HAUTE ÉCOLE
D'INGÉNIERIE ET DE GESTION
DU CANTON DE VAUD
www.heig-vd.ch

BACHELOR THESIS

Genomic Data Processing

Hardware Implementation of an Indexed Sequence Mapper

Autor:
Arthur PASSUELLO

Supervisor:
Prof. Yann THOMA
HEIG-VD

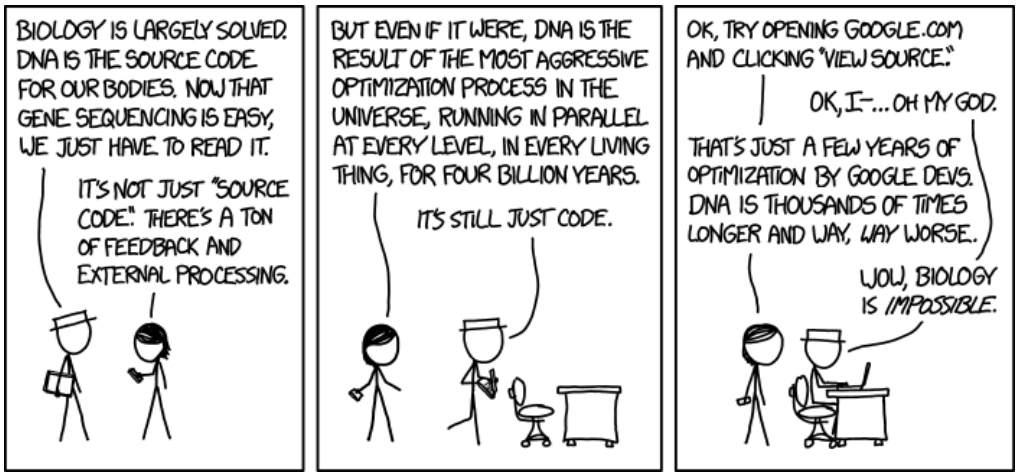
Expert:
????
???????

Presented at the 2018 Autumn session under the supervision of Prof. Yann Thoma

at the

Faculté d'Informatique Embarquée
HEIG-VD

June 12, 2018



DNA

" Relevant xkcd.[1]"

[1] source : *https://xkcd.com/1605/*

Abstract

Genomic Data Processing

Arthur PASSUELLO

Résumé

Traitement de Données Génomiques

Arthur PASSUELLO

BLABLABLA

Glossary

Abréviation	Terme	Définition
HMC	Hybrid Memory Cube	Mémoire constituée de couches empilées de mémoire de type DRAM.
BWT	Burrows-Wheeler Transform	Prétraitement des données voir ici
FM-Index	Full-text Minute-size Index	Principe de compression sans perte, Voir ici
REDS	Reconfigurable Embedded Digital System	Institut de prestige de l'HEIG
PEHAGA	Power-Efficient Hardware Acceleration of Genomic Data	???

Contents

Chapter 1

Introduction

DNA sequencing is considered to be a major breakthrough in both medical and biological research. Despite being at first a very lengthy process, the development of new parallel *high-throughput sequencing* (HTS) [1] technologies allowed much faster and far less costly genome sequencing [2]. Unfortunately, the same breakthrough has not been made with the processing of those data. As a matter of fact, the rate at which genomic data is generated exceeds substantially the rate at which it can be processed and this issue might become the next major *Big Data* issue, surpassing current massive data originator[3] .

Current DNA sequencing software, such as *Bowtie*[4], use optimal data structures and highly parallelized algorithms in order to reach a maximum processing speed but the whole sequencing still requires several days of computation. Indeed, even though software sequencing tools run on ultra-fast CPUs and pass through highly optimizing compilers, they suffer from various forms of processing overhead, severely reducing performances in the form late

1.1 Genomics & Genomic Data

Genomics is an area within *Genetics* that focuses on sequencing and analyzing organism's genome (Dr Ananya Mandal, 2014). The whole *genome* is contained in every living cell, is encoded in *DNA* and for us, distributed over 22 chromosomes, each of which in two copies, and two sex chromosomes. Every DNA sequence consists of combinations of four different nucleobases A, C, G and T (respectively *adenine*, *cytosine*, *guanine* and *thymine*) in a double-helix form. This shape consists of two complementary strands i.e. both strands contain the same information and one is the *reverse* complement of the other (A always pairs with T, and

[1] CITE ILLUMINA
[2] CITATION
[3] http://journals.plos.org/plosbiology/article?id=10.1371/journal.pbio.1002195
[4] citer

G with C). The sequence length varies a lot between organisms, from around a million of pairs for bacteria, to hundred of billions for some fish and plants. Typical length for humans is around 3 billions of pairs.

The size of those genomes therefore requires a large amount of processing to acquire, of space to store and most importantly, a huge amount of both to extract useful information from it. The goal of genomic data analysis is to determine the functions of specific genes, their origins and their evolution among many other informations. An efficient way to do so is to determine for each species a reference genome which can then be used to as a map. This approach allows researcher to identify measured samples as belonging to those references and eventually enables the identification of specific sequences to precise species and biological functions.

1.2 Problematic Introduction

Genomic data is thus foreseen to become a major *Big Data* issue, it's amount increasing at a faster rate than current technologies improvements in terms of data processing. Despite multiple highly optimized tools designed to increase processing throughput, the current pipeline, detailed below, cannot seem to catch up. In the next figure 1.1

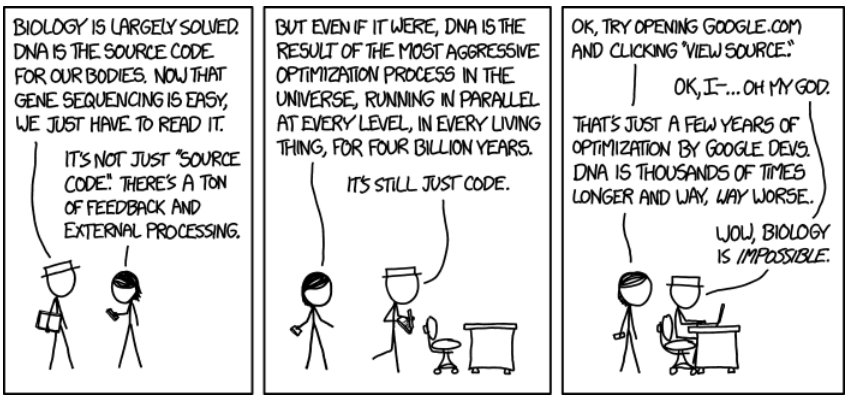


Figure 1.1: Work Tasks and Genomic Data Processing Pipeline

Genomic data is thus a *Big Data* issue, its amount is increasing way faster than Moore's Law leaving processing systems way behind, despite multiple highly efficient tools developed to increase the speed of such operations. Those tools are mostly software and take full advantage of the specific nature of such data : a very long *text* containing an awful lot of redundancy. Redundancy means compression is possible and textual format allows easier manipulation (than a video for example). Hence, those software tools take full advantage

of the specific nature of genomic data by using structures allowing manipulation and compression of those input, this is the subject of the next section.

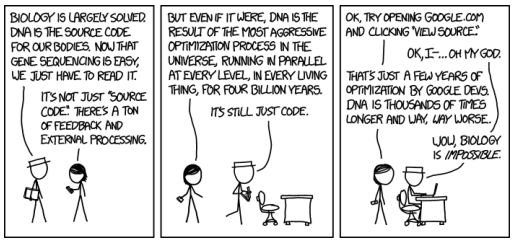
1.3 Research Question

[Poser ici ma vraie thèse, à savoir que je vais vouloir vérifier si 1) c'est possible d'implémenter tout ça sur FPGA 2)si c'est bel et bien plus rapide]

1.4 Method Introduction

[Présenter grosso modo ce que je vais faire, présenter aussi le plan du document]

Chapter 2



Software Implementation of the FM-Index

Although the main scope of this project revolves around the hardware implementation of the string matching algorithm, several software versions of the *Burrows-Wheeler Transform* and the *FM-Index* have been implemented. The first is needed to construct all the required data structures for string matching (i.e. the BWT, the suffix array and FM-Index samples) while the second will be used as a reference to compare the results with those obtain via the hardware implementation.

2.1 Burrows-Wheeler Transform and FM-Index

Introduced by *Michael Burrows* and *David Wheeler* in 1994, the *Burrows-Wheeler Transform* (BWT) rearranges characters in a *string* into sequences of similar characters. Although it does not reduce the size of the input data, the rearrangement gotten as output combined to the easy reversibility of the process and greatly improved compression, among other interesting properties, makes it particularly useful in genomic data processing and sequence mapping.

2.1.1 Burrows-Wheeler Transform Steps

The transform can be divided in 4 main steps[1] :

1. Append a special character (usually represented as a $), lexicographically smaller than any others, at the end of the input string T.

2. Construct a conceptual square matrix M_T, whose row are cyclic shifts of T$.

[1] National Tsing Hua University, 2013.

3. Sort M_T in lexicographical order

4. Get the transformed text output T^{bwt} by selecting the last column of M_T

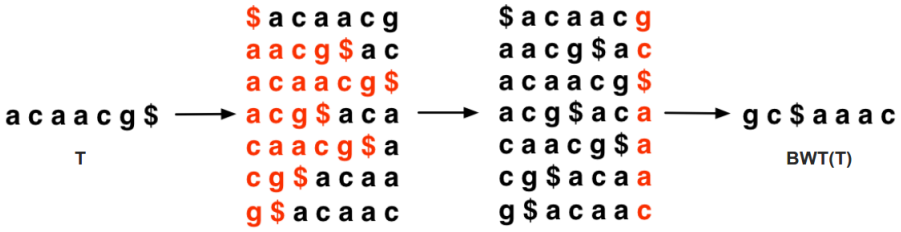


Figure 2.1: Illustration of the *BWT* applied to the string *acaacg*

The figure above[2] illustrates the process of the *BWT*, it will be shown next what is used in the mapping process, but first let us note the following properties :

1. the process is revertible

2. all columns are permutations of the input T

3. the last columns contains mostly all similar character consecutively

4. the first column contains all characters in lexicographical order, so the first row will always start with the inserted $ symbol (i. e. be the first rotation applied to T)

Those properties will show up to be useful in the indexing process, which is the subject of the next paragraph.

2.1.2 FM-Index

An *FM-Index* is a lossless text compression based on the *BWT*[3], it was created by *Paolo Ferragina* and *Giovanni Manzini* whom refer to it as an *"opportunistic data structure"*, it allows compression and fast sub-string queries on the compressed data, which highly improves sequence matching performances, as explained below.

Creating an FM-Index In order to create the *FM-Index* of an input full-text *string* T, one must create its *BWT* transform T^{bwt} as explained in the previous section. If you recall `Figure 1.1`, the left matrix illustrate in red a structure, called the *suffix array*, derived from the input. This structure, implicitly defined by T^{bwt}, can be accessed using the *FM-Index*

[2] David K. Gifford, 2014.
[3] Wikipedia, The Free Encyclopedia, 2017.

data structure. (SOURCE = USING THE MULTI STRING BURROW WHEELER paper) This index takes full advantage of the properties described in the previous section to construct a mapping algorithm between the T^{bwt} and the suffix array of an input T, stating that the j^{th} occurrence of a symbol c in T^{bwt} corresponds to the j^{th} suffix beginning with that symbol in the suffix array, this is called the *Last-to-First Mapping* or LF-Mapping.(CITER SOURCE WIKI P EX OU PAPIER).
So, given a T^{bwt} of N symbols and an alphabet Σ of σ elements, we define the FM-Index F as a two-dimensional matrix of value with $(N + 1)$ rows and σ column. For a given index i (row) and a given symbol c (column), $F[i][c]$ contains the number of occurrences of c before the index i in T^{BWT}. From this definition, we can construct the whole index for T^{bwt}, initializing $F[0]$ at 0 and $\forall\ i$ from 0 to N, copy $F[i]$ into $F[i + 1]$, incrementing $F[i + 1][c]$ only if $T^{BWT}[i] = c$. The last entry $F[N + 1]$ represents the total number of occurrences of all symbols in T^{bwt}. Using this index, we can statically compute the Occ array of length σ, which stores the index of the first suffix starting with each symbol in the suffix array.

LF-Mapping, Walk-Left and String Matching Algorithms We now have all the elements required to describe the LF-Mapping function for i the current index and $c \in \Sigma$ the queried symbol in T^{bwt} :

$$LF(i, c) = Occ[c] + F[i][c]$$

The return value is the index of the corresponding character in the suffix array.
Using this function, we can define the *Walk-Left* algorithm to invert the BWT and retrieve to original string T.

Algorithm 1: Walk-Left - Inverting the BWT

Input: BWT T^{bwt}, FM-Index F, First occurrences Occ, index i

Output: The original input strinp T

```
i = 0;
t = "";
while T^bwt[i] ≠ "$" do
    t = T^BWT[i] + t;
    i = LF(i, T^BWT[i]);
end
```

Using the same function, we now define an algorithm that, given a BWT T^{bwt} and a query string q, will return the index range in the implicit suffix array containing q :

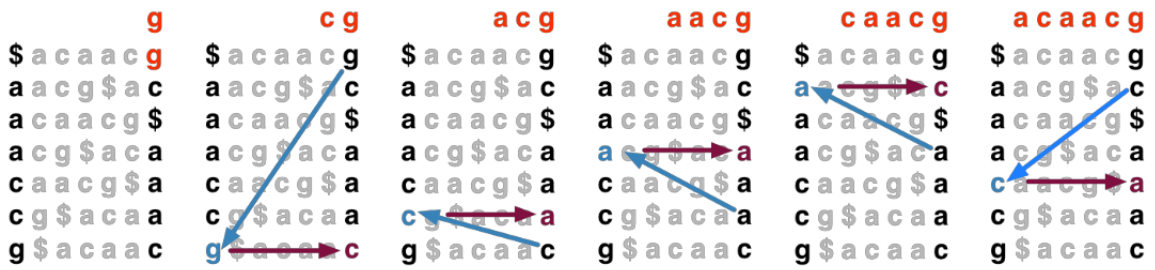


Figure 2.2: Illustration of the Walk-Left Algorithm inverting the T^{bwt} to retrieve original text T

Algorithm 2: Exact Suffix Matching in Suffix Array

Input: BWT T^{bwt}, FM-Index F, First occurrences Occ, query string q

Output: Index Range top, bot for q in the suffix array corresponding to T^{bwt}

```
1 top = 0;
2 bot = len(T^BWT);
3 for qc in reverse(q) do
4 |   top = LF(top, qc);
5 |   bot = LF(bot, qc);
6 end
```

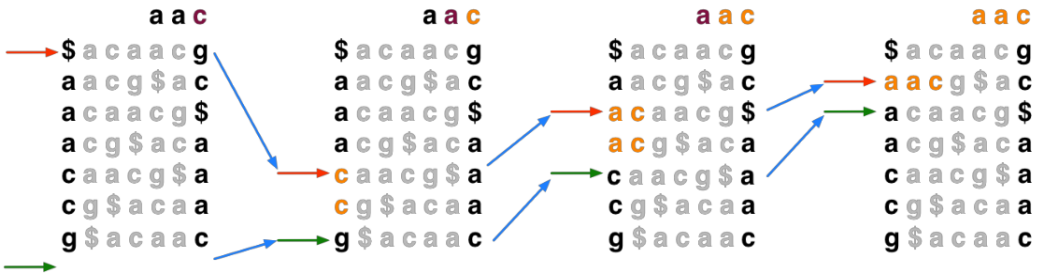


Figure 2.3: Illustration of the matching process of the string q = "aac" in the suffix array derived from T^{bwt}

If the return range $[top, bot]$ is empty, then the queried string q does not appear in the reference text.

Putting it all Together Now that all the data structures and computational tools have been introduced, we can define the process of exact string matching using the FM-Index, in an input reference string T and a string query q :

1. Compute the Burrows-Wheeler Transform T^{bwt} of T, use it to find Occ and to get the FM-Index F.

2. Use Algorithm 2 to determine the index of the rotation in the implicit suffix array containing q

3. If such an index exists (i.e. q occurs at least once in T), use the Walk-Left Algorithm to walk back to the beginning of the text from this index

4. The number of steps corresponds to the offset of q in T

Performance Analysis As mentioned earlier in this document, the size of a genomic dataset is usually in the billion of entry range. Hence, we consider the following items in the previously defined process to be way too costly time or space-wise :

- Step 4 of the procedure requires a number of step linear to the length of T. Doing this may require billions of iterations over the lines 4 and 5 of Algorithm 1, meaning billions of memory access, which is way too much time consuming.

- Storing the whole FM-Index requires storing $(N + 1) * \sigma$ values, with N the length of T and σ the size of its alphabet Σ. This would require in our case at least 6 times the size of T in memory space, which is way too big.

In order to reduce both those costs, the following solution is introduced :

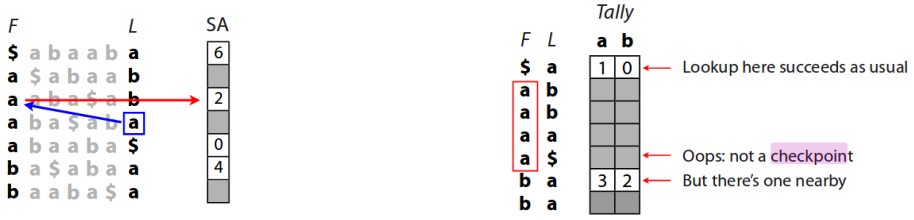
- Keep samples of the suffix array index (e.g. every 32^{th} row), when "walking" back along T^{BWT} check if an entry exists for the current position, if yes, then add this value to the number of step already done to obtain the position of q in T. This allows to Walk-Left algorithm to operate in a constant time, while limiting the space needed to store those values.

- Keep only some samples of the FM-Index (e.g. one every 128 entries), referred as " checkpoints ". The LF function would now take the index of the current symbol in T^{bwt} and find the closest surrounding checkpoint, then "walk" along T^{bwt}, counting the occurrences of the same symbol along the way and adding/subtracting (respectively if the closest checkpoint is above or below the current index) this count to the rank value stored in the checkpoint. This reduces greatly the space needed to store the index, while keeping a constant computation time for the LF function.

Conclure

2.2 Python Naive Implementation

In order to get a better grasp of the concepts involved in the FM-Index implementation, a first one has been done in *Python*. The main advantage of this language is the syntax simplicity, allowing to focus on the algorithm and data-flow aspects of the string matching



(a) SA sample example

(b) Checkpoint example

Figure 2.4: Illustration of the two proposed improvement

process while neglecting the data structures and global performances.

Good results have been reached using an adapted version of (CITER REPO), and this project has been used as a good basis for the less simple but way more precise *C++* implementation described in the next section. Below is a schema describing the software structure and execution flow.

[Poser un genre d'UML]

Using a terminal interface, the program is used as follow :

[Mettre un screenshot]

2.3 C++ Implementation

All the software components of this project interacting with the board will be written in *C++*. This choice is based on legacy reasons (*REDS* environment) and conceptual reasons. The latter comes from the need to define a precise encoding of the *nucleotide reads* to be used on the FPGA system along with well defined and space-optimized data structures to store the *BWT* in the memory, the suffix array samples and the FM-Index checkpoints.

2.3.1 Reads encoding and Data Structures

Reads encoding Considering there are 4 different nucleotides within any DNA sequence, adding the uncalled base N type read and the "$" equivalent character, we need to encode 6 symbols and thus need at least 3 bits. Therefore, it has been chose to use the following encoding :

Name	Encoding
"$"	000
A	010
C	100
G	101
T	110
N	111

With this defined, we can follow on the data type used in the software to represent the input text :

```
typedef unsigned char nucl_read;  /* Type on 8 bits (only last 3 used) */

static const nucl_read eos    = 0x00;           /* $ character */
static const nucl_read a_read = 0x02;
static const nucl_read c_read = 0x04;
static const nucl_read g_read = 0x05;
static const nucl_read t_read = 0x06;
static const nucl_read n_read = 0x07;
```

Data Structures The different needed data structures are the following :

Checkpoint Entry -

```
typedef struct checkpoint_entry
{
uint32_t eos_count = 0;
uint32_t a_count = 0;
uint32_t c_count = 0;
uint32_t g_count = 0;
uint32_t t_count = 0;
uint32_t n_count = 0;

} checkpoint_entry;
```

BWT Class Attributes -

```
/* Class attributes */
nucl_read * L;          /* Last column = BWT(T) */
```

```
nucl_read * F;          /* First column of ordered suffix array */
checkpoint_entry * occ; /* Occ array with each symbol 1st occ idx */
checkpoint_entry * checkpoints; /* FM-Index samples */
uint64_t * suffix_idx;  /* Sampled suffix array index's */
```

Below is a schema describing the program structure and execution flow :

[Justifier les types, faire un UML ou quelque chose comme ca pour illustrer le fonctionnement]

Using a terminal interface, this program is used as follow :

[Poser un screenshot d'execution]

2.4 Tests & Validation

[Benchmark pour C++ ??]

Chapter 3

Hardware Implementation of the FM-Index

The main scope of this thesis is to implement the aforementioned algorithm on a specific *FPGA* board, using a special type of memory to store the reference data, called *Hybrid Memory Cube* (HMC). Our hope is to attain a high parallelization factor, improving greatly performances over a large set of query. The different hardware tools used in this project will be described in this section.

3.1 Tools Introduction

3.1.1 Hybrid Memory Cube and Micron AC-510

Hybrid Memory Cube (HMC) is a high-performance *RAM* interface for stacked *DRAM* memory. Combining THROUGH-SILICON VIAS and *microbumps* (WIKIPEDIAA) to connect multiple layers of memory cells on top of each others, it offers very high throughput parallel serial bus' for i/o.

FAIRE UNE MEILLEURE DESCRIPTION

DECRIRE LA BOAR, PARLER DE PCI-EXPRESS AUSSI ET DE L'API

3.1.2 Vivado Project

3.2 Specifications

3.3 Model Conception

3.4 VHDL Implementation

3.5 Test & Validation

Chapter 4

On-Board Porting

4.1 Inclusion into the Vivado Project

4.2 C++ Interface Implementation

4.3 On-Board Testing

4.4 Validation

Chapter 5

Results

Chapter 6

Project Status and Possible Improvements

Chapter 7

Conclusion

7.1 Personal Comment

7.2 Acknowledgment

THANK YOU <3

Journal de Travail

23.02.2019

Séance avec M.Thoma et M.Wortenbroek - Explication plus détaillée du contexte du travail et du travail demandé. Présentation d'un plan initial de déroulement et définition d'une première étape, à savoir implémenter dans un language de haut niveau un programme permettant d'appliquer la BWT et l'indexage FM-Index à des entrées en format FASTA.

Installation - récupération du dépôt GitLab et mise en place de la structure de ce dernier. Début d'ébauche d'un cahier des charges et de rapport. Mise en place de l'environnement sur ma machine.

28.02.2018

Cahier des charges Complétion d'un cahier des charges un peu formel à faire valider par le professeur.

Documentation Récupération de documents (cours, ...) relatifs à la BWT et à l'indexage FM.

02.03.2018

Cahier des charges Finalisation du cahier des charges

Documentation Lecture et prise de note sur les principes de transformation et d'indexage

Rapport Début de la rédaction de l'introduction du rapport.

09.03.2018

Cahier des charges Dernière correction du cahier des charges.

Rapport Continuation de la rédaction de l'introduction du rapport.

Software Début d'implémentation du software (objectif **[2]**

Bibliography

David K. Gifford, Massachussets Institute of Technology (2014). *Library Complexity and Short Read Alignment (Mapping)*. URL: https://ocw.mit.edu/courses/biology/7-91j-foundations-of-computational-and-systems-biology-spring-2014/lecture-slides/MIT7_91JS14_Lecture5.pdf.

Dr Ananya Mandal, MD (2014). *What is Genomics ?* URL: https://www.news-medical.net/life-sciences/What-is-Genomics.aspx.

National Tsing Hua University (2013). *Burrows-Wheeler Transform* . http://www.cs.nthu.edu.tw/~wkhon/ds/ds10/tutorial/tutorial7.pdf.

Wikipedia, The Free Encyclopedia (2017). *FM-index* . URL: https://en.wikipedia.org/wiki/FM-index.