IMPLEMENTATION OF A DNA COMPRESSION ALGORITHM USING DATAFLOW COMPUTING

Informe Final Practica Académica Presentado Como Requisito Para Optar al Título de Ingeniero Electrónico

Modalidad Trabajo de Grado

Rubén David Caro Serna

UNIVERSIDAD DE ANTIOQUIA
1803
FACULTAD DE INGENIEIRÍA

____
Sebastián Isaza Ramírez
Profesor - Asesor

UNIVERSIDAD DE ANTIOQUIA
FACULTAD DE INGENIERÍA
DEPARTAMENTO DE INGENIERÍA ELECTRÓNICA Y TELECOMUNICACIONES
OCTUBRE DE 2018
MEDELLÍN

IMPLEMENTATION OF A DNA COMPRESSION ALGORITHM USING DATAFLOW COMPUTING

Abstract

The amount of DNA sequences databases has increased a lot in the last years, the amount of space required to store the sequences is increasing more than the space available to store them, that means a higher cost to store DNA sequences and also the read sequences which are fragments of the whole sequence. This situation has led to the use of compression algorithms for storing DNA files.

The main objective of the project is to increase the efficiency of the compression of DNA sequences because the process requires a lot of compute. An FPGA with dataflow architecture has been used to develop the project with the aim of exploiting the available parallelism in the algorithm chosen.

The compression method has been developed to process sequence reads with a fixed amount of mutations per read and the test has been developed for 4, 8, 12 and 16 mutations per reads using an architecture that allows up to 160 reads to be processed in only one thick. Experimental results showed that even with a low amount of processing units, the performance increases a lot using the DFE architecture, the only disadvantage is the store/reading time.

Index

1 Introduction

As the time progresses DNA sequencing is easier, this is because this technology is growing up fast, and because of this, the sequencing cost is being reduced. As a consequence, DNA data requires a huge amount of storage capacity.

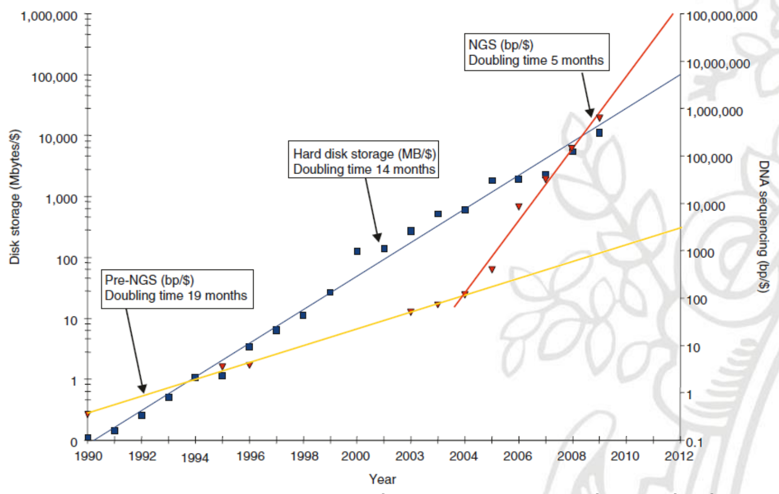


Figure 1. DNA sequencing and storage costs through the time[1]

As described in the Figure 1, even if the cost of DNA sequencing is decreasing fast, the storage cost doesn't decrease in the same rate. The results is that the amount of DNA data is normally larger than the storage capacity available.

To solve this problem, specialized compression algorithms take advantage of redundancies in DNA sequences to achieve higher compression rates. This compression needs high amount of computations to be performed for big sequences and a new alternative to improve the processing time is needed.

In this project, we have chosen a hardware alternative that can be adapted to compression algorithms to achieve a reduction in execution time. Particularly, the programming platform used is a dataflow architecture with a computation model that is ultimately executed on an FPGA.

2 Objectives

2.1 Main Objective

To develop a hardware version of a DNA compression algorithm to improve the compression speed using a FPGA-based computation platform.

2.2 Specific Objectives

- To select a specialized algorithm used in DNA compression with high parallelization opportunities using the most possible hardware in an FPGA.
- To implement the selected algorithm using the specific programming model of the used FPGA, considering especially the data flow of the algorithm.
- To implement optimizations based in the preliminary performance measurements and select the best version of those.
- To make tests on FASTQ files from bacteria, plants and humans with the purpose of characterize the performance of the developed implementation

3 Background

3.1 The DNA sequences

DNA sequences arrays of bases called adenine, cytosine, guanine, and thymine, that is aligned in a selected order. DNA can be read as information that describes a lot of characteristics for an individual and is unique for every individual.

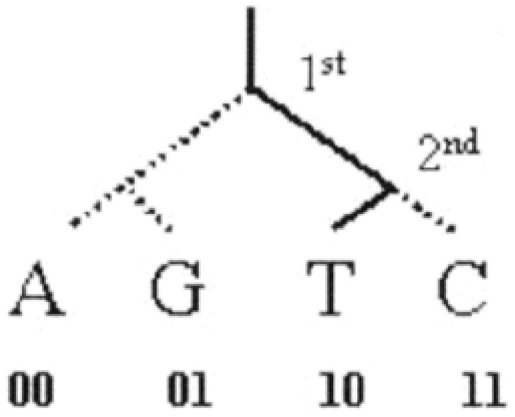


Figure 2. DNA Information representation

The DNA sequence size is very different between the different species, for example, human DNA can hold up to 3 billion of bases that can be 4 possibilities, to be clear, Figure 2 shows a common representation for DNA sequences, each base has 2 bits, that is repeated 3 billion of times, it means about 6 billion of bits.

3.1.1 Reads and full genomes

Reads are fragments of DNA and they are obtained in the sequencing process. They are the essential part to build the whole genome of an individual and they can be a lot bigger than the whole sequence because of the redundancy that is needed to build perfectly the whole genome.

3.1.2 FASTQ Files

FASTQ files are one of the representations for the read sequences, it is a text file that contains a determined number of reads and each one has 3 basic parts: the metadata, the sequences and the quality scores.

The metadata field contains information about the sequencing technology used to produce the data.

The sequence has the base combination of the fragment, it is an array of the nucleotides within the DNA molecule and it also can have N characters which means that there is an undefined base.

Quality scores have the same size of the sequence and for every character it has a corresponding base in the read sequence, it describes the probability for a base to have the exact value.

3.1.3 Read mapping

Read mapping is the process to organize the read according to the position in the sequence. A set of reads is taken, and they are organized in the best way to match each other, these processes are called sequence comparison methods. Those methods can pair all the read sequences to generate the whole genome sequence, some of those methods can be performed through pairwise comparison, database search, statistical analysis, alignment by structural data, etc.

3.2 DNA Compression

The DNA sequences are big, and they are compressed because they have high rate of compression and the size of those sequences need to be reduced to low the storage costs. There are many ways to compress DNA and

3.2.1 General purpose compression

The general-purpose compression is the method to convert some information to another equal information to reduce the size. There are two ways to compress information, lossy and lossless compression.

Lossless and lossy compression are terms that describe whether or not, in the compression of a file, all original data can be recovered when the file is uncompressed. With lossless compression, every single bit of data that was originally in the file remains after the file is uncompressed.

On the other hand, lossy compression reduces a file by permanently eliminating certain information, especially redundant information. When the file is uncompressed, only a part of the original information is still there (although the user may not notice it). In this project we deal with a lossless compression algorithm.

3.2.2 Read compression

The read compression methods are algorithms with the purpose of reducing the storage size for a set of reads. This is an effective mode because the read information has a lot of redundancy and it can have some aspects which can help to reduce significantly the size of this information. There are multiple methods for reads sequence compression.

3.2.3 Referential compression

Referential compression is the method of compressing reads by storing the differences with a previously selected reference data object. This compression type is more complex because it needs to compute differences among very long sequence objects. The referential compression takes one read and look in the whole sequence the best match according to a previously selected metrics and this match will have some properties that explain the process to construct the read by taking a fragment of the reference.

3.3 UdeACompress

UdeACompress[3] is the name of a compression algorithm developed in the High Performance Computing track of the SISTEMIC research lab. It uses a lossless compression method and generates output bytes according to an alignment input data.

3.3.1 Sequence alignment

The first approach before coding one instruction to bits is the process of the read alignment. It consists of comparing every read with the whole genome sequence and generate an output showing all the alignment data. This alignment data has information like mapping position that is useful to build the map an important part to decompress the coded bits into reads and build the codified instruction. The alignment also generates data like the type of alignment, the amount of mutations and data about those mutations.

3.3.2 Inst2Bin function

The inst2bin function is a function which processes alignment data and converts it to an array of bytes. This function processes a read based on the alignment data and store it in one array. The codification of one read doesn't depend on the other read, that is why this function can be accelerated through parallelism.

3.3.3 Prelude, offset and error bits

The inst2bin function has 3 basic parts:

The prelude is composed of 3 basic parts, the more frags flag that indicates the next read is aligned in the same position; the error bit that shows there are errors in the alignment (the offset and error bits exists in that coded read); and the matching bits that show what type of alignment has been performed.

An error or mutation represents the differences between the original read and the fragment read obtained in the sequence where the alignment has been made. The error appears where the fragment of the sequence has differences with the original read and it takes

The offset is divided in two parts, the low bits, 8 bits that are going to be stored in the next position and the high bits that are stored together with the error bits, it represents all the distance between the last error.

The error bits are divided in 3 parts, the more error bit that shows the next 2 bytes are error data, the error type bits that shows the type of mutation which that read have, and the base bit that means which base are replaced in the original read or which distance between the reference base and the read base is.

3.4 The Maxeler platform

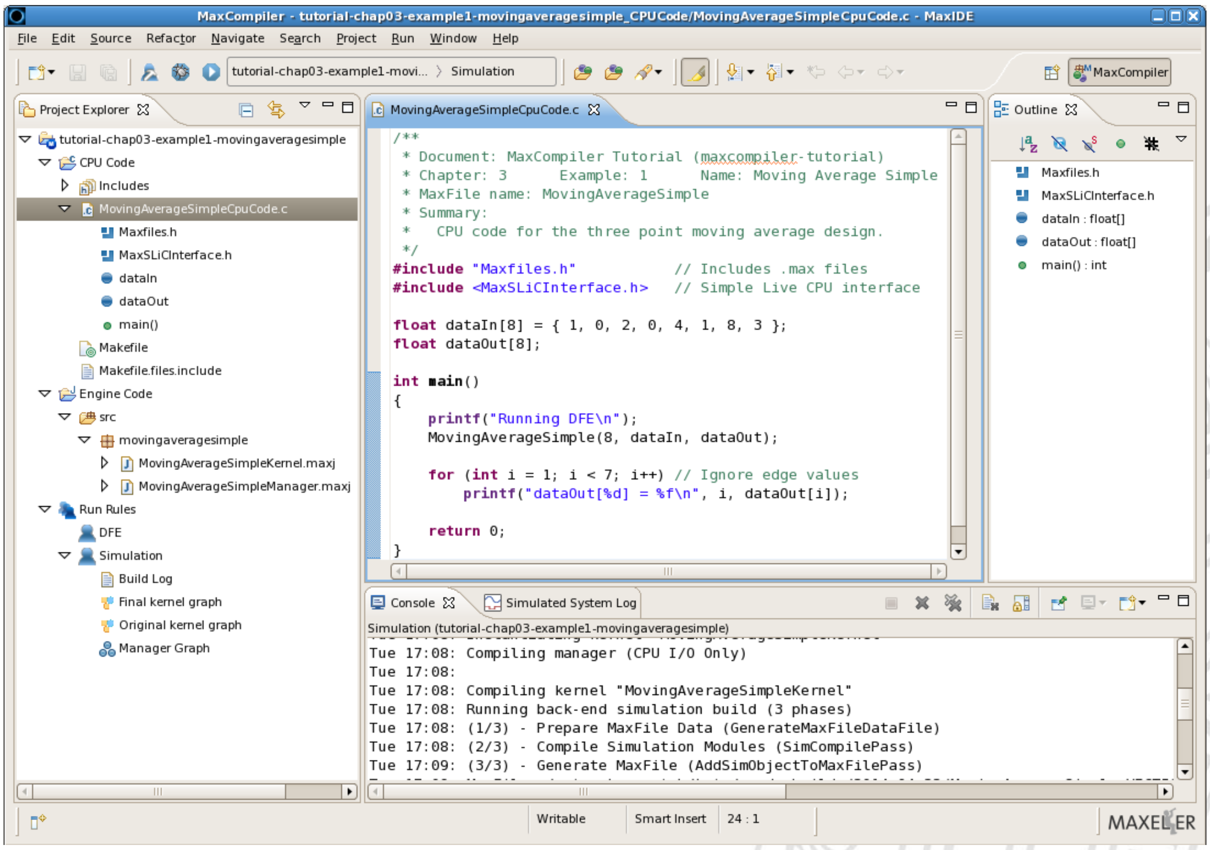
Maxeler is the platform where the whole project has been developed. It has multiple tools to transform from a high-level programming language like Java (MAXJ in this case) to a hardware language like VHDL and use external tools according to the FPGA to generate the bitstream to be used in this project.

MAXJ is the Maxeler platform language and it is used to describe a hardware structure through high-level programming language like Java. It is easy to use, and the model is predefined. It allows to do low level optimizations with low complexity compared to a hardware language.

However, the optimizations that can be done are not as deep as pure hardware language but the optimizations that can be done through Maxeler language are enough for the project.

3.4.1 Maxeler IDE

The Maxeler IDE is the Maxeler development environment, based on the Eclipse open source platform. In the process Kernel designs are going to be created, configuring Managers, building .max files for simulation and DFEs, and programming the CPU application software using the SLiC Interface.


MaxCompiler - tutorial-chap03-example1-movingaveragesimple_CPUCode/MovingAverageSimpleCpuCode.c - MaxIDE
File Edit Source Refactor Navigate Search Project Run Window Help
tutorial-chap03-example1-movi... Simulation
MaxCompiler
Project Explorer
tutorial-chap03-example1-movingaveragesimple
CPU Code
Includes
MovingAverageSimpleCpuCode.c
Maxfiles.h
MaxSLiCInterface.h
dataIn
dataOut
main()
Makefile
Makefile.files.include
Engine Code
src
movingaveragesimple
MovingAverageSimpleKernel.maxj
MovingAverageSimpleManager.maxj
Run Rules
DFE
Simulation
Build Log
Final kernel graph
Original kernel graph
Manager Graph
MovingAverageSimpleCpuCode.c
Outline
Maxfiles.h
MaxSLiCInterface.h
dataIn : float[]
dataOut : float[]
main() : int
Console
Simulated System Log
Simulation (tutorial-chap03-example1-movingaveragesimple)
Writable
Smart Insert
24 : 1
MAXELER

```
/**
 * Document: MaxCompiler Tutorial (maxcompiler-tutorial)
 * Chapter: 3      Example: 1      Name: Moving Average Simple
 * MaxFile name: MovingAverageSimple
 * Summary:
 *   CPU code for the three point moving average design.
 */
#include "Maxfiles.h"            // Includes .max files
#include <MaxSLiCInterface.h>    // Simple Live CPU interface

float dataIn[8] = { 1, 0, 2, 0, 4, 1, 8, 3 };
float dataOut[8];

int main()
{
    printf("Running DFE\n");
    MovingAverageSimple(8, dataIn, dataOut);

    for (int i = 1; i < 7; i++) // Ignore edge values
        printf("dataOut[%d] = %f\n", i, dataOut[i]);

    return 0;
}
```

```
Tue 17:08: Compiling manager (CPU I/O Only)
Tue 17:08:
Tue 17:08: Compiling kernel "MovingAverageSimpleKernel"
Tue 17:08: Running back-end simulation build (3 phases)
Tue 17:08: (1/3) - Prepare MaxFile Data (GenerateMaxFileDataFile)
Tue 17:08: (2/3) - Compile Simulation Modules (SimCompilePass)
Tue 17:09: (3/3) - Generate MaxFile (AddSimObjectToMaxFilePass)
```

Figure 3. MaxCompiler IDE with an imported project

The figure 4 shows how a project looks using Maxeler IDE environment. It has multiple tools based on Eclipse for debugging, compiling and running both simulation and DFE hardware implementation.

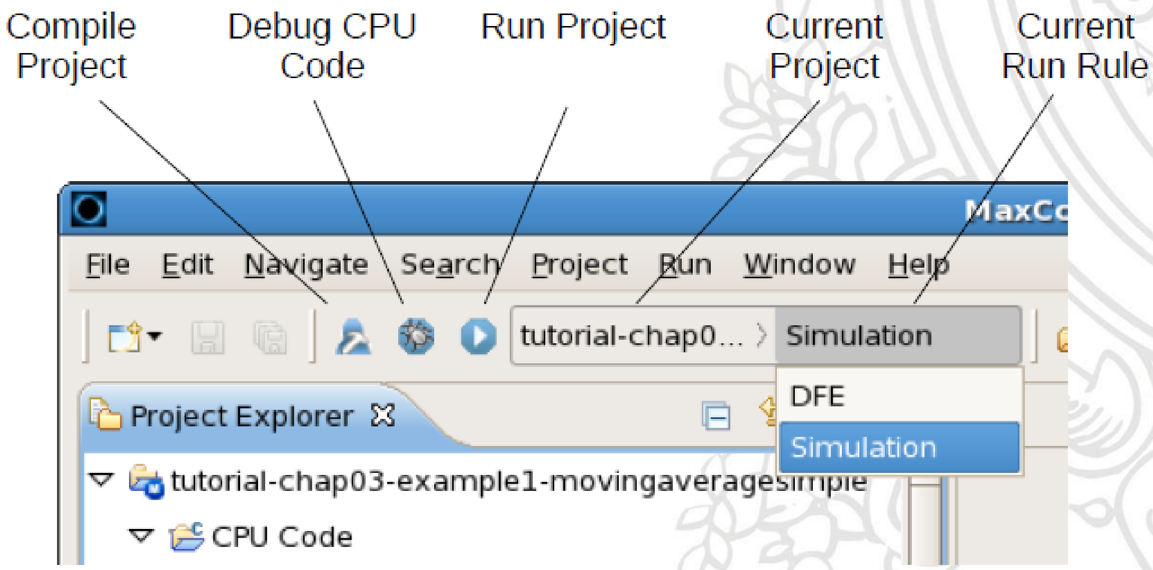

Compile Project
Debug CPU Code
Run Project
Current Project
Current Run Rule
MaxC
File Edit Navigate Search Project Run Window Help
tutorial-chap0... Simulation
DFE
Simulation
Project Explorer
tutorial-chap03-example1-movingaveragesimple
CPU Code

Figure 4. MaxIDE buttons for building and running a project

3.4.2 Dataflow model of computation

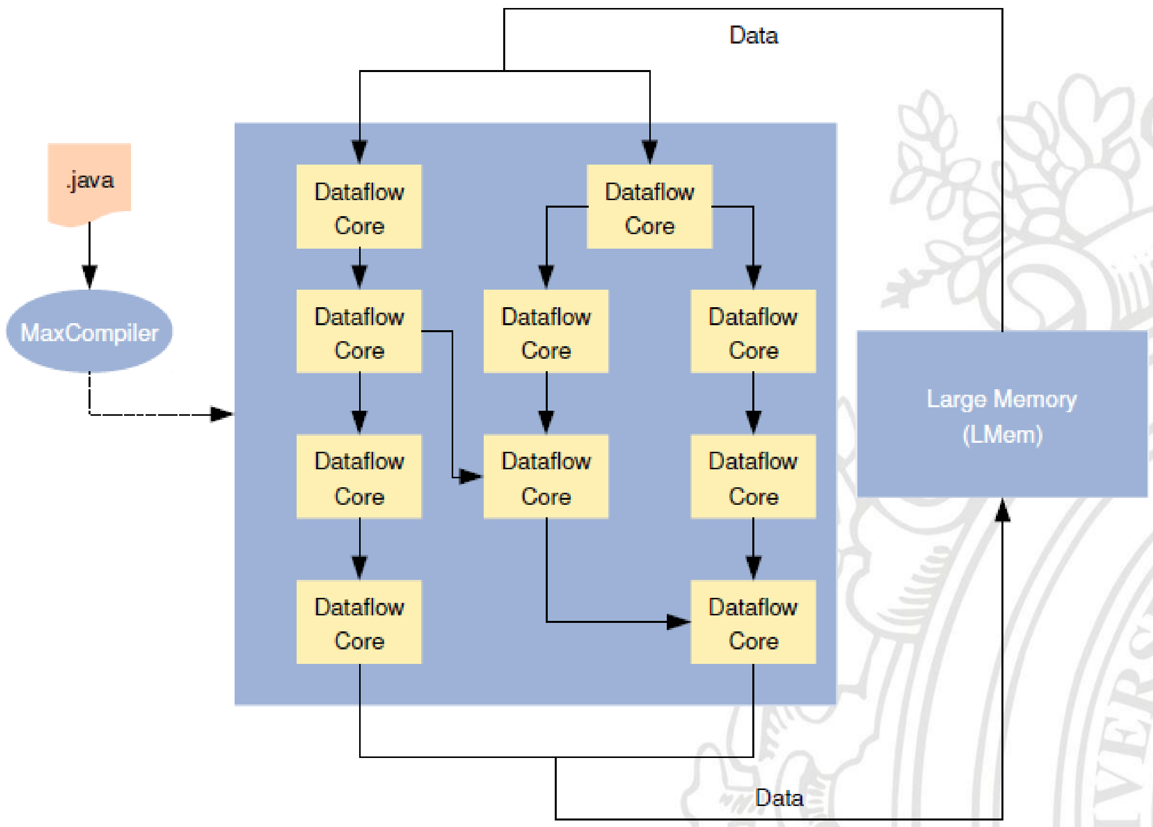


Figure 5. Dataflow program running

Dataflow is a model which consist of a set of processing elements in charge of controlling an input array, processing it, and storing it in memory. The advantage of this model is that the data is processed very fast and the performance can be high without using a complex code, the structure is predefined, and the user is focused only on creating an efficient kernel that processes all the information, most of the connections, pipeline and structure are defined automatically without losing any performance. The figure 5 shows how a dataflow program structure is defined.

3.4.3 Dataflow logic

The logic in the dataflow is defined by dataflow variables, these variables acts like signals and can interact between each other using adders, multipliers, multiplexers, counters and a lot of more logical circuits.

The logic is implemented in the kernel, each kernel has a selected input and output and it is processed by a set of logical circuits defined in MAXJ code. The DFE variables helps connecting the input into the logic circuits that processes the output, and it can interact as loops using counters or conditionals using multiplexers.

However, the java for cycles and conditionals can only have defined java constants and they are used just to make parallelism or define more restrictions in the structure that is going to be compiled and synthesised but not the variables that are processed in the kernel.

3.4.4 Variable types

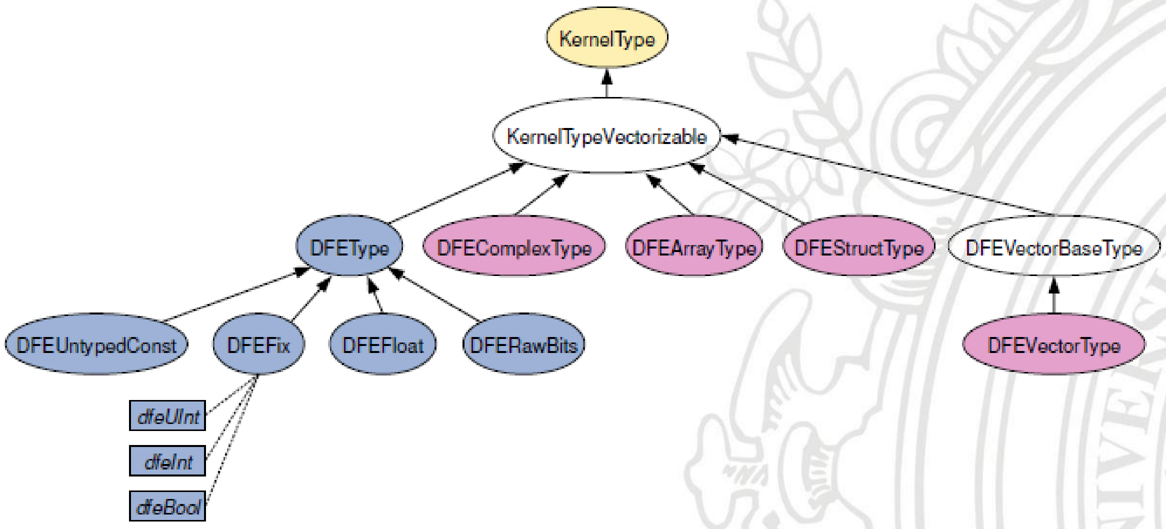


Figure 6. Class hierarchy for data types.

The figure 7 shows the hierarchy for the multiple variable types that can be used in a MAXJ. The structure shows from the basic types of signals to a Kernel Type of signal. Those variable types are used for all the data manipulated in the MAXJ files.

The variable types like integer, bool and float can be used but only as constants, constraints and fixed values for cycles and conditionals for hardware parallelization in the dataflow architecture but not for data management.

3.4.5 Common operators

Kernel Type	=	+[1]	-[1]	*[1]	/[1]	-[1]	<, <=, >, >=	<<, >>, >>>[1]	&, ^, |[1]	?:	~	[][5]	<==[6]	===, !==
DFEFix, DFEInt	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓[2]	✓	✓[3]	✓	✓
DFEUInt, dfeBool	✓	✓	✓	✓	✓		✓	✓	✓	✓[2]	✓	✓[3]	✓	✓
DFEFloat	✓	✓	✓	✓	✓	✓	✓					✓[3]	✓	✓
DFERawBits	✓							✓	✓	✓[2]	✓	✓[3]	✓	✓
DFEComplexType	✓	✓	✓	✓	✓	✓							✓	✓
DFEStructType	✓												✓	✓
DFEVectorType	✓											✓[4]	✓	✓

[1] Includes compound assignment operators (+=, *=, >>= etc.).
[2] Kernel Type must be 1-bit wide.
[3] Equivalent of `.slice(i)` to select a single bit.
[4] Returns indexed element from the `DFEVector` stream.
[5] Not allowed as left-hand-side in a statement (i.e cannot do `x[y]=z;`).
[6] <== is the *connect* operator, where `x<==y` is equivalent to `x.connect(y)`.
[7] Contained type of the multi-pipe stream must be 1-bit wide.

Table 1. Overloaded operators available by Kernel Type.

The table 2 shows all the overloaded operators for the dataflow variables. Most of them are used in the same way as Java language but there are some difference like the comparative operators for difference and equallity and the connect operator.

3.4.6 Input and output streams

One of the most important things about the DFE is the communication with the CPU, it is made using input and output streams which are defined in the manager, more specifically in the SLiC interface where all these variables are declared and defined both in CPU and FPGA.

The input streams work as windows which take a set of data and processes it in each tick, every tick the set of data is displaced n positions depending of the size of the window. The output streams work in the same way, every tick a set of data is stored in the output stream which is connected directly with the CPU.

The streams are declared in the manager where the parameters for these variables are set but they are also declared in the kernel where the size of the window can be predefined using a vector for declaring its size.

3.4.7 Kernel and manager

The kernel is the fragment of the code which processes the information generating an output from a determined input. In the kernel all the components are defined: the processing units, the variables and how the variables interact between each other. The manager is the fragment of de code in charge of determine how the input and output connections are going to be, it defines the parameters of the kernels, the size of the arrays, and how they are going to interact.

3.4.8 SLiC interface

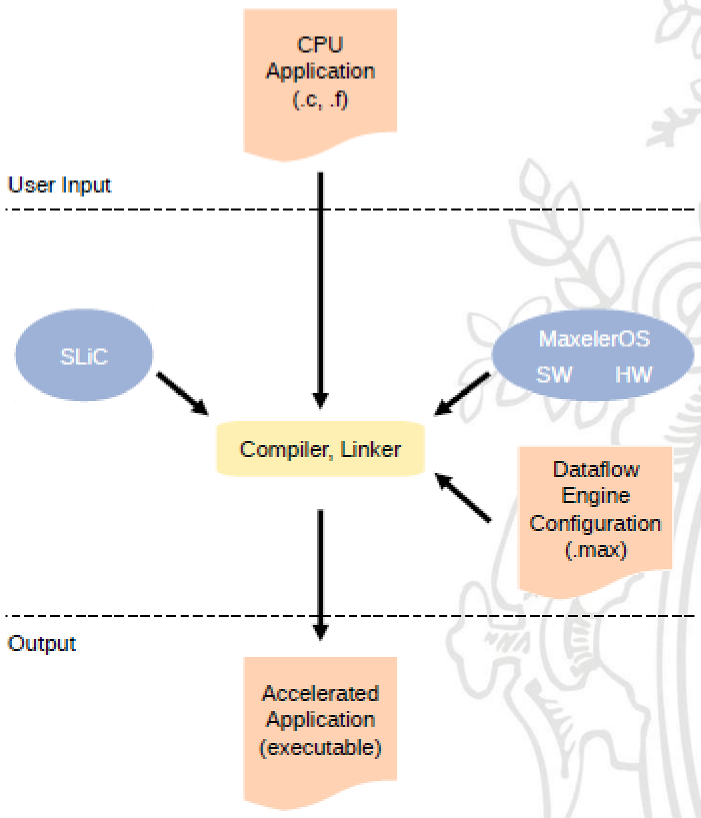


Figure 7 SLiC interface structure

SLiC interface is the interface that communicates the CPU application with the Dataflow architecture. It is a set of functions and variables that are declared both in the CPU header and the DFE manager.

3.4.9 CPU and DFE Code integration

One of the most important part of Maxeler technology is the implementation of an interface used for communicating the CPU and the DFE through functions, the SLiC interface is defined in the DFE and is based on the CPU input size.
The CPU Code has an autogenerated C class which contains all parameters defined on the DFE manager and one set of function per interface created on DFE Code, it can be used

4 Accelerator Development Methodology

4.1 CPU Code analysis

The main purpose of the project is to adapt an algorithm from CPU code to dataflow code. First, the CPU code must be adapted to receive and give the same data as the FPGA code. To make the hardware version of the algorithm the CPU code had to be extracted to adapt all the input data to one readable for the FPGA and test both the CPU run, and the FPGA run to establish how is the performance between those two.

The CPU segment code extracted to make the hardware version is the Inst2Bin code, the input data are the alignment data and the output is the binary instruction array. The inst2bin code consists in 2 basic structures: The prelude which contains the data related to the alignment type and the error which only appears in reads with errors and consists of the offset bits and additional bits for error type, flags and bases.

For the inst2bin code, all functions should be merged into one function to make correct measurements, and all the input data must be changed from structures of data to arrays. The CPU code also has been optimized to execute arrays of data and all the inst2bin sub functions has been merged into the main function to optimize and control all the process from that function.

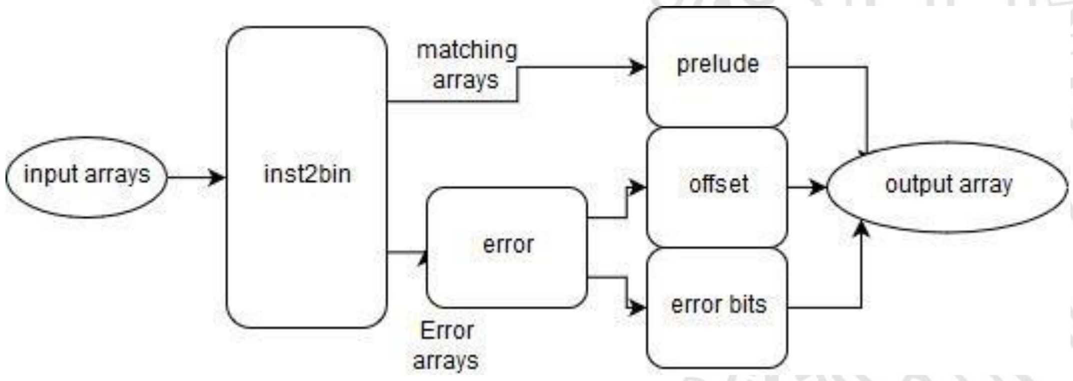


Figure 8. CPU code structure for inst2bin function

The figure 8 shows how inst2bin structure works, that code must be extracted with the corresponding sub functions and should be merged into one function called inst2bin with the purpose of managing only input and output arrays of data.

4.2 The prelude model

This model shows how the alignment data is going to change into codified instructions. In the prelude model the only codified data is the prelude that shows information about the alignment and the information and it appears in every read.

The hardware cost of this module is low, but the purpose is to introduce to the main part of the instruction coding

4.2.1 Detailed prelude structure

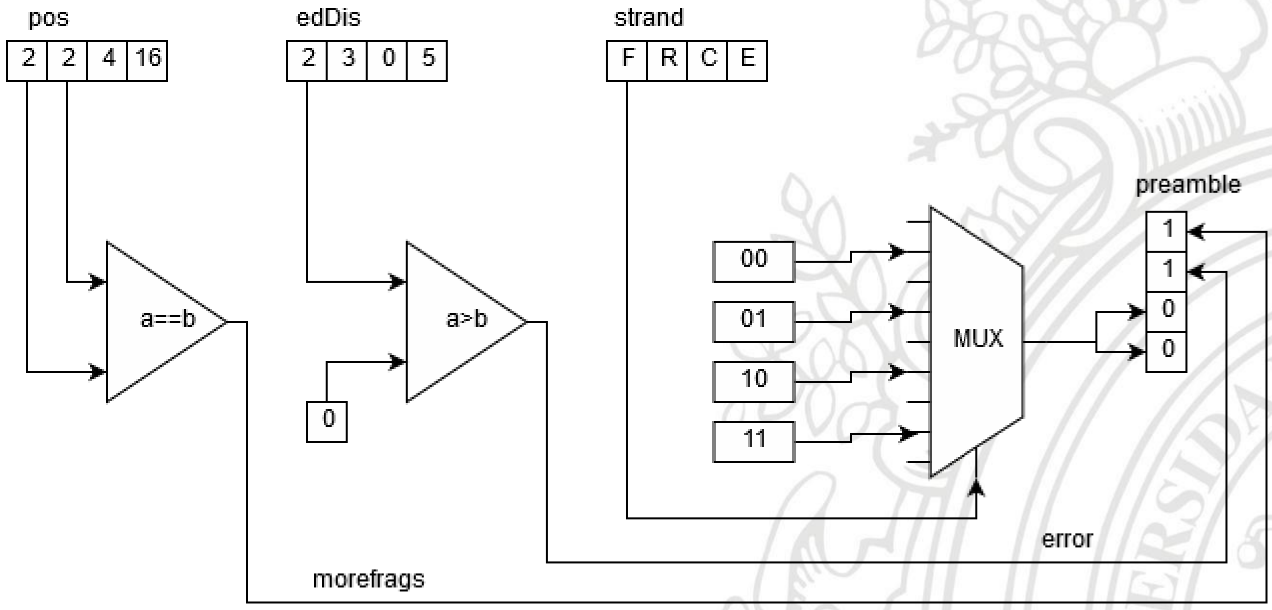


Figure 9. Preamble hardware structure

The figure 9 shows how one prelude is codified. The squares in figure 9 represents a stream of data, every clock cycle, the stream process one instruction and the index is moved one position forward.

The input data as shown is the pos (32 bit), which means the read position in the map. The edDis (16bit) which shows the amount of errors in the read and the strand (8bit) that shows the type of alignment. The output data is of course the preamble.

4.2.2 Prelude size optimization

This model was optimized using 2 prelude models to codify 1 byte that contains the data of 2 preludes. That allows to use less hardware and optimize the size of the error module.

Because every prelude doesn’t depend from the other this optimization has been done easily, the problem is to adjust the output type to receive the 2 preludes without any problem

4.2.3 Input and output management and kernels

The input and output are connected to the kernel which processes all the data, but before that, there is a part which made the pre-sets for the kernel, that is called the manager.

The manager oversees all the input and output data generation, it selects the size for every stream and select the parameters and the amount of ticks for the kernel.

4.2.4 Parallelism optimization

The parallelism in this module has been optimized by using vectors in the architecture. For this optimization, an input and output module were built to manage all the parallelism. This optimization has been made using vectors and cycles which mean the action is going to be repeated simultaneously, and it will create another similar unit in hardware. All the cycles are parameterized using the vector size parameter, which means the amount of times the same hardware is going to be replicated. It also means a reduction of the ticks necessary to process all the information, the ticks are divided by the amount of cores.

4.3 The prelude and error model

The prelude and error module were the first approach to codify an entire instruction. It has been developed using one processing element per read only for prelude processing and a parameterized number of processing elements for read only for error processing. In this case both the error and prelude processing are independent from each other.

4.3.1 Error kernel definition

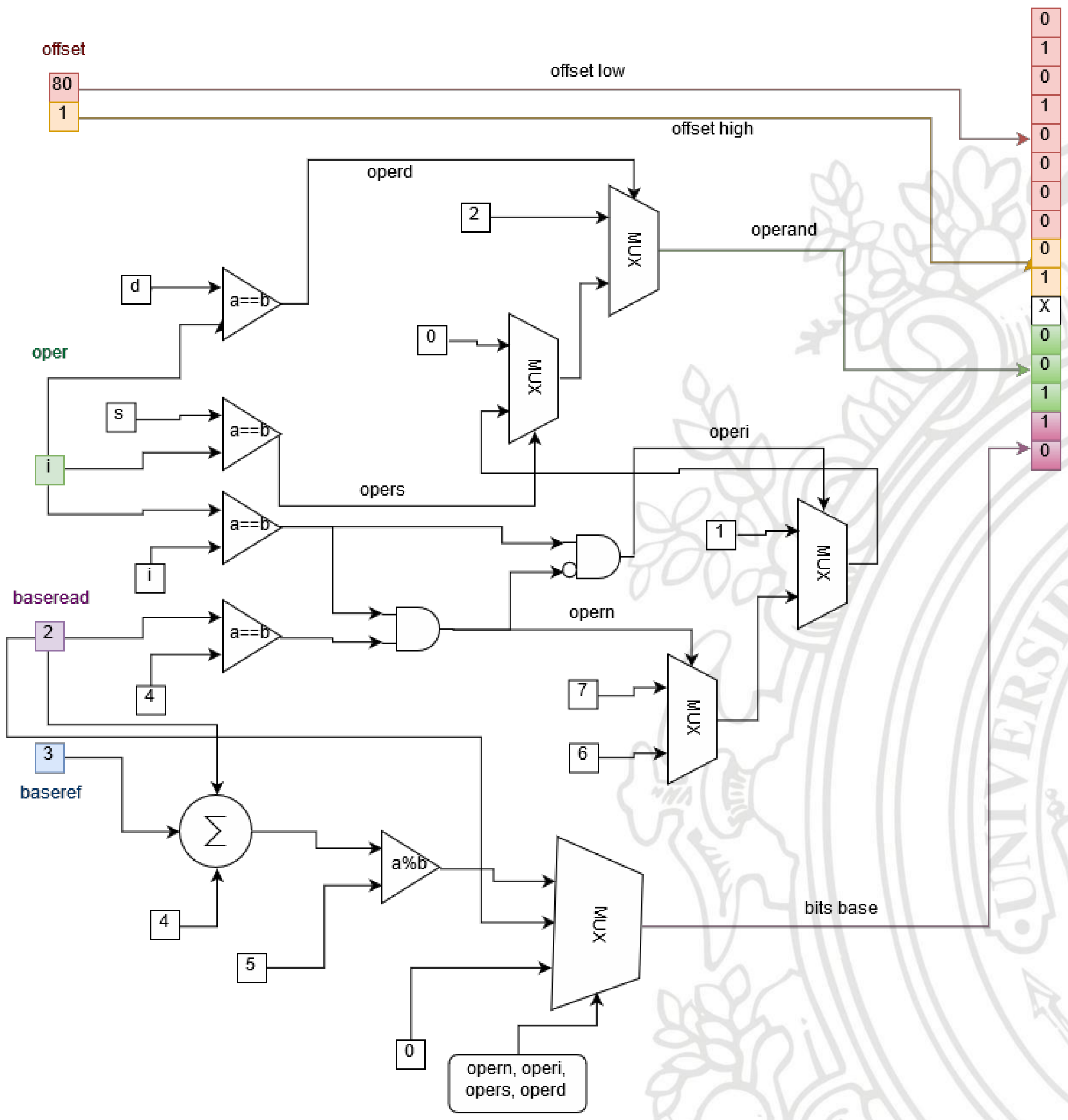


Figure 10. Error unit

The error unit is the basic unit to codify one error. It is not very useful to codify one read because normally all the errors depend from each other. To improve the situation, an error management module is going to be built, it manages all the errors that depends from each other.

4.3.2 Bits base and input multiplexing

In this unit as shown in the figure, the bits base is multiplexed, it is selected depending the type of instruction, for deleting and N insertion it is always 0, for insertion, it is equal to the read base and for substitution it is the distance between the base in the read, and the base in the reference.

4.3.3 Operation bits generator

The operation bits depend only on the input operand and the base read input. The input operation selects the operations bits but only in the case of single deletion, insertion, single substitution and N insertion. For N insertion the base read input is read if it is N, and the insertion operand is ignored.

4.3.4 Rearranging the offset

The offset is split in two parts, the high part with 2 bits that is in the first 2 bits of the second byte of the error, and the low part with 8 bits which is the first byte of the error. However, the 2 high bits needs to be added to the second error byte in the first 2 spaces with the more error flag, the error identification and the base bits.

4.3.5 Parallelism and singularities

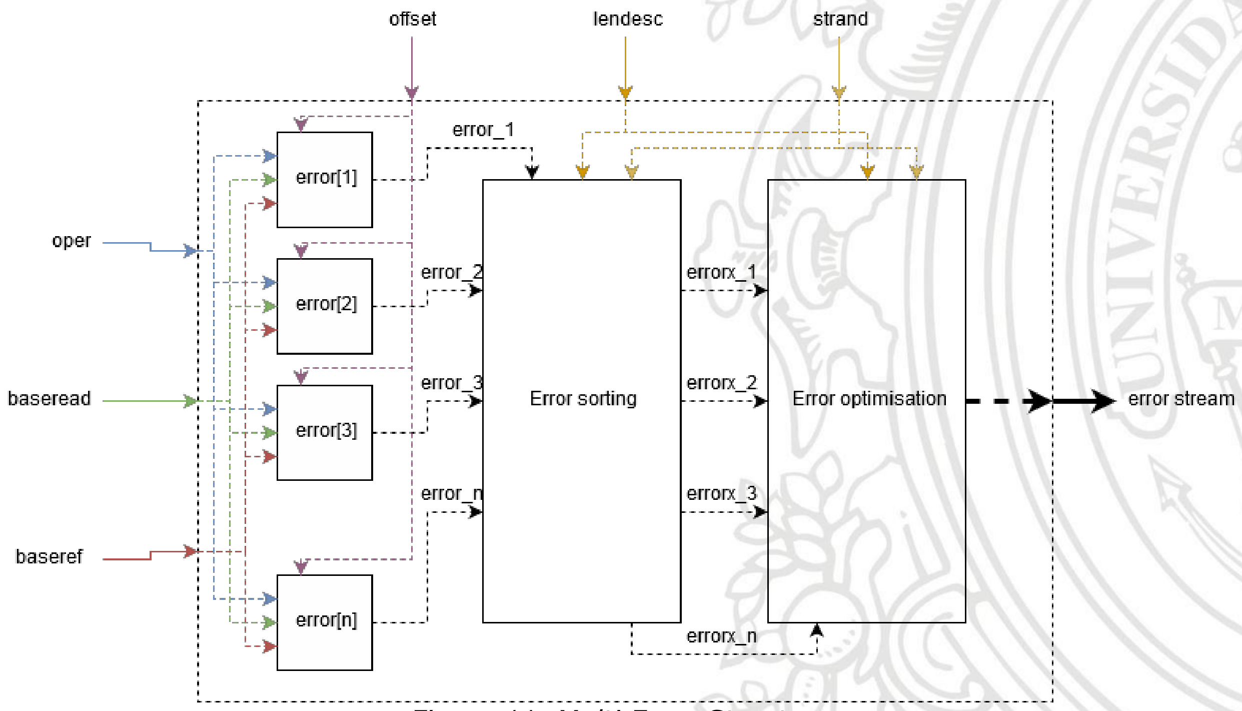


Figure 11. Multi Error Structure

This is the process to solve the errors in one read, there are multiple error modules that processes error individually, then, they are sorted according of the matching type, and finally they go to an error optimization module which add the more error bit for the according instruction and changes the single deletion or single substitution in the necessary cases.

However, the complexity of this module is high, and it requires high amount of hardware and lack of optimization because the input error size is different for every read and most of the time all the hardware is not used.

4.4 Inst2Bin code for fixed amount of errors

To solve the problem of unused modules and high complexity hardware, the inst2bin function had to be rewritten to perform a better optimization. The amount of errors of the input now are determined.

4.4.1 Error module modification

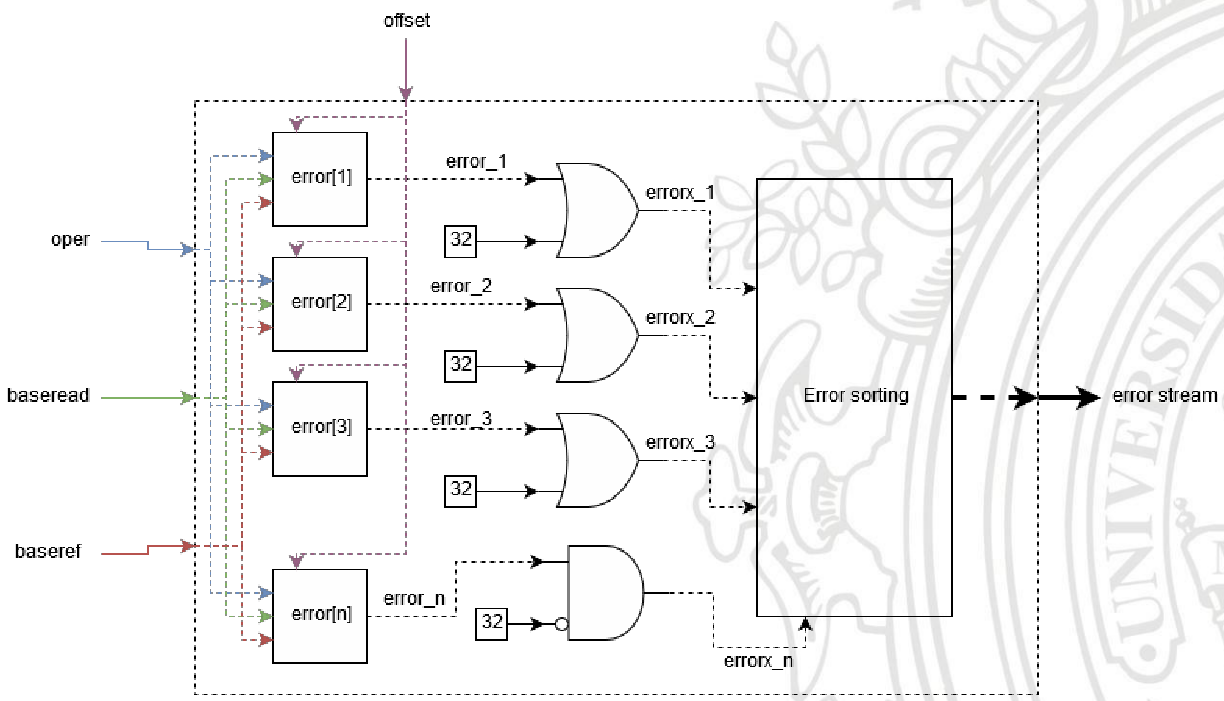


Figure 12. Fixed size error module

The diagram shows the modification to the fixed size structure. The purpose of the OR gate is to add the more error bit in each error and clear the more error bit in the last error. The purpose for the last module is to sort all the codified errors and pull them to the output stream.

4.4.2 Parallelism optimization

In this case, optimizing the parallelism was not a problem. The amount of hardware that were used was determined and it was replicated because each instruction doesn't depend on the next one and the output size is known before the modules have the output done. That means that all the processing time is divided by the number of cores that the structure has.

4.4.3 Inst2Bin for N amount of errors

Maxeler IDE allows to make parameters, in this case, the N parameter defined the amount of error that the model can do. For that all the vectors and hardware redundancy were based on a parameter instead of a constant. That allows to test every unit in different cases, allowing to measure the best design in terms of cores and giving more adaptability for the cores. The mutations are parameterized as well, there are 2 parameters in the manager which defines all the structures

4.4.4 Memory restructuration

In this case all the size of the blocks of memory are going to be restructured, also the mapping address will depend of 2 variables, the number of processing elements, and the amount of mutations per core.

The memory and the input and output streams are redefined according to 2 parameters, the amount of mutations and the amount of processing units can be different depending on the design.

4.4.5 Bug fixing

Several changes should be done to make all the project working. In this case the flags like more error should be fixed, that allows the decompressor where the codified reads end.

The Kernel structure has changed in 2 ways, it can detect uppercase and lowercase letters, and one additional module has been added to fix the problem when a mapping position is in the end of one vector of data.

4.5 Performance test input generation

For the design the dataset has been selected, in this case the reads are artificially generated into a txt format file which contains all the alignment data for the code. 12 files are generated for each test, with 4, 8, 12 and 16 mutations per read and 2 million reads for Bacteria, 5million reads for plants and 15 million reads for humans.

The input files are generated in a txt and the data is processed by a stream reader storing everything in arrays of data.

4.5.1 Input data for Inst2Bin CPU and DFE code

The input changed the structure for inst2bin CPU Code and DFE code. The CPU code includes more cases for multiple deletion and multiple substitution, anyway, the changes doesn't reflect several changes in the code structure, just allow to get all the cases of the read sequences.

The DFE code changed in the same way, the multiplexers have different amount of inputs allowing the possibility of processing different types of deletions and substitutions.

4.5.2 Dataset generation from humans, plants and bacteria genomes

The dataset has been generated by an artificial read generator, it takes a selected genome sequence and start generating reads and alignment data which is the one useful for the experiment. In this case the dataset generated is a fixed error dataset, for 4, 8, 12 and 16 mutations per reads.

The number of reads is going to be 2 million for bacteria, 5 million for plants and 15 million for humans. These data are stored in txt files with all the descriptions delimited by a text description.

4.6 DFE Implementation for performance tests

To perform the DFE for the selected dataset, the design has been made using a LUT with the following parameters

Input	Output
s	000
i	001
d	010
D	011
T	100
C	101
S	110
n	111

Table 2. LUT for instruction bits mapping

Each input character is an ascii character which is processed giving selected output set of bits according this table, the output has a width of 3 bits and those are the operand bits for each mutation in the codified instruction.

4.6.1 DFE parameters and limits

The design has limitation on hardware, for this, there is a maximum number of processing elements used in the design according to the device logic, in the design the limits are defined in this way:

4 Mutation design: 160max processing elements
8 Mutation design: 80max processing elements
12 Mutation design: 64max processing elements
16 Mutation design: 48max processing elements

The result code has been compiled according these parameters.

4.6.2 CPU Code time measurement for CPU and DFE Inst2Bin function

The time measuring has been implemented in the CPU code, the libraries to be used are clock C++11 library for times more than 500ms (storing and sorting times) and the timeval structure for times less 500ms (DFE and CPU code processing).

5 Implemented System

5.1 CPU Code structure

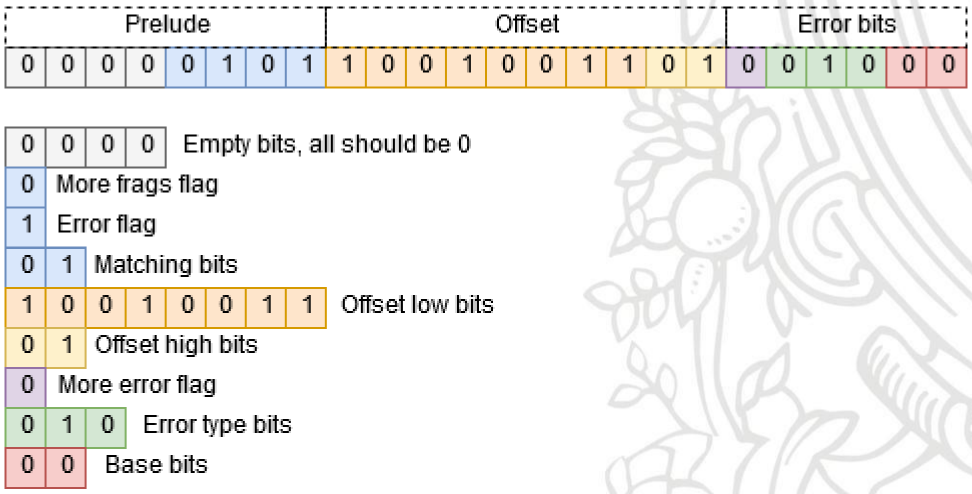


Figure 13. Inst2bin Code bits for UdeACompress algorithm

The figure shows how each instruction is codified. Each color means one function that does the work respectively. The prelude will appear only once in each codified read while the offset and error bits will appear for each error that the codified read have.

5.2 Prelude model structure

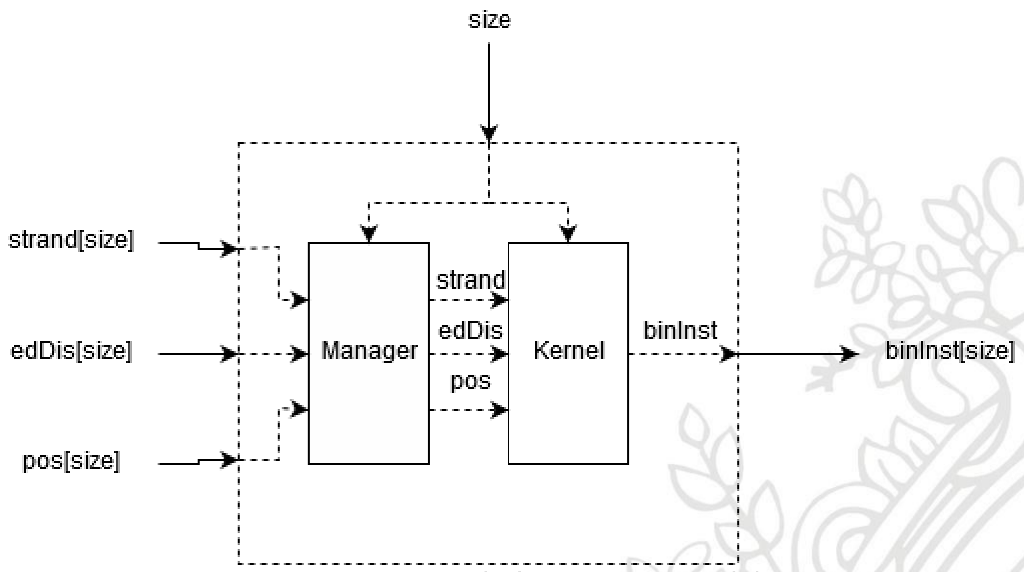


Figure 14. Prelude structure model

This is the basic prelude structure; the diagram shows how are the stream processed, the size of the array is chosen by the CPU parameter size and it decides the amount of ticks used for the kernel. The manager gets the data from the CPU, transform it into parameter and streams which are processed to the kernel and then, it pushes all the data to the output.

This core is only useful to codify reads without errors, the error management is not processed so it must be processed apart. This is the basic structure when a read comes without errors from the CPU.

5.3 Prelude and error model structure

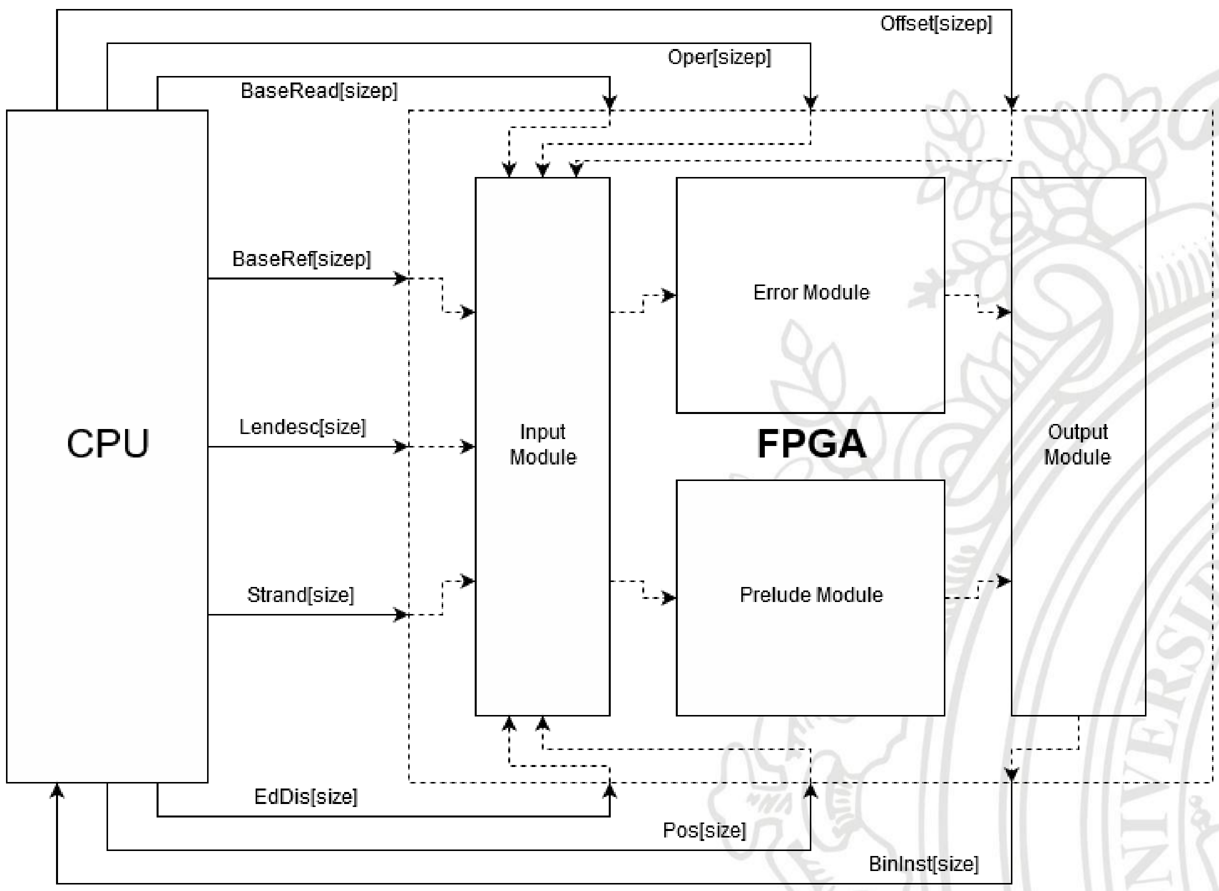


Figure 15. Prelude and error module with input and output kernels

This is the basic structure of the FPGA inst2bin multicore function. There is one module which get all the data and throw multiple data in multiple error and prelude modules. The data is processed by several error and prelude modules and organized by an output module which pull all the information to the CPU.

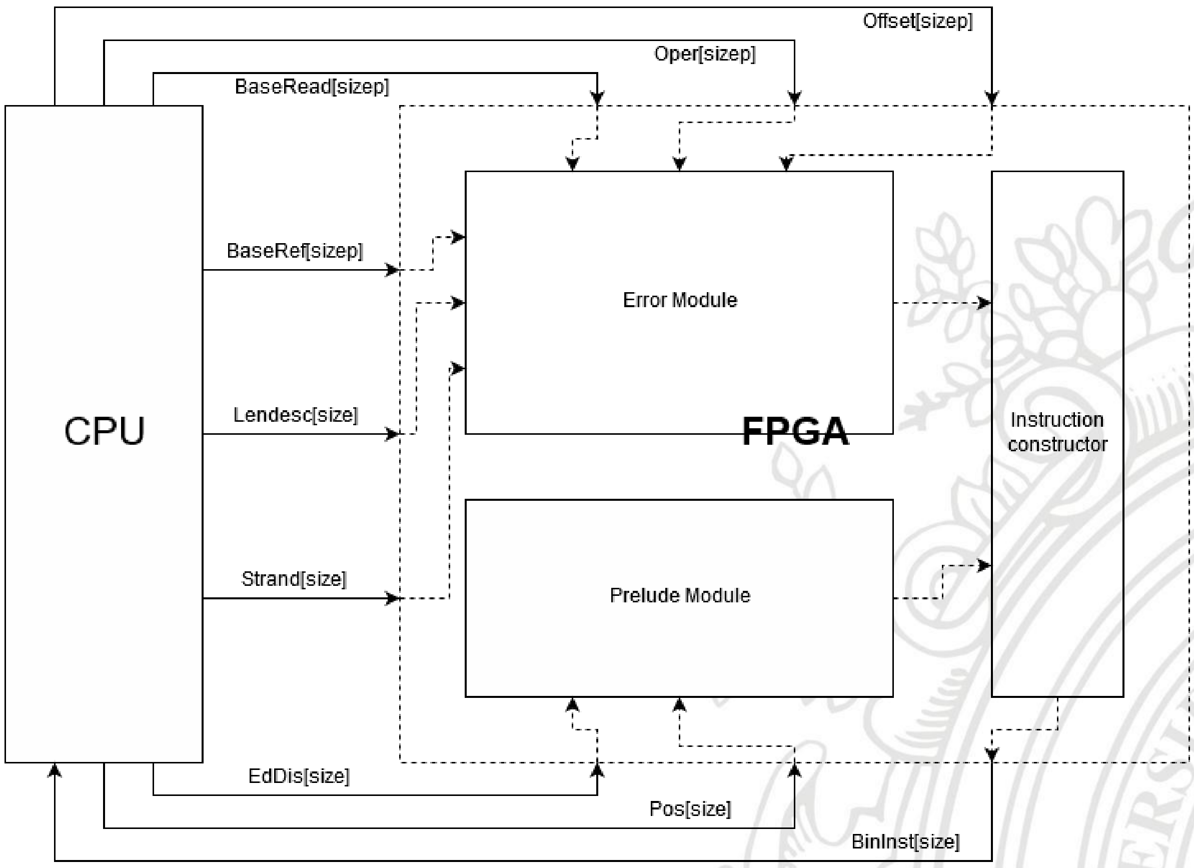


Figure 16. Prelude and error structure

This is the whole diagram; the input and output module are removed, and the streams are stored in vectors which works in parallel. The instruction constructor organizes the data and pull it to the output.

5.4 Inst2Bin for fixed error size structure

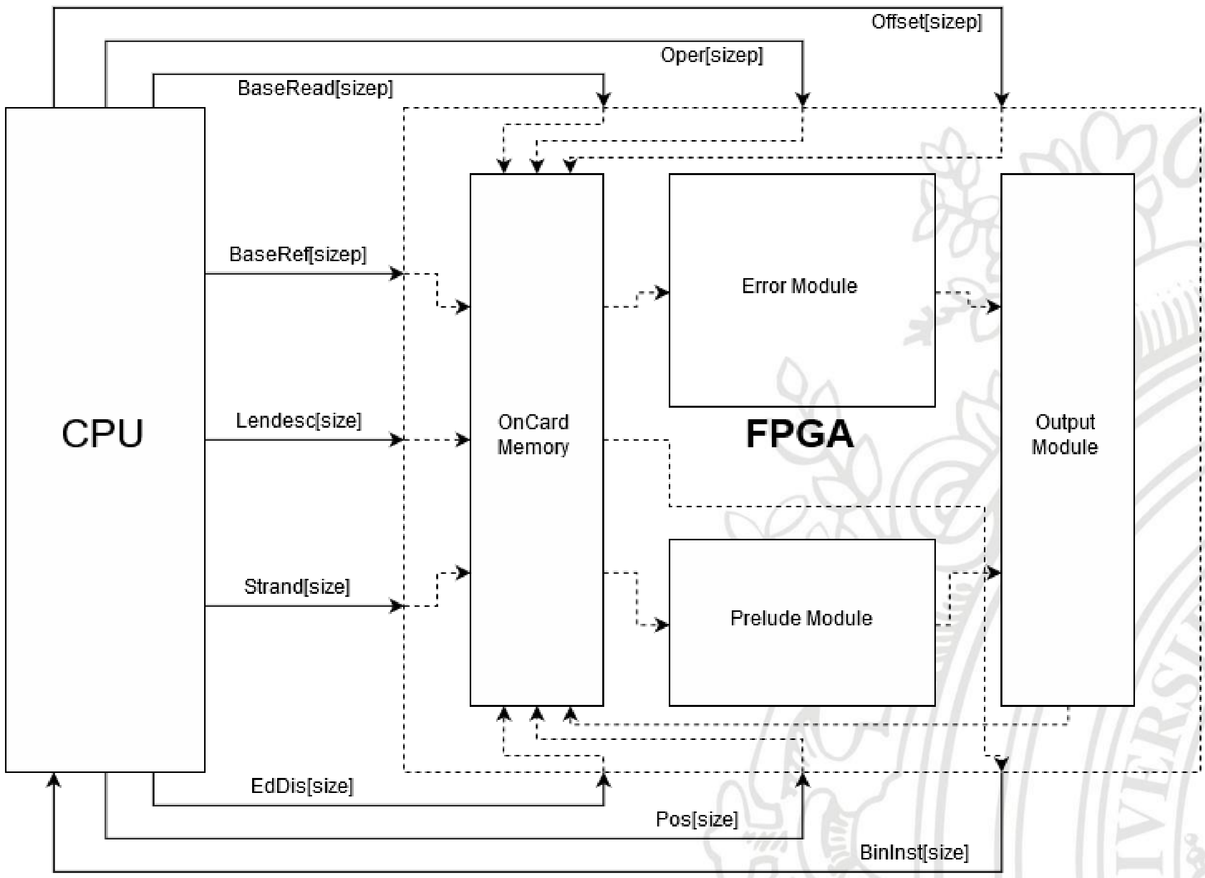


Figure 17. Inst2Bin for fixed error size structure

This is the whole compressor architecture, all the streams are stored in the memory and then, they are fetched by error and prelude modules. Then, the streams are processed by the error and prelude module and organized by the output module, and finally they are stored in the memory and fetched by the CPU.

After making several changes, the OnCardMemory module has been removed because it increases a lot the reading and storing time for reads which is a lot higher than the process time which is the one improved.

6 Results and analysis

The results presented in this chapter are obtained in some cases by direct measurements while in others they are estimations. Results were estimated when the Maxeler compilation tool, still not mature enough, failed to produce a bitstream to be downloaded to the FPGA. The dataset has been limited to 4 and 8 mutations per read and the number of processing elements for the result of the experiment can be 1 and 5 for 4 mutations and 1 for 8 mutations.

The code has been developed in Maxeler IDE version and the place and route has been made by Altera Quartus 13.1 using a DFE architecture from maxeler and it has been executed in a GALAVA FPGA with the following specs:

- Programmable logic fabric with 490,000 elements
- 500 programmable multipliers
- 5.6 MB of on-chip Fast Memory (FMEM)
- Single cycle FMEM access (e.g., avg 5ns@200MHz)
- 12 GB DDR3 DRAM Large Memory (LMEM)
- LMEM average latency of 250ns
- PCI-e link with 2GB/s (peak) bandwidth

The CPU code has been developed in a server with the following specs:

2x Intel(R) Xeon(R) CPU E5-2620 v2 @ 2.10GHz (12 cores, 24 threads for both)
40GB DDR3 RAM Memory

6.1 Compressing bacterium reads

The following diagram shows how the store and read time are affected for a bacterium genome which contains 2 million reads and 4 mutations per read.

PE	CPUtime(us)	DFEtime (us)	Speedup
1x	133943	180960	0,74
5x	135203	119891	1,13
10x	135203	112257,375	1,20
20x	135203	108440,563	1,25
40x	135203	106532,156	1,27
80x	135203	105577,953	1,28
160x	135203	105100,852	1,29

Table 3. Processing time in CPU and FPGA for bacterium

The yellow values are the values measured from the experiment and the red values are estimations based on the following model:

$$t_{DFE} = t_h + \frac{t_d}{n}$$

Equation 1. Elapsed time equation for FPGA

Where t_{DFE} means the total time elapsed compressing reads

t_h means the store and read time which cannot be improved
t_d means the time that can be improved
n means the number of processing elements (PE) for the experiment

In the experiment the obtained t_h and t_d values were 104623,75us and 76336,25us respectively and the red values were obtained from this

The speedup in the experiment was not great, a higher than 1 speedup has been obtained but it is not very different from CPU, 2 million of reads were not enough to test how fast can the FPGA run the process.

6.2 Compressing plant reads

In this case a set of 5million reads with 4 mutations is going to be analyzed in the next table.

PE	CPUtime(us)	DFEtime (us)	Speedup
1x	338977	282523	1,20
5x	334533	160154	2,09
10x	334533	144857,875	2,31
20x	334533	137209,8125	2,44
40x	334533	133385,7813	2,51
80x	334533	131473,7656	2,54
160x	334533	130517,7578	2,56

Table 4. Processing time in CPU and FPGA for plants.

In the experiment the obtained t_h and t_d values were 129561,75us and 152961,25us respectively. Based on those parameters, the red values were obtained according to the equation 1.

In this case the speedup is higher, that means the efficiency of the read processing increases according to the amount of reads to be processed. It is because the latency of the FPGA increases a lot the time required, in this case it has been performed in low time compared to the number of reads.

6.3 Compressing human reads

In this case a set of 15 million reads is going to be tested for a 4 mutations design. The results are shown in the next figure.

PE	CPUtime(us)	DFEtime (us)	Speedup
1x	997800	851978	1,17
5x	999594	462775	2,16
10x	999594	414124,625	2,41
20x	999594	389799,4375	2,56
40x	999594	377636,8438	2,65
80x	999594	371555,5469	2,69
160x	999594	368514,8984	2,71

Table 5. Processing time in CPU and FPGA for human

In the experiment the obtained t_h and t_d values were 365474,25us and 486503,75us respectively. Based on those parameters, the red values were obtained according to the equation 1.

The results in the table were very similar to the plants results, it means that the speedup can increase even higher using more reads to be processed.

6.4 Compressing very long artificial reads

A new experiment had to be done using a new set of reads to see how much speedup this set can reach, because of that a new artificial dataset of reads was built. In this case a new set of 200 million reads which is a considerably high amount for testing the performance in the FPGA. The results are shown in the next table.

PE	CPUtime(us)	DFEtime (us)	Speedup
1x	32526801	20352874	1,60
5x	32653485	4921084	6,64
10x	32653485	2992110,7	10,91
20x	32653485	2027623,85	16,10
40x	32653485	1545380,425	21,13
80x	32653485	1304258,713	25,04
160x	32653485	1183697,856	27,59

Table 6. Processing time in CPU and FPGA for custom genome

In the experiment the obtained t_h and t_d values were 1063137us and 19289737us respectively. Based on those parameters, the red values were obtained according to the equation 1.

In this experiment a higher performance has been obtained, the speedup obtained for real experiment was 6,64. It can be even higher according to the model and it could go even higher using a higher amount of hardware.

6.5 Number of PE comparison

The result dataset in the experiment were limited because of the problems using the FPGA, anyway some results can be shown to determine how is the time according to the number of mutations. The next table shows how is the processing time for a single processing element architecture with 8 mutations.

Genome	CPUtime(us)	DFEtime(us)	Speedup
Bacterium	216279	179385	1,21
Plant	523894	303079	1,73
Human	1945198	909419	2,14

Table 7. Results for a single core and 8 mutations

In this case the results are a lot better than using a single core architecture with 4 mutations, that is because the mutation units are more efficient in FPGA than CPU because of the amount of parallelization of an FPGA.

Using more cores can lead to more efficient results but the results are strictly limited because of the compilation issues.

6.6 Scalability

The next topic to evaluate is how the performance is improved according to the number of cores, the selected data is the best result obtained from the amount of reads analysis, in this case 200million reads from the custom genome.

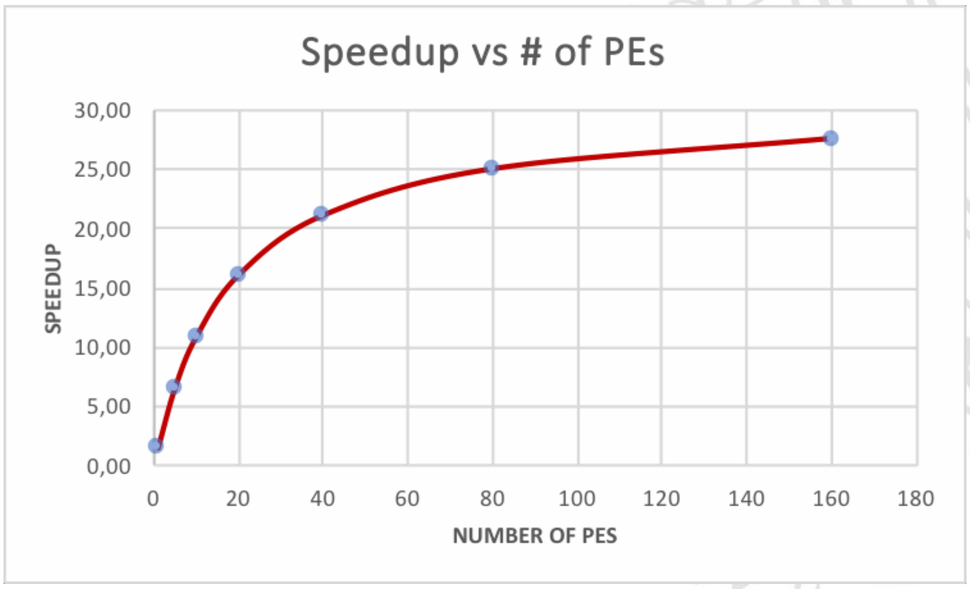


Figure 18. Speedup according to the number of PEs

The figure shows a clearly exponential behavior which shows that the speedup has a limit defined by the read/storing time that cannot be changed in the DFE architecture. The speedup increases a lot in the beginning, but it decreases a lot when many processing elements are used.

One important thing to measure is the time of CPU and DFE takes to execute a selected amount of instruction, the next figure shows how is the behavior according to the number of reads to be processed. The architecture selected is 4 mutations and 5 processing elements.

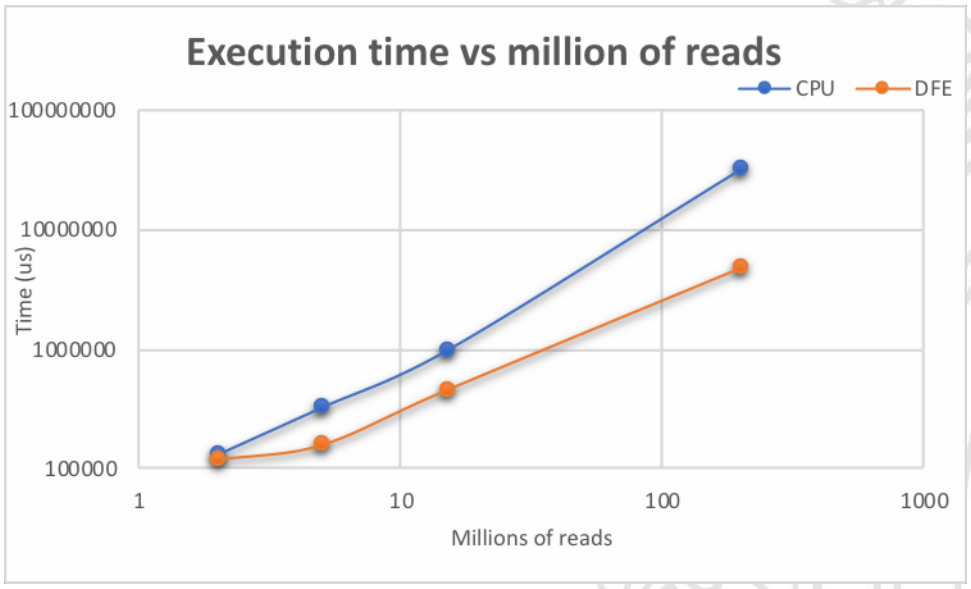


Figure 19. Execution time vs number of reads comparison

The figure shows the behaviour of increasing the amount of reads to be processed. In this case both graphics are linear but the DFE increases in lower rate than the CPU code, it means that the code is a lot more efficient using a higher amount of reads to be processed.

7 Conclusions

A compression algorithm for DNA reads has been studied in order to learn its parallelization possibilities. Then we have studied the novel computing paradigm of dataflow computing and used it to develop an accelerator.

A fixed mutation algorithm has been performed in the dataflow model with a parameterized architecture with high parallelism allowing to select the best design according to the parameters.

The algorithm has been tested and improved with setting a fixed amount of reads and removing the memory operations and replacing them with streams allowing to reduce the reading/storing time compared to the memory structure.

Several alignment files have been generated with different amount of mutations allowing to test humans, plants, and bacteria with 4, 8, 12 and 16 mutations to see how fast the process can be performed with the FPGA and the results shows that using more mutations and more amount of reads the performance is increased by the FPGA.

Maxeler is a very good tool to accelerate algorithms, especially if the amount of information to be processed is high and the operations made are very complex in terms of computation. The disadvantage of using it, is the amount of time required to execute a function, if the amount of data to process is low or the operations to perform are few, one cannot take advantage of this technology.

8 References

[1] ZACHARY D., Stephens, et al. Big Data: Astronomical or Genomical [en línea]. PLoSBiol13 3(7): e1002195, 2015 [Consult August 2018]. Available in: https://doi.org/10.1371/journal.pbio.1002195

[2] ORENGO, Christine; JONES, David; THORNTON, Janet. Bioinformatics: genes, proteins and computers [online] Reino Unido: BIOS Scientific Publishers Ltd, 2003. ISBN 0-203-44154-0 [Consult August 2018]. Available in: https://www.researchgate.net/file.PostFileLoader.html?id=58906dcab0366d768802e98a&assetKey=AS%3A456545336074241%401485860298614

[3] A. Guerra, J. Aedo. S. Isaza. Tackling the challenges of FASTQ referential compression. Submitted to Bioinformatics and Biology Insights. 2018.

[3] WALDENT, Sebastian; BUX, Marc; LESER, Ulf. Trends in Genome Compression. Alemania: 2013, [Consult August 2018]. Disponible en Internet: https://www.researchgate.net/publication/263474675_Trends_in_Genome_Compression

[4] WAIDYASOORIYA, Hasitha Muthumala; HARIYAMA, Masanori. Hardware-Acceleration of Short-Read Alignment. Vol. 27, NO. 5, MAY 2016, [Consult August 2018]. Available in: http://ieeexplore.ieee.org/document/7122348/

[5] What is lossy and lossless compression? Available in: https://whatis.techtarget.com/definition/lossless-and-lossy-compression.
Posted by: Margaret Rouse. June 2015. [Consult August 2018].

[6] Maxeler Technologies. Multiscale Dataflow Programming. Version 2016.1.1. Pacific Business Center. 2225 E. Bayshore Road. August 12, 2016. [Consult August 2018].

9 Annexes

9.1 Maxcompiler install process

9.1.1 Necessary files:

Maxcopiler-install-guide.pdf
MaxelerOS-tutorial.pdf

jdk-6u45-linux-x64-rpm.bin
apache-ant-1.7+
quartus 13.1
maxcompiler installer
maxelerOS installer
rpmforge-release-0.5.2-2.el5.rf.x86_64.rpm

9.1.2 Installing 3rd party software

Development tools:

```
$sudo yum groupinstall "Development Tools"
```

LABPACK and BLAS

```
$sudo yum install lapack blas
```

Java Development Kit

Download jdk-6u45-linux-x64-rpm.bin from oracle website

```
$sudo chmod +x jdk-6u45-linux-x64-rpm.bin
$./jdk-6u45-linux-x64-rpm.bin
```

Add JAVA_HOME environment variable

Apache Ant

Download from website version below 1.10, recommended 1.9

Follow the install guide

Add ANT_HOME environment variable

Skip XILINX ISE installation

9.1.3 Install Altera Quartus

Install all the quartus dependencies with the following command

```
$sudo yum install compat-libstdc++-33.i686 expat.i686 fontconfig.i686 freetype.i686 glibc.i686 gtk2.i686 libcanberra-gtk2.i686 gtk2-engines-2.18.4-5.el6.centos.i686 libpng.i686 libICE.i686 libSM.i686 libuuid.i686 ncurses-devel.i686 ncurses-libs.i686 PackageKit-gtk-module.i686 tcldevel.i686 tcl.i686 zlib.i686 libX11.i686 libXau.i686 libXdmcp.i686 libXext.i686 libXft-devel.i686 libXft.i686 libXrender.i686 libXt.i686 libXtst.i686
```

Note: Verify if the selected package is installed, some of them have different name

Execute the executable file without root

The following steps are just to select the packages, install all. In the devices, the Stratix V package is the only one chosen.

After installed it will ask for the license file, first, add the license environment variable to the PATH and add it also as LM_LICENSE_FILE variable.

You should change the network interface name to read correctly the license. The commands for temporary change the interface names are the following below

```
sudo /sbin/ip link set eno2 down
sudo /sbin/ip link set eno2 name eth2
sudo /sbin/ip link set eth2 down
```

The following step is to validate the license file, the checkbox "Use environment variable" must be checked.

Restart the Quartus interface until the validation message doesn't appear anymore

Finally, open the file /etc/selinux/config and change the line

```
SELINUX=enforcing
```
to
```
SELINUX=permissive
```

Then restart the machine and the Altera Quartus program should be ready to be used

9.1.4 Maxcompiler installation

Decompress the maxcompiler installation file using the following command
```
$tar xzvf maxcompiler-2016.1.1-full-installer.tar.gz
```

Go to the decompressed folder and install using the following command.
```
$sudo ./install --edition c /opt/maxcompiler
```

It should say only one missing package, the XILINX ISE. Execute the command again with the flag ignore-missing
```
$sudo ./install --edition c /opt/maxcompiler --ignore-missing
```

Then select yes

NOTE: The selected command will delete all the files in the folder /opt/maxcompiler

Copy the license file to the folder /opt/maxcompiler/license

Add environment variables MAXCOMPILERDIR, MAXCOMPILERBIN and MAXLICENSEDIR to the PATH

The license is obtained sending a mail to Maxeler

9.1.5 Installing Maxeler OS

Dependencies:

Install rpmforge and dkms using the commands in the download folder
```
$sudo rpm -i rpmforge-release-0.5.2-2.el5.rf.x86_64.rpm
$sudo yum install dkms
```

Install MaxelerOS

Execute
```
$sudo rpm --install maxeleros-2016.1.1-1.el6.x86 64.rpm pre-install maxeleros-
2016.1.1 1 /post-install maxeleros-2016.1.1 1
```

The MaxelerOS Daemon will be installed on /etc/init.d/maxeleros
The utils are installed in /opt/maxeleros/utils

Test the maxelerOS installation using /etc/init.d/maxeleros restart
The status must show OK in all lines

Finally, add MaxelerOS environmental variables.

The bash.rc file should look like this:

```
export ANT_HOME=/usr/lib/apache-ant-1.10.1
export JAVA_HOME=/usr/lib/jvm/java-1.8.0
export PATH=${PATH}:${ANT_HOME}/bin

export LMGRD=/home/rdavid.caro/flex
export PATH=$PATH:$LMGRD

export QUARTUS=/home/rdavid.caro/altera/13.1/quartus/bin
export QUARTUSLINUX=/home/rdavid.caro/altera/13.1/quartus/linux
export PATH=$PATH:$QUARTUS:$QUARTUSLINUX

export MAXCOMPILERDIR=/opt/maxcompiler
export MAXELEROSDIR=/opt/maxeleros
export MAXCOMPILERBIN=/opt/maxcompiler/bin
export PATH=$MAXCOMPILERBIN:$PATH
export PATH=$MAXCOMPILERDIR:$PATH
export PATH=$MAXELEROSDIR:$PATH

export LM_LICENSE_FILE=/home/rdavid.caro/altera/13.1/Altera_1-
FV0CHV_License.dat
export PATH=$LM_LICENSE_FILE:$ALTERAD_LICENSE_FILE:$PATH
```

Try an example copying the maxcompiler tutorial folder to the home directory, then go to an example in the folder examples, go to the CPUCode in the example folder and execute the command:

```
$make RUNRULE=DFE run
```

It takes about 2 hours to execute the script.