dna(i, MostProbable_j.. MostProbable_j + K)
12         ScoreMotifs = Score(Motifs, k, t, output);
13         ScoreBestMotifs = Score(BestMotifs, k, t, output);
14         if(ScoreMotifs < ScoreBestMotifs)
15           BestMotifs(i, 0..K) = Motifs(i, 0..K)
16      return BestMotifs
```

Figure 3.1: C pseudocode for Gibbs Sampler [1]

```
ProfileNot_i(Motifs, k, t, i_to_exclude, pseudocount)
    ProfileArray = {pseudocount}
    for j ← 1 to K
        count_a = 0; count_c = 0; count_g = 0; count_t = 0;
        for i ← 1 to T
          if (i == i_to_exclude) continue;
          if(Motifs(i,j) == 'A') count_a++;
          else if(Motifs(i,j) == 'C') count_c++;
          else if(Motifs(i,j) == 'G') count_g++;
          else if(Motifs(i,j) == 'T') count_t++;
        sum = count_a + count_c + count_g + count_t + 4 * pseudocount;
        ProfileArray(0,j) = (ProfileArray(0,j) + count_a) / sum;
        ProfileArray(1,j) = (ProfileArray(1,j) + count_c) / sum;
        ProfileArray(2,j) = (ProfileArray(2,j) + count_g) / sum;
        ProfileArray(3,j) = (ProfileArray(3,j) + count_t) / sum;
    return ProfileArray
```

Figure 3.2: C pseudocode for `Profile` function [1]

```
Score(Motifs, k, t)
    Score = 0;
    for j ← 1 to K
        count_a = 0; count_c = 0; count_g = 0; count_t = 0;
        for i ← 1 to T
          if(Motifs(i,j) == 'A') count_a++;
          else if(Motifs(i,j) == 'C') count_c++;
          else if(Motifs(i,j) == 'G') count_g++;
          else if(Motifs(i,j) == 'T') count_t++;
        if (count_a == max)  Score += count_c + count_g + count_t;
        else if(count_c == max) Score += count_a + count_g + count_t;
        else if(count_g == max) Score += count_a + count_c + count_t;
        else if(count_t == max) Score += count_a + count_c + count_g;
return Score
```

Figure 3.3: C pseudocode for `Score` function [1]

```
1  Probabilities(DnaLineExcluded, DnaLength, k, ProfileArray)
2      ProbabilitiesArray = {1};
3      for j ← 1 to DNA_LENGTH - K + 1
4          count_a = 0; count_c = 0; count_g = 0; count_t = 0;
5          for i ← 1 to K
6            if(DnaLineExcluded(i,j) == 'A')
7              ProbabilitiesArray(i)*= ProfileArray(0,j);
8            else if(DnaLineExcluded(i,j) == 'C')
9              ProbabilitiesArray(i)*= ProfileArray(1,j);
10           else if(DnaLineExcluded(i,j) == 'G')
11             ProbabilitiesArray(i)*= ProfileArray(2,j);
12           else if(DnaLineExcluded(i,j) == 'T')
13             ProbabilitiesArray(i)*= ProfileArray(3,j);
14 return ProbabilitiesArray
```

Figure 3.4: C pseudocode for `Probabilities` function [1]

3.2 OpenMP Optimizations

In order to optimize the performance of the algorithm, OpenMP was used to utilize the multiple cores of the processor. The main part of the algorithm which causes its latency to increase is the multiple for loops, most of which are also nested for loops, so we decided to try executing them in parallel. The pragma that was used for this purpose was `pragma omp parallel for` [17]. In some cases we needed to keep some variables `private`, which means that they cannot be accessed by other threads, but most of them could be `shared` between the threads.

3.3 FPGA Implementation & Optimizations

The main purpose of this thesis is to observe how this algorithm could be accelerated on an FPGA device. So, we had to implement the Gibbs sampler algorithm in a way that it could be run on an FPGA, then optimize it and observe the way that it performs.

3.3.1 FPGA implementation

The core of the algorithm remained the same for its FPGA implementation, but we needed to do some drastic changes because some things that are supported by the software are not an option when it comes to hardware.

1. The first C function that is not supported by Vivado HLS is function `malloc()` because it is using resources that exist in the memory of the operating system, which are created and released during run time. In order to be able to synthesize a hardware implementation the design must be fully self-contained and to specify all the resources that it is going to need. [18]. So, all the arrays that were previously stored in dynamically allocated memory needed to be transformed to static arrays.

2. The second C function that is not supported by Vivado HLS is `rand()`, which was used to produce the random numbers that are needed for the functionality of a random algorithm. This function is not supported as it cannot be synthesized So, it had to be replaced with a pseudo number generator function that imitates its activity, which was created manualy.

3. The third C function that is not supported by Vivado HLS is function `strncpy()`, which was used to copy `BestMotifs`, `Motifs` and `Dna` arrays and was critical for the correctness of the algorithm, so we changed it with a function with the same functionality that we implemented.

4. The fourth thing that needed to change was the dimensions of both the input array `Dna` and the output array `BestMotifs`. The reason for that was the OpenCL framework that was used. In fact, a two-dimensional(2D) array can not be passed as a kernel argument in OpenCL, and it has to be reduced into a one-dimensional(1D) array. [15] Dna array was reduced to an 1D array of size `L * T`, where `L` is the length and `T` is the number of the Dna sequences. `BestMotifs` array was also reduced to an 1D array, in this case of a size of `K * T`, where `K` is the length of the motifs that the algorithm tries to construct.

It is worth mentioning that for the Gibbs sampler algorithm to be able to be syntesized on Vivado HLS, we had to implement a host function that was the one that gathered all the data that the user gave and then sent it to the `Gibbs_Sampler` function. As for the Vitis

Unified Software platform, where the actual FPGA on which we did the measurements was connected, we had to also create a host function with OpenCL language, which also gathered the information and after the proper processing then sent it to the kernel that it created.

3.3.2 FPGA optimizations

Comparing the results that can be found in Chapter 4 related with the x86 performance and the initial run on the FPGA we observe quite an important difference, with x86 being significantly faster than the FPGA. This means that in order to use the FPGA on its full extent, we need to further optimize the algorithm.

Precise optimizations

1. We tried to utilize the BRAM of the FPGA device by using the function `memcpy()` to copy both the `Dna` array that comes from the host to a local variable that is stored in BRAM, but also `BestMotifsArray` from a local variable that is stored in BRAM to the variable that is sent by the host. These optimizations resulted on a very high drop of the time the algorithm needed to run on the FPGA.

2. We used **`pipeline pragma`**, which reduces the initiation interval(II) for a function or a loop by allowing the concurrent execution of operations. A loop or a function that is pipelined can process new inputs every N clock cycles (N = II of the loop or function).Loop pipelining can be prevented by loop carry dependencies so this pragma was used along with **`dependence pragma`**, which is presented next. [19]

3. We also used **`dependence pragma`**, which provides additional information and enables us to overcome some loop carry dependencies and allows loops to be pipelined with lower II. This was exactly the result that we got, as we had a further drop of II. [20]

4. Additionally, we tried **`inline pragma`**, which removes a function as a separate entity. Sometimes, inlining a function may allow operations within the function to be shared and optimized more effectively with surrounding operations. As this is a complex algorithm that calls many functions this was a useful optimization. [21]

5. Moreover, we used **`array_partition pragma`**, which splits an array in multiple BRAMS and as a result it increases the amount of read and write ports and improves the

throughput and the design. In this particular algorithm, we partitioned Dna 2D array in dimension 2 with a factor of DnaLength/2 in order to have `DnaLength/2` memory ports and allow for multiple loads from this array on the same time, which resulted in to reduce II. We also partitioned arrays `BestMotifs` and `Motifs` completely in dimension 2, `ProfileArray` and `ProbabilitiesArray` in dimension 1 with a factor 4, in order to reduce further the II. [22]

6. We also used **`unroll pragma`**, which transforms loops by creating multiple copies of the loop body in the RTL design. This allows the parallel execution of some or all loop iterations and can increase data accesses and throughput. All the loops were completely unrolled in an effort to reduce II, which was finally achieved. [23]

7. We attempted to use **`loop_flatten pragma`**, which allows nested loops to be flattened into a single loop and has as a result improved latency. It is important to be mentioned that only perfect and semi-perfect loops can be flattened. Imperfect loop nests are the loops whose inner loop has variable bounds, or its loop body is not completely inside the inner loop. [24]

8. We tried to eliminate the function that we had made to resemble function `strncpy()`, because it caused high latency by using pointer logic. So, we replaced it with a fully unrolled version(that we unrolled by hand), which had the same functionality but far better results.

9. Finally, we tried to eliminate the function that we had made to resemble the function `rand()`, because it caused high latency, as it did multiplications every time it was called in order to produce a pseudo random number. Instead, we replaced it with two buffers(of sizes `T` and `N` because this is how many times this functions was called to produce some random numbers in different ranges.These buffers were created by the host and stored in BRAM of the FPGA in the beginning of the algorithm. This optimization also produced great results.

Approximate optimization

Gibbs sampler result depends on the number of the iteration of its main loop, which every time produces a new `MotifsArray`, which is then compared with `BestMotifsArray` and if its `Score` is lower, then `BestMotifsArray` it takes the values of `MotifsArray`. The number of iterations is very high, and most of the time higher than needed, in order to ensure that the results are as close to the optimal solution as they can be. So, our idea was to reduce the number of the iterations, depending on the initial number of iterations. For example, from 2000 iterations we reduced them to 500, that is 75% less than the original number of iterations. We noticed that this resulted in a significant reduction of the time needed to run the algorithm on the FPGA device and most importantly the results remained optimal. We tried to reduce the iterations further, but the results weren't close to the optimal results.

Chapter 4

Results & Analysis

This chapter presents the results of the acceleration of the Gibbs sampler algorithm for motif finding. Firstly, there are presented the results on x86 and ARM processors after the implementation in C, and an analysis of the time of each function of the algorithm to analyze which is the part that takes the most time to execute. Secondly, there are presented the results on x86 after the OpenMP optimizations. Thirdly, there are presented the results after the alterations of the algorithm in order to be implemented to run on the FPGA device. Fourthly, there are presented the results after precise and approximate optimizations on the FPGA device. Lastly, all those results are combined, and we make some conclusions regarding the algorithm.

4.1 Testcase analysis

In this section, we are going to present the example that most of the metrics were taken from. The inputs for Gibbs sampler algorithm are the `DnaArray` shown in Figure 4.1, `T` = 10 which is the number of DNA sequences inside DnaArray, `L + 1` = 1974 which is the length of the sequences in `DnaArray`, `K` = 15 which is the size of the motifs that will be produced and `N` = 2000 which is the number of iterations that Gibbs sampler algorithm has to do, in order to converge to an optimal `BestMotifs` array shown in Figure 4.4, which is returned by function `GibbsSampler`. While it is executed, it also creates array `Motifs` shown in Figure 4.4.

Now we are going to talk about the functions that are called in the main loop of `N` = 2000 iterations of `GibbsSampler`. Function `ProfileNot_i` produces `ProfileArray` shown

in Figure 4.2 and function `Probabilities` produces `ProbabilitiesArray` shown in Figure 4.3.

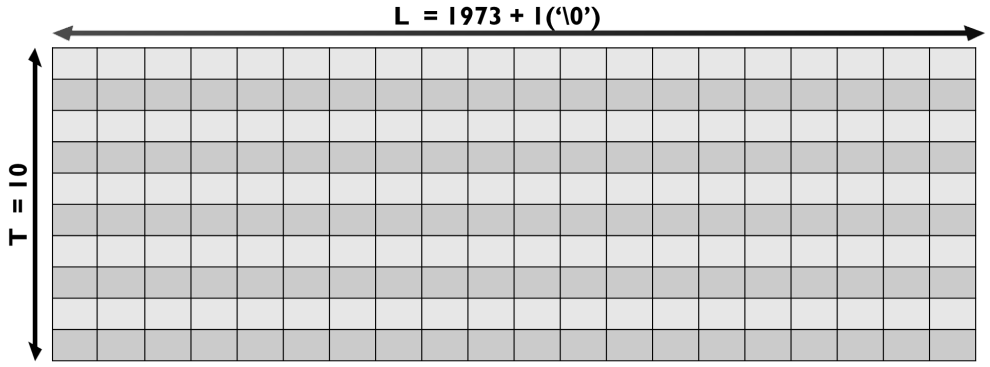

Figure 4.1: `DnaArray` of size 1974 * T.

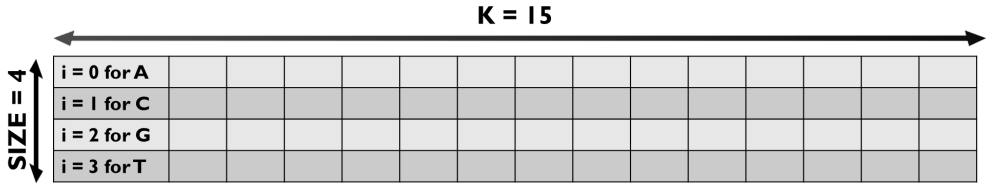

Figure 4.2: `ProfileArray` of size 4 * K.

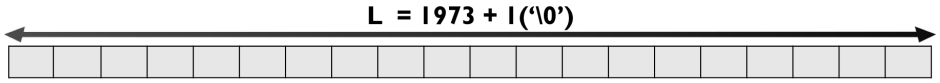

Figure 4.3: `Probabilities` array of size L - K + 1.

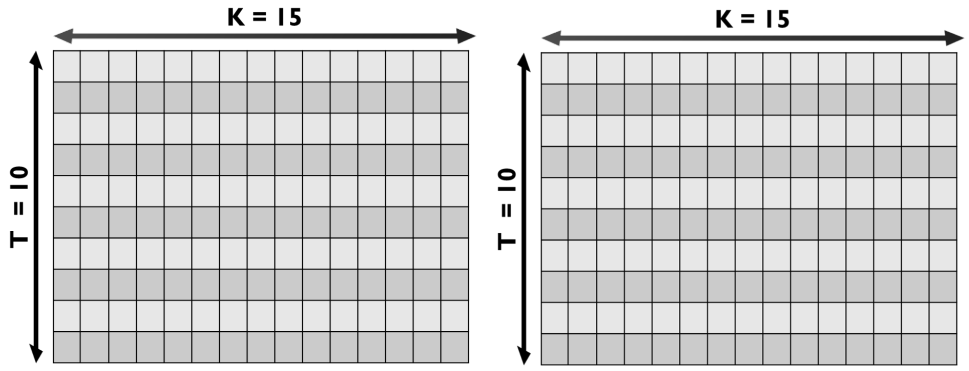

Figure 4.4: `Motifs` and `BestMotifs` arrays of size K * T.

4.2 C Implementation results

After implementing the pseudocode for Gibbs sampler algorithm in C, we analyzed its performance, regarding how much time it takes to produce the `BestMotifs` array using gprof tool and `clock()` function, and how much time each of its functions take to complete. In this section are presented the results of this analysis for x86 and ARM processors.

Results and analysis for x86 proccessor

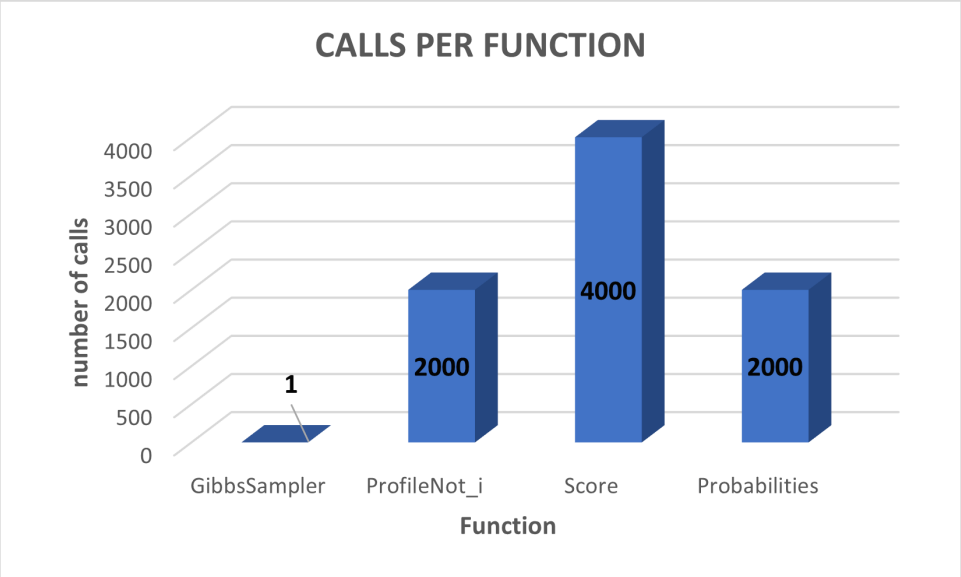

Figure 4.5: Bar chart of how many times a function is called.

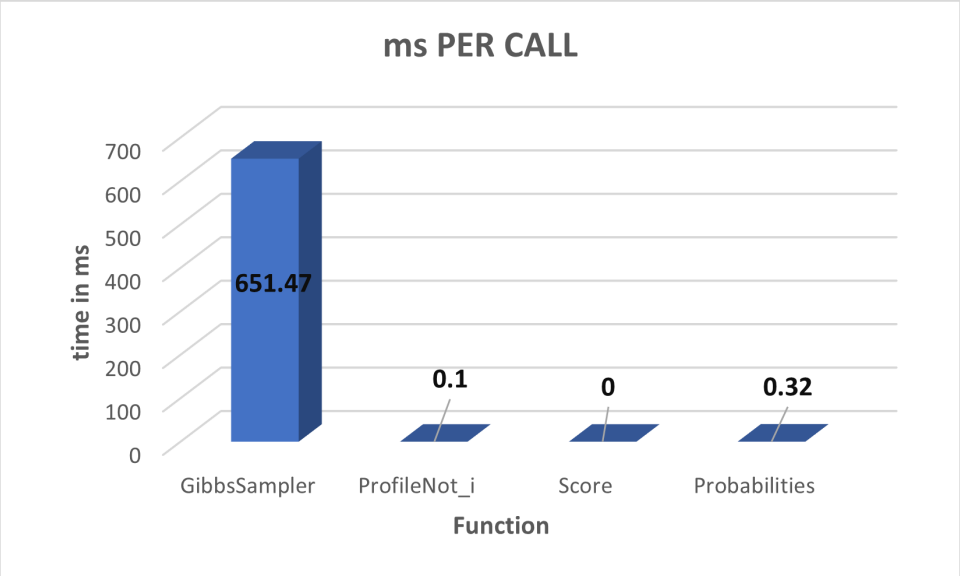

Figure 4.6: Bar chart of the time in ms that each call of the functions take on x86.

As shown in Figure 4.5 and Figure 4.6 function `GibbsSampler` is called only 1 time but as it calls all the other functions in its main loop of N = 2000 iterations it takes the most time to complete, in this case 651.47 ms. On the other hand, the function `Score` may be called the most times (N * 2 = 4000), as in each iteration the score of `Motifs` and `BestMotifs` arrays are compared, it takes near to zero time to complete. The subfunction that takes the most time per call to complete is `Probabilities` function, which takes 0.32 ms per call to complete, and it is called 2000 times. The subfunction `ProfileNot_i` takes 0.1 ms per call to complete, and is called 2000 times.

Results and analysis for ARM proccessor

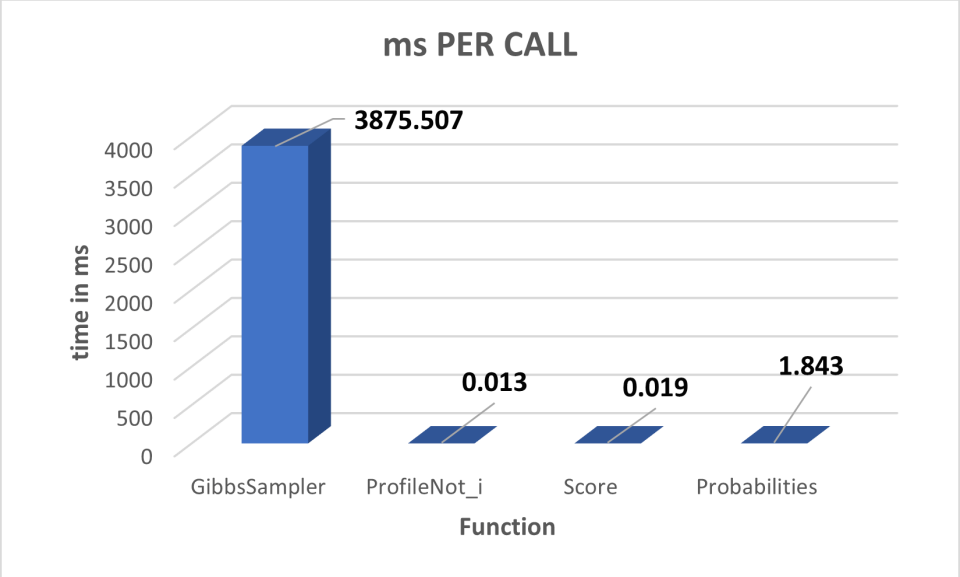

Figure 4.7: Bar chart of the time in ms that each call of the functions take on ARM.

As shown in Figure 4.7 function `GibbsSampler` takes the most time to complete, in this case 3875.507 ms. The subfunction that takes the most time per call to complete is `Probabilities` function, which takes 1.843 ms per call to complete. The function `Score` takes 0.019 ms, and the function `ProfileNot_i` takes 0.013 ms per call to complete.

Comparison of x86 proccessor and ARM processor results

As we can see in Figure 4.6 and Figure 4.7, x86 processor takes 651.47 ms to run Gibbs sampler algorithm and ARM processor 3875.507 ms. So, x86 is **5.94 times faster** than ARM processor.

4.3 OpenMP results for x86 processor

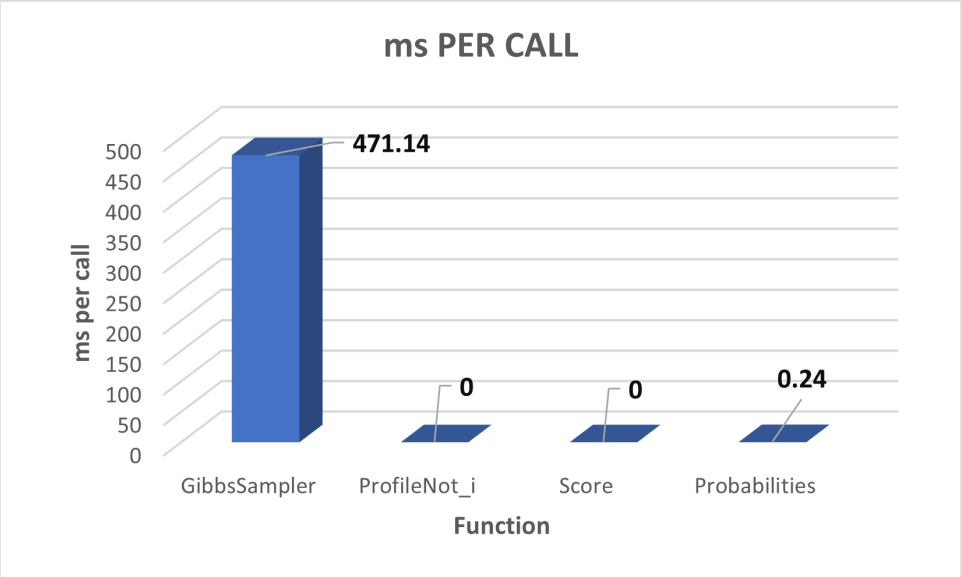

Figure 4.8: Bar chart of the time in ms that each call of the functions take on x86 after OpenMP optimizations.

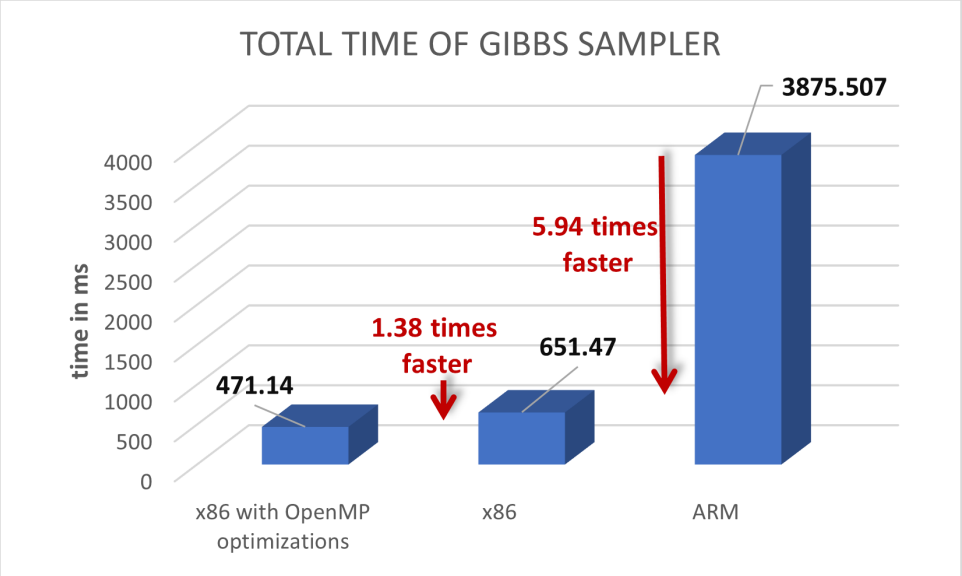

Figure 4.9: Bar chart of the time in ms that Gibbs sampler takes to run in ARM, x86 and x86 with OpenMP optimizations.

As we can see in Figure 4.6 and Figure 4.8, x86 processor takes 651.47 ms to run Gibbs sampler algorithm without OpenMP optimizations and 471,14 ms with OpenMP optimizations. This speedup is expected as we parallelized all the loops that was possible. The speedup

is not very high as there are many dependencies between the functions of the program, and even between some loop iterations inside those functions. Nevertheless, x86 with OpenMP optimizations runs Gibbs sampler algorithm **1.38 times faster** than x86 without OpenMP optimizations.

In Figure 4.9 we can see a comparison between the time that Gibbs sampler algorithm takes to run in x86 and ARM processor without OpenMP optimizations and in x86 with OpenMP optimizations. It also shows how much faster x86 is from ARM and x86 with OpenMP optimizations, from x86 without them.

4.4 FPGA Implementation results

After doing all the changes that were essential for the algorithm to be able to run on an FPGA device, we finally got to the stage that we could run the Gibbs sampler algorithm on the FPGA. The initial results were a bit disheartening compared to x86 processor with and without optimizations, but were better than the results that we got from ARM processor.

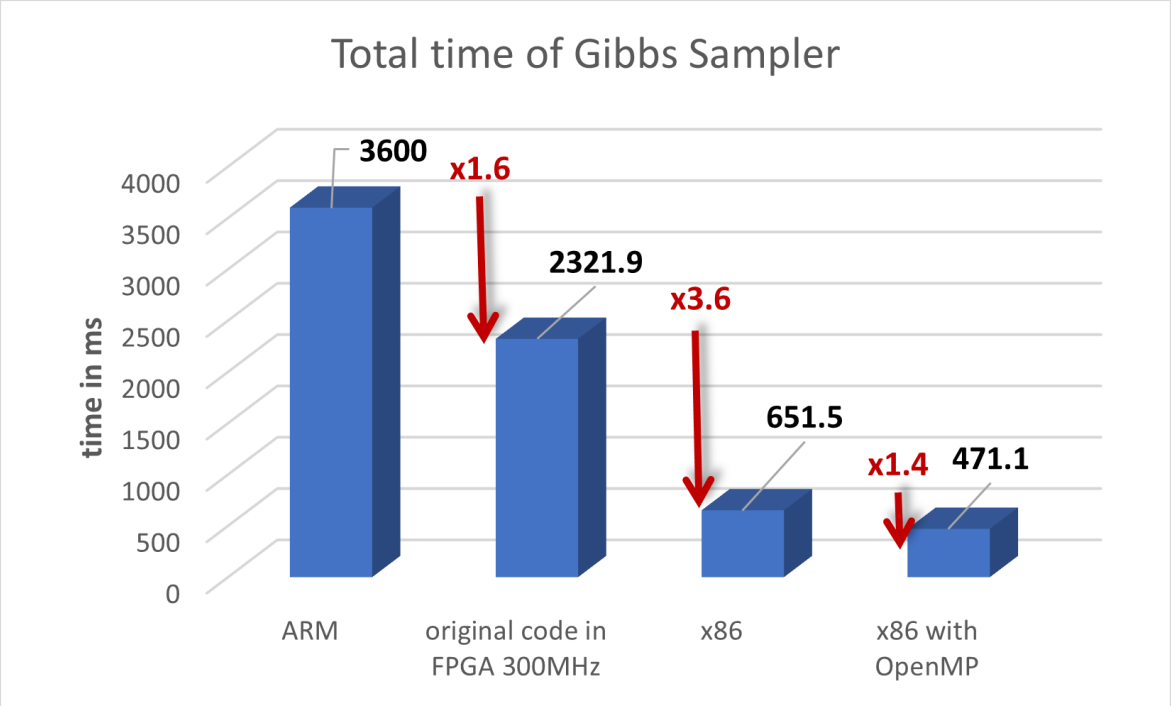

Figure 4.10: Bar chart of the time in ms that Gibbs sampler takes in FPGA with clock = 300 MHz compared with ARM, and x86 processor with and without OpenMP optimizations.

As we can see in the Figure 4.10 the code of Gibbs sampler algorithm is significantly slower when it is run on the FPGA compared with the time that it takes on x86 processor.

In fact, it is 3.6 times slower than x86 without OpenMP optimizations and 4.9 times slower than x86 with OpenMP optimizations.n On the other hand, it is 1.6 times faster than ARM processor.

4.5 FPGA optimization results

There is no doubt that some changes need to happen in order to utilize the full capacity of the FPGA, and as a result get better measurements for Gibbs sampler algorithm.

4.5.1 Memory burst

Transfer input array on BRAM

The first optimization that we tried was copying the input data of the `Dna` array to the BRAM in order to be able to access them faster. This optimization had a huge impact on the time that the algorithm run, even if the II was still 10.

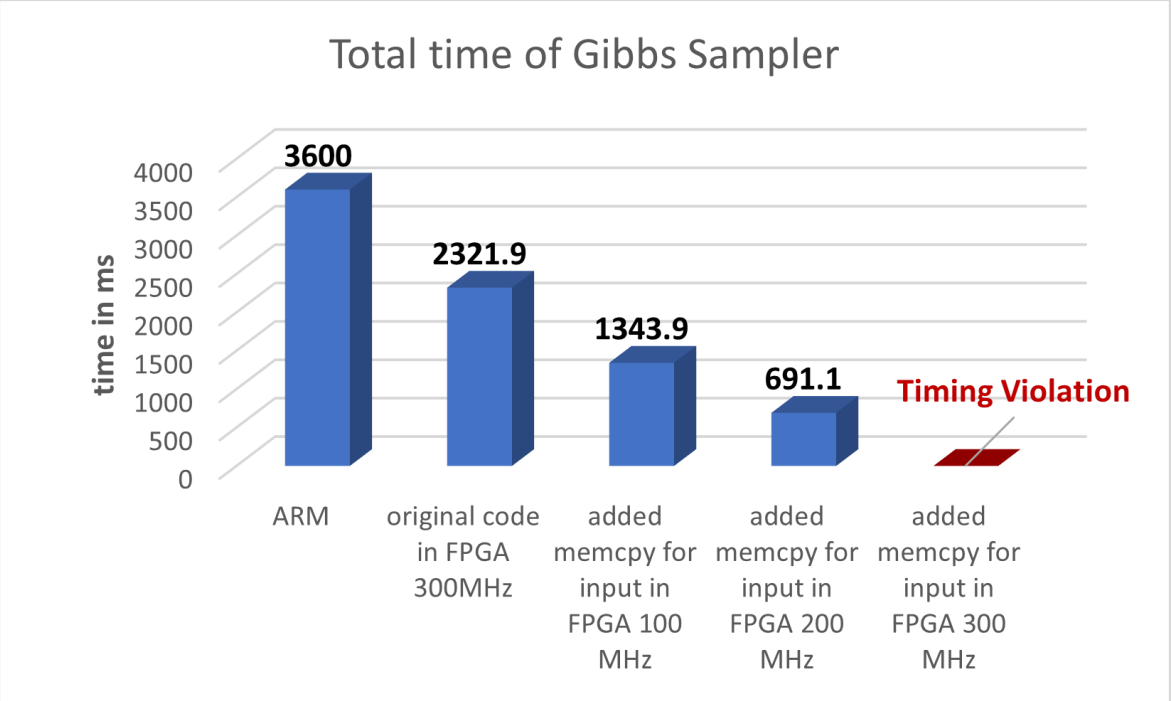

Figure 4.11: Bar chart of the time in ms that Gibbs sampler takes to run in FPGA with utilizing the BRAM for the input array compared with the previous results of the FPGA device and ARM processor.

As we can see in the Figure 4.11 the algorithm now takes less time than it did previously

on the FPGA without this optimization, with a current clock on 200MHz and a previous one on 300 MHz. The structure of the algorithm with the multiple for loops, most of which are nested, caused the synthesis to fail on 300 MHz with this optimization, because a timing violation was detected.

As we can see in Figure 4.12, the time that it takes for Gibbs sampler algorithm to run on the FPGA, is approaching the time that it takes to run on x86.

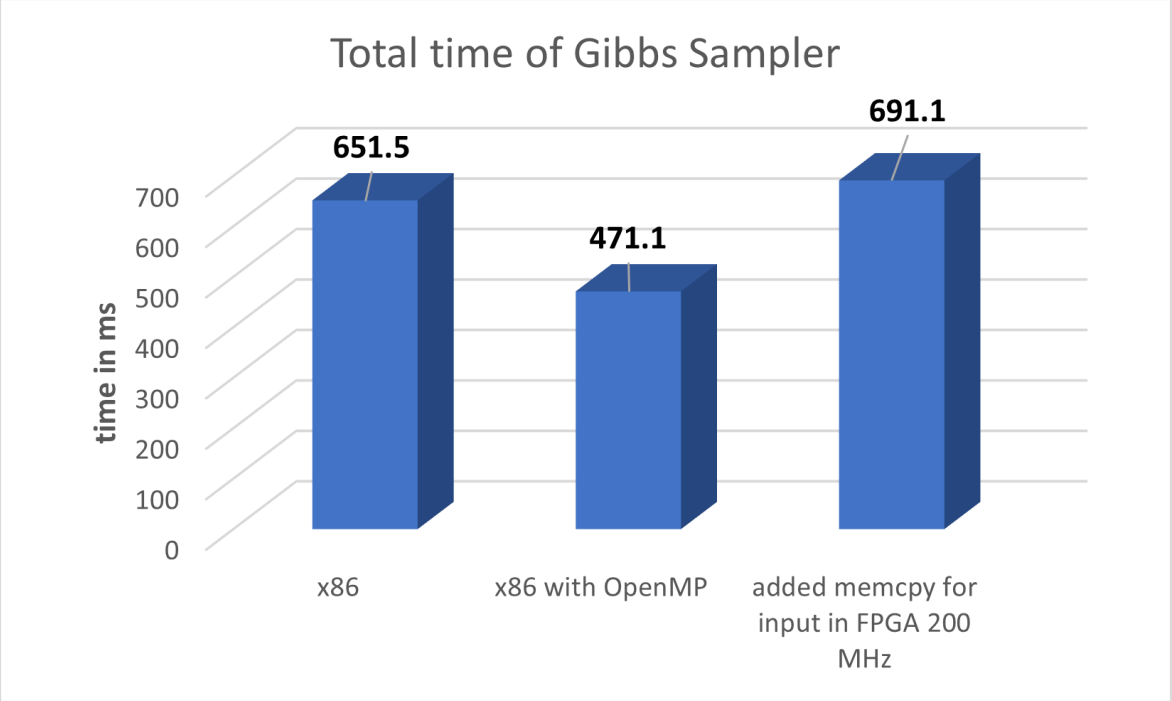

Figure 4.12: Bar chart of the time in ms that Gibbs sampler takes to run in FPGA with utilizing the BRAM for the input array compared with the previous results x86 processor with and without OpenMP optimizations.

Transfer output array on BRAM

After copying the input data of the `Dna` array to the BRAM, we tried copying the output array `BestMotifsArray` to the BRAM when the algorithm finished so that we do not have to access host memory whenever we try to read or write from this array. This optimization also had a huge impact on the time that the algorithm run, even if the II was 15.

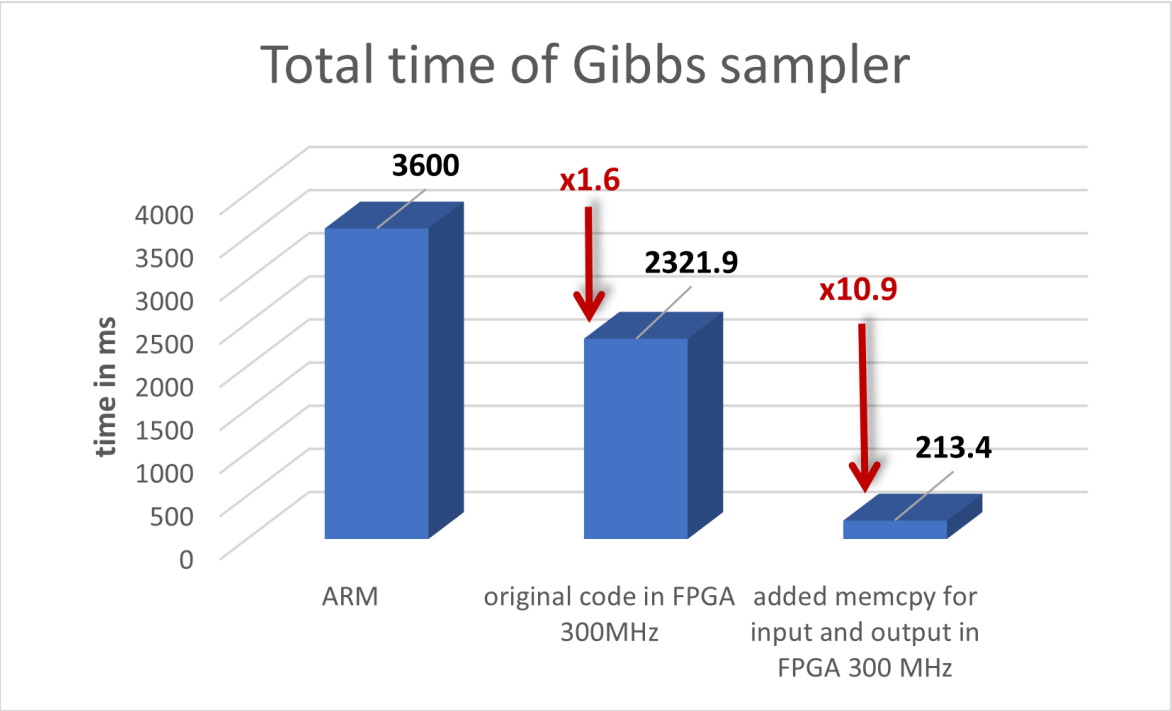

Figure 4.13: Bar chart of the time in ms that Gibbs sampler takes to run in FPGA with utilizing the BRAM for the input and the output arrays, compared with the previous results of the FPGA device and ARM processor.

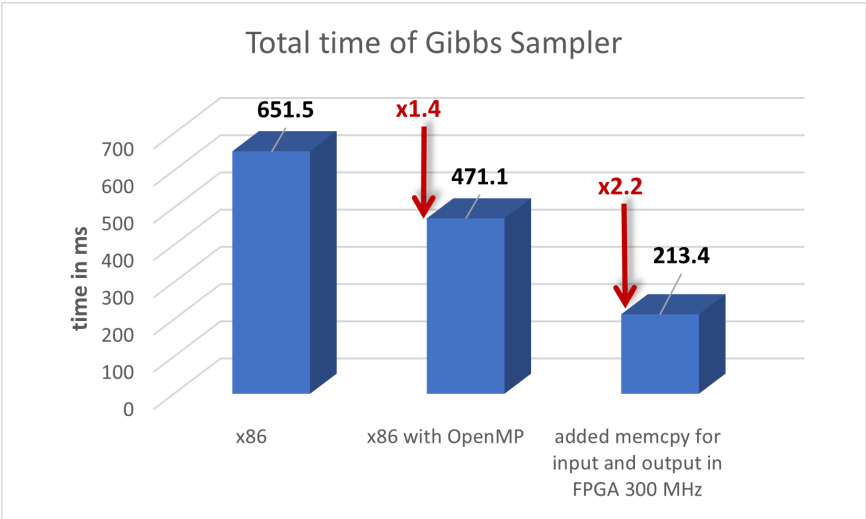

Figure 4.14: Bar chart of the time in ms that Gibbs sampler takes to run in FPGA with utilizing the BRAM for the input and the output arrays, compared with the previous results x86 processor with and without OpenMP optimizations.

As we can observe on Figure 4.13 we have achieved a time of 213.4 ms which is quite good compared with ARM processor, which is 16.87 times slower than that. The original

code was also far slower, to be exact, 12.78 times slower than the current time that we had achieved with a clock of 300 MHz, even if the II was still 10. Regarding Figure 4.14 which presents the speedup that we have accomplished in comparison with the time that the algorithm takes to run on x86 processor, we can clearly see that the code on the FPGA is without OpenMP optimizations and 2.21 times faster with OpenMP optimizations. The II was 15 and the percentage of the LUTs was 28%.

4.5.2 II optimizations

Using all the optimizations that we mentioned in Chapter 3 with different variations we tried to reduce II to see if we could get better results. There were many attempts using all the `pragmas` mentioned also in Chapter 3 combining them with removing the functions that we made to imitate functions `rand()` and `strncpy`. At this point we need to clarify that we are talking about an algorithm that has many loops, most of which are nested, and many `if`, `else if` and `else` statements. This made our job very hard, because many of the optimizations that we tried failed due to timing violations and/ or created more than 80% LUTs. Below are shown the results of these attempts.

Results for II = 8

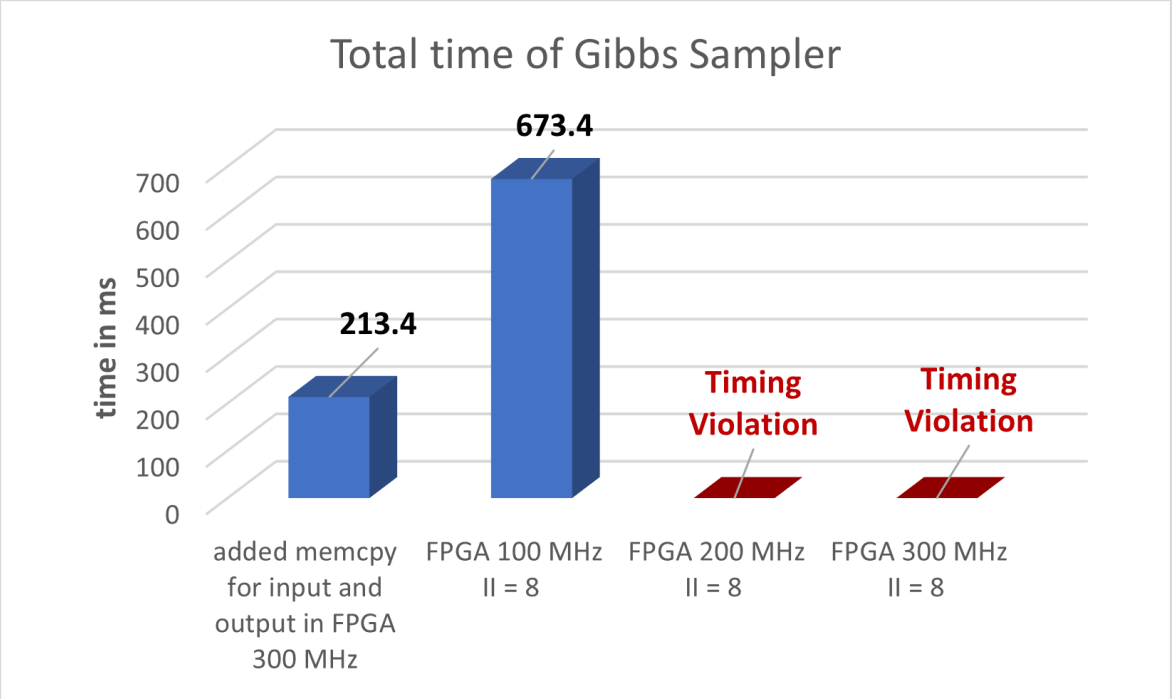

Figure 4.15: Bar chart of the time in ms that Gibbs sampler takes to run in FPGA with II = 8.

The II dropped to 8 when we replaced the function we had created to imitate the C function `strncpy()` with simple variable assignments, which also copied the contents of the two strings, but more effectively. We also changed the function that imitated the `rand()` function with two arrays which contains all the values that this function returned and was now made by the host to avoid some computations in the loops. We applied `memcpy()` to transfer those arrays, that were passed as an input to the FPGA, to the BRAM, we also partitioned them with `array_partition pragma`, one of them completely and the other one in dimension 1 with a block factor 4. We also used `pipeline pragma`, `unroll pragma`, `inline pragma`, `tripcount pragma` and `flatten pragma`. Finally, we used `array_partition pragma` to partition the `Dna` array in dimension 2 with a block factor of 1974/2 = 987, `ProbabilitiesArray` and `ProfileArray` in dimension 1 with a block factor of 4, `MotifsArray` and `BestMotifsArray` in dimension 1 with a block factor of 4.

As we can observe on Figure 4.15 we have achieved a time of 673.4 ms with the clock set to 100 MHz, but at 200 MHz and 300MHz we had timing violations. Even though the II was improved, we were incapable to run the algorithm on a high clock frequency, so the results that we got were as expected worse than those with the addition of `memcpy()` for input and output arrays. To be exact, the newly obtained result was 3.2x slower than the best result that we had gotten yet. The percentage of the LUTs was 56%.

Results for II = 4

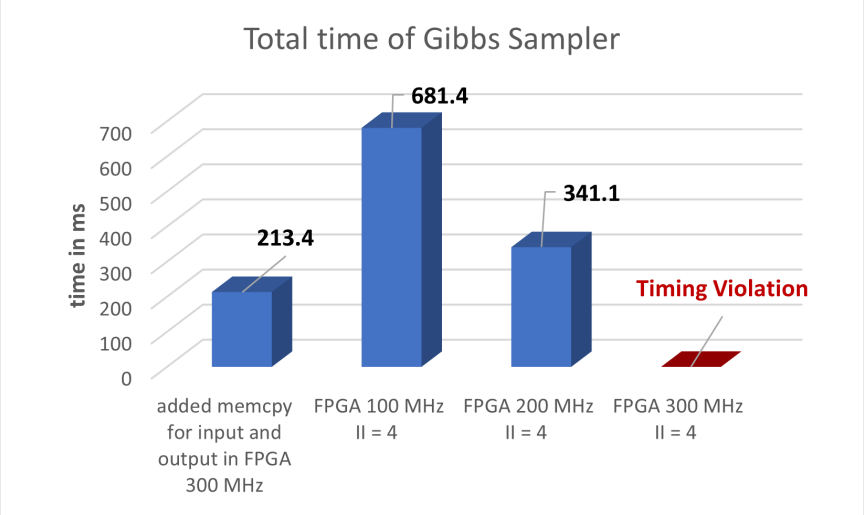

Figure 4.16: Bar chart of the time in ms that Gibbs sampler takes to run in FPGA with II = 4.

The II dropped to 4 when we used `dependance pragma`, to determine that in function `Score`, variable `Score` did not have a dependence in the same loop iteration(intra dependence), and variable `Motifs` did not have a dependence between different loop iterations(inter dependence). [20] We also used `array_partition pragma` to partition `MotifsArray` and `BestMotifsArray` completely in dimension 2 instead of partially with a block factor 4 in dimension 1.

As we can observe on Figure 4.16 we have achieved a time of 341.1 ms with the clock set to 200 MHz, 681.42ms with the clock set to 100 MHz, and with the clock set to 300MHz we had timing violation. Even though the II was improved, we were incapable to run the algorithm on a 300MHz clock frequency, so the results that we got were as expected worse than those with the addition of `memcpy()` for input and output arrays. The percentage of the LUTs was 47%.

To be exact, the newly obtained result was 1.6 times slower than the best result that we had gotten yet.

Results for II = 1

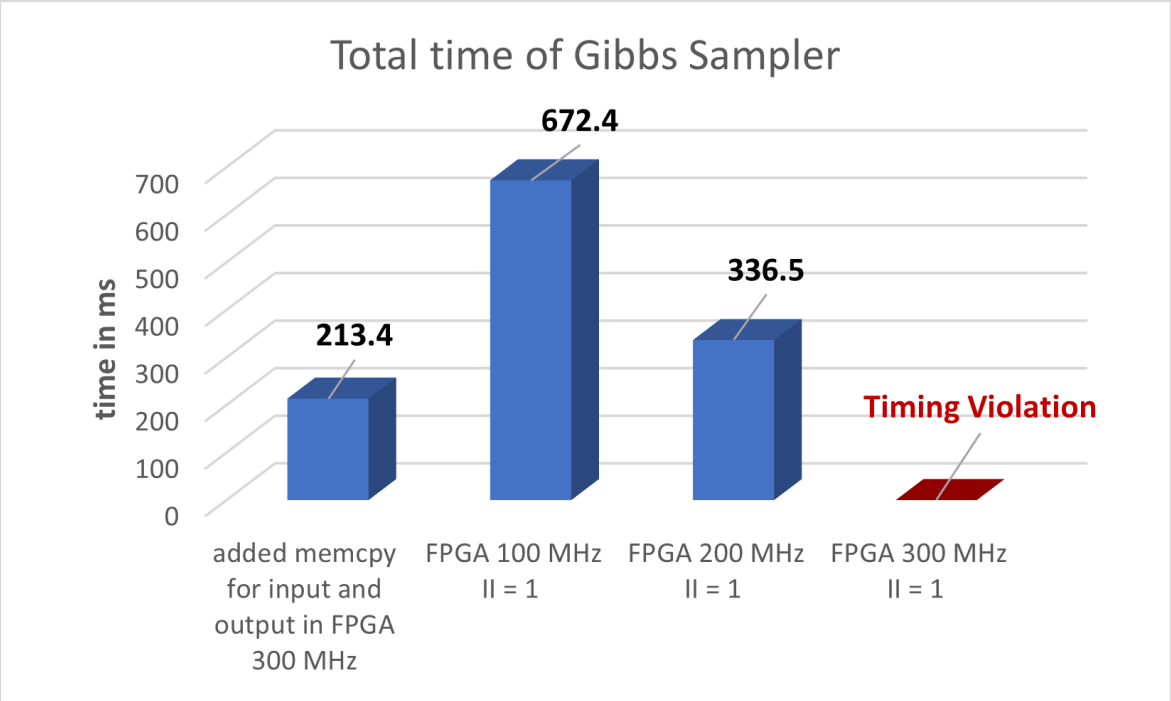

Figure 4.17: Bar chart of the time in ms that Gibbs sampler takes to run in FPGA with II = 1.

The II dropped to 1 when we used `dependance pragma`, to determine that in function `ProfileNot_i`, variable `ProfileArray` did not have a dependence between different

loop iterations(inter dependence). We also used `array_partition pragma` to partition `ProfileArray` both in dimension 1 and 2 with a block factor 4, to do that we also needed to convert `ProfileArray` to a 2D array, instead of an 1D that it was previously.

As we can observe on Figure 4.16 we have achieved a time of 336.48 ms with the clock set to 200 MHz, 672.37 ms with the clock set to 100 MHz, and with the clock set to 300MHz we had timing violation. Even though the II was improved, we were incapable to run the algorithm on a 300MHz clock frequency, so the results that we got were as expected worse than those with the addition of `memcpy()` for input and output arrays. The percentage of the LUTs was 70%.

To be exact, the newly obtained result was 1.58 times slower than the best result that we had gotten yet.

4.5.3 Approximate optimization

After trying all those things, we had to think of another way to see if we could further reduce the time, the Gibbs sampling algorithm needed to run on the FPGA. By reducing N, the iterations of the main function of this algorithm, we didn't have any problem with the results, which were still very accurate.

Results for N = 500

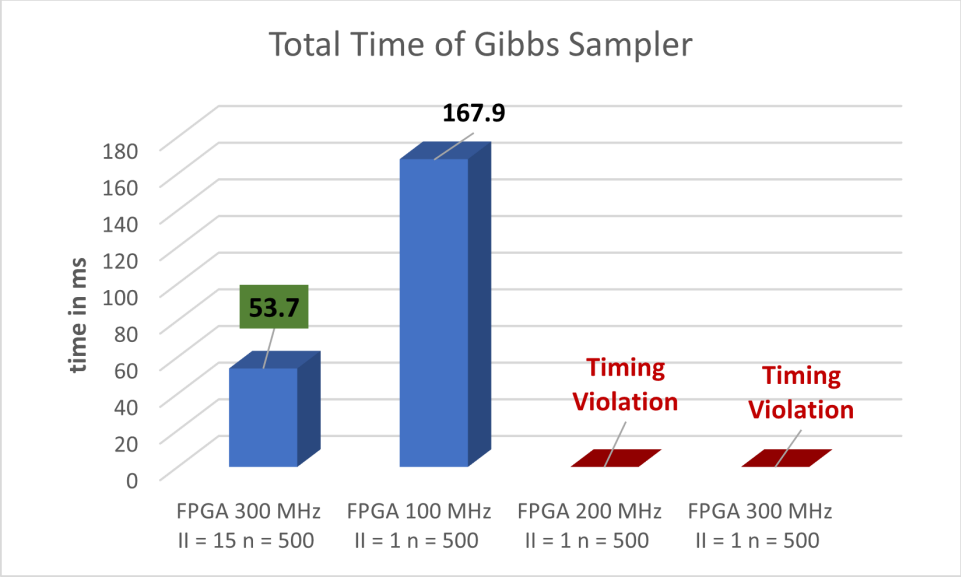

Figure 4.18: Bar chart of the time in ms that Gibbs sampler takes to run in FPGA with N = 500.

As we can see in Figure 4.18 even though we still can't get results with II = 1 and the clock on 300MHz or 200MHz, we did get results with the clock on 300 MHz for II = 15 that were pretty good, compared with what we had started with. Also, we got results for II = 1, with a clock set to 100MHz. As we can see, the results are better than all our previous tries. With green highlight we have the best time that we have achieved yet, 53.75ms.

4.5.4 Final Results

After all those experiments, we can see in Figure 4.19 that we were able to accelerate Gibbs Sampler algorithm quite a lot. Even though, we did not manage to run it with a clock frequency set to 300 MHz and II = 1, we still got very good results with II = 15. This algorithm had a lot of loops with inter loop dependencies and many conditional statements, which made it harder for us to drop II efficiently and not simultaneously increase the critical path of the main loop of Gibbs sampler and the percentage of the LUTs, resulting in a timing violation when routing was taking place.

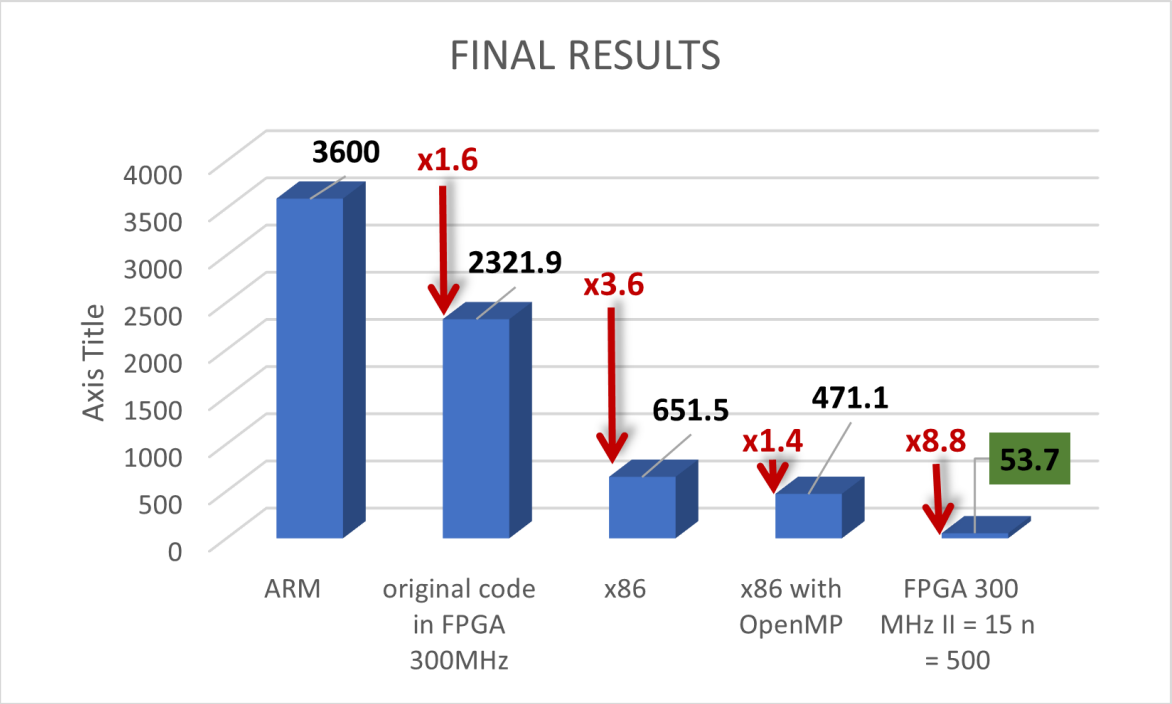

Figure 4.19: Comparison of the results of ARM processor, baseline FPGA implementation, x86 processor with and without OpenMP optimizations, and optimal FPGA implementation.

As we can see in Figure 4.19 the best result that we got from the FPGA device is 67x faster than ARM processor, 43.2x faster than the original code without optimizations on the

FPGA, 12.1x higher faster than x86 without OpenMP optimizations and 8.8x faster than x86 with OpenMP optimizations.

Table 4.1: Table of all the results combined.

Run	*Time in ms*
ARM	3600
original code in FPGA 100MHz	2727.24
original code in FPGA 300 MHz	2321.92
x86	651.47
x86 with OpenMP	471.14
FPGA *memcpy* input 100 MHz	1343.88
FPGA *memcpy* input 200 MHz	691.099
FPGA *memcpy* input 300 MHz	**Timing violation**
FPGA *memcpy* input output 200 MHz	318.998573
FPGA *memcpy* input output 300 MHz	213.39547
FPGA 100MHz II = 8	673.392767
FPGA 200MHz II = 8	**Timing violation**
FPGA 300MHz II = 8	**Timing violation**
FPGA 100MHz II = 4	681.419784
FPGA 200MHz II = 4	341.093692
FPGA 300MHz II = 4	**Timing violation**
FPGA 300MHz II = 15 N = 500	**53.744152**
FPGA 100MHz II = 1 N = 500	167.876326
FPGA 200MHz II = 1 N = 500	**Timing violation**
FPGA 300MHz II = 1 N = 500	**Timing violation**
FPGA 100MHz II = 1	672.366548
FPGA 200MHz II = 1	336.48165
FPGA 300MHz II = 1	**Timing violation**

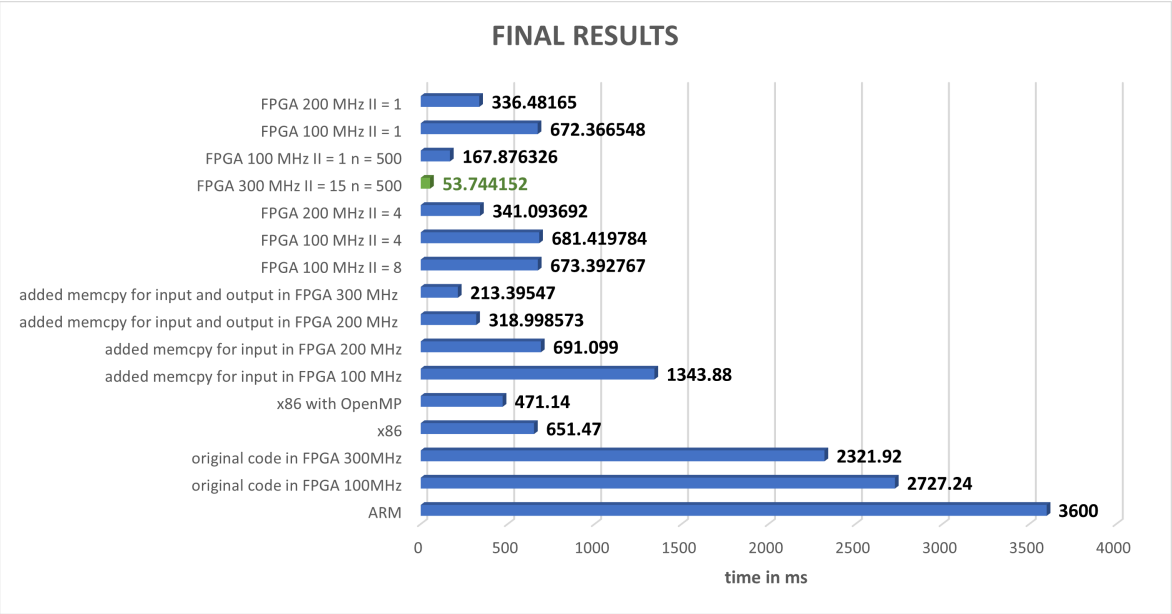

Figure 4.20: Combination of the results of the implementation and optimizations on software and hardware.

4.5.5 Memory utilization

Table 4.2 shows the percentage of memory utilization for each optimization that we tried on the FPGA device. Table 4.3 shows the total number of the available resources of the FPGA.

Table 4.2: Table of memory utilization.

	$LUT(\%)$	$FF(\%)$	$BRAM(\%)$	$DSP(\%)$
Baseline Implementation	3	1	0	0
II = 15 (Optimal Implementation)	28	18	2	16
II = 8	56	4	1	0
II = 4	47	4	1	0
II = 1	70	4	2	0

Table 4.3: Table of FPGA available resources.

	LUT	FF	$BRAM$	DSP
Available resources	274080	548160	1824	2520

Chapter 5

Conclusions

This chapter will summarize this thesis and draw some conclusions regarding the acceleration of Gibbs Sampler algorithm, both in software and in hardware.

5.1 Summary & Conclusions

The purpose of this thesis was to analyze the performance of Gibbs sampling algorithm for motifs finding, both in software and hardware, and try optimizing it to achieve a better performance.

Firstly, the algorithm was implemented in C language, and we analyzed its behavior. We observed which of the functions that it called took the most time to complete and how many times each function was called. For the analysis we used Gprof tool and function `clock()`. Then, we took measurements of execution time of the algorithm on an x86 and an ARM processor with the function `clock()`.

Secondly, we tried optimizing the algorithm's performance on software using OpenMP. We also observed which of the functions that it called took the most time to complete and how many times each function was called, just like we did on the version of the code that was not optimized. Then, we took measurements of execution time of the algorithm optimized with OpenMP on an x86 processor with the function `clock()`.

Thirdly, the algorithm was implemented by being altered so that it could be run on an FPGA device. The alterations that needed to be made were the replacement of some C functions with user made functions with the same functionality, and the reduction of the input and output arrays from 2D to 1D arrays, in order to be able to be passed by OpenCL as kernel

arguments. Then, we took measurements of execution time of the algorithm on the FPGA device.

Fourthly, we did several precise and approximate optimizations in order to optimize the performance of the algorithm on the FPGA device. Then, we took measurements of execution time of the algorithm after the optimizations on the FPGA device.

To conclude, as we can see in Figure 4.19, which combines all the results of this thesis, we managed to run the algorithm on the FPGA 67 times faster than ARM processor, 43.2 times faster than the original code without optimizations on the FPGA, 12.12 times higher faster than x86 without OpenMP optimizations and 8.77 times faster than x86 with OpenMP optimizations. So, the purpose of this thesis to accelerate Gibbs sampler algorithm was accomplished.

5.2 Future work

Some possible extensions of this thesis could be the compression of the input array `Dna`. As our input contains only four characters, it is not necessary to store them in a char type variable, but we can convert those values to binary, and they need only two bits to be mapped. For example, 'A' = 00, 'C' = 01, 'G' = 10 and 'T' = 11. This could also reduce the arrays `Motifs` and `BestMotifsArray` and could possibly increase even more the performance of the algorithm.

Another interesting development of this thesis could be running the Gibbs Sampler algorithm on a bigger FPGA device than Zynq UltraScale+ MPSoC ZCU102 that we used. It could be interesting to see if that way we could run the optimized algorithm with II = 1 without getting a timing violation error and see if this further decreases its runtime.

Bibliography

[1] P. Compeau and P. Pevzner. *Bioinformatics Algorithms: An Active Learning Approach*. Active Learning Publishers, 2015.

[2] Openmp wikipedia. `https://en.wikipedia.org/wiki/OpenMP`. Accessed: 2021-07-06.

[3] Vitis unified software platform. `https://www.xilinx.com/products/design-tools/vitis.html`. Accessed: 2021-07-10.

[4] Vitis platform. `https://www.xilinx.com/products/design-tools/vitis/vitis-platform.html`. Accessed: 2021-07-06.

[5] Opencl khronos. `https://www.khronos.org/opencl/`. Accessed: 2021-07-10.

[6] Zynq ultrascale+ mpsoc zcu102 evaluation kit. `https://www.xilinx.com/products/boards-and-kits/ek-u1-zcu102-g.html`. Accessed: 2021-07-06.

[7] Bioinformatics wikipedia. `https://en.wikipedia.org/wiki/Bioinformatics`. Accessed: 2021-07-07.

[8] Achuthsankar S Nair. Computational biology & bioinformatics: a gentle overview. *Communications of the Computer Society of India*, 2:1–12, 2007.

[9] Charles E Lawrence, Stephen F Altschul, Mark S Boguski, Jun S Liu, Andrew F Neuwald, and John C Wootton. Detecting subtle sequence signals: a gibbs sampling strategy for multiple alignment. *science*, 262(5131):208–214, 1993.

[10] Gprof wikipedia. `https://en.wikipedia.org/wiki/Gprof`. Accessed: 2021-07-10.

[11] Leonardo Dagum and Ramesh Menon. Openmp: an industry standard api for shared-memory programming. *IEEE computational science and engineering*, 5(1):46–55, 1998.

[12] Rohit Chandra, Leo Dagum, David Kohr, Ramesh Menon, Dror Maydan, and Jeff McDonald. *Parallel programming in OpenMP*. Morgan kaufmann, 2001.

[13] Xilinx ug1400, howpublished = `https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_2/ug1400-vitis-embedded.pdf`, note = Accessed: 2021-07-10.

[14] Xilinx ug902. `https://www.xilinx.com/support/documentation/sw_manuals/xilinx2020_1/ug902-vivado-high-level-synthesis.pdf`. Accessed: 2021-07-10.

[15] Opencl wikipedia. `https://en.wikipedia.org/wiki/OpenCL`. Accessed: 2021-07-09.

[16] Fpga wikipedia. `https://en.wikipedia.org/wiki/Field-programmable_gate_array`. Accessed: 2021-07-09.

[17] Openmp pragmas. `https://software.intel.com/content/www/us/en/develop/documentation/cpp-compiler-developer-guide-and-reference/top/optimization-and-programming-guide/openmp-support/openmp-pragmas-summary.html`. Accessed: 2021-07-06.

[18] Xilinx ug902. `https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_3/ug902-vivado-high-level-synthesis.pdf`. Accessed: 2021-07-09.

[19] Pipeline pragma xilinx. `https://www.xilinx.com/html_docs/xilinx2017_4/sdaccel_doc/fde1504034360078.html`. Accessed: 2021-07-09.

[20] Dependance pragma xilinx. `https://www.xilinx.com/html_docs/xilinx2017_4/sdaccel_doc/dxe1504034360397.html`. Accessed: 2021-07-09.

[21] Inline pragma xilinx. https://www.xilinx.com/html_docs/xilinx2017_4/sdaccel_doc/jka1504034359550.html. Accessed: 2021-07-09.

[22] Array_partitioning pragma xilinx. https://www.xilinx.com/html_docs/xilinx2017_4/sdaccel_doc/gle1504034361378.html. Accessed: 2021-07-09.

[23] Unroll pragma xilinx. https://www.xilinx.com/html_docs/xilinx2017_4/sdaccel_doc/uyd1504034366571.html. Accessed: 2021-07-09.

[24] Flatten pragma xilinx. https://www.xilinx.com/html_docs/xilinx2017_4/sdaccel_doc/igd1504034366745.html. Accessed: 2021-07-10.