Description & Implementation of the ACMI Component

A Hardware Accelerator for the Vision Transformer Algorithm using MinHash to Lineage Assignment by ENICS Lab
Final Project - Computer Engineering, BIU

NOAM DIAMANT & ITAY GOLDBERG

July 31, 2024

הפקולטה להנדסה
ע"ש אלכסנדר קופקין
אוניברסיטת בר־אילן

Contents

Chapter 1

Theoretical Background

1.1 Minhash

MinHash is a variant of locality sensitive hashing designed to approximate the similarity between two datasets. The fundamental idea behind MinHash is that the probability that two sets have the same minimum hash value is equivalent to their Jaccard similarity. Jaccard similarity measures the proportion of shared elements between two sets equal to:

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|} \tag{1.1}$$

We use the Bottom-k Minhash whose overview is presented in Fig. 1.1 In the genomic context, the datasets comprise sequences of kmers. The signatures A and B are generated by a single hash function, h, which ranks the kmers by assigning a numerical value to each kmer. The k minimal ranking values comprise the signature subsets $S(A)$ and $S(B)$. The equation in Fig. 1.1 approximates Jaccard similarity between the two original kmer sequences.

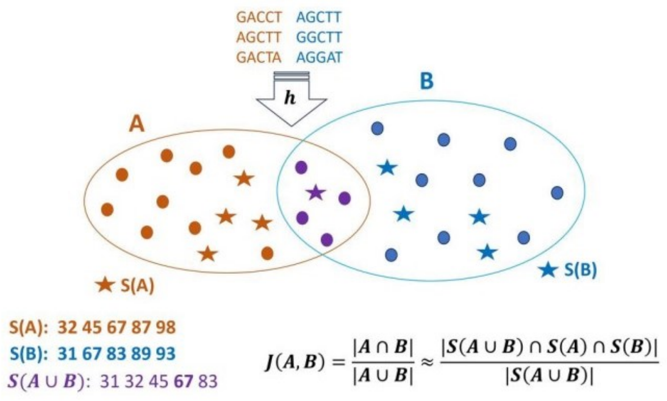
GACCT AGCTT
AGCTT GGCTT
GACTA AGGAT
h
A
B
S(A)
S(B)
S(A): 32 45 67 87 98
S(B): 31 67 83 89 93
$S(A \cup B)$: 31 32 45 67 83
$J(A,B) = \frac{|A \cap B|}{|A \cup B|} \approx \frac{|S(A \cup B) \cap S(A) \cap S(B)|}{|S(A \cup B)|}$

Figure 1.1: Minhash Bottom-k overview

We leverage Bottom-k to convert low coverage genomes into sequences of fixed-size fragments. Genomic samples that come from similar genomes are likely to have overlapping hash values, so that Bottom-k is instrumental in capturing their similarity

1.2 DNA Sequencing and Genome Analysis

DNA is composed of four nucleotides: Adenine (A), Guanine (G), Cytosine (C), and Thymine (T), which are frequently referred to as DNA basepairs, bases, or bps. Accordingly, a DNA data element is a DNA base that can have one of four values (A, G, C, and T). DNA sequencing is the process of determining the bases of a DNA chain in a given biological sample. Contemporary high-throughput DNA sequencers can sequence multiple biological samples in parallel. A DNA sequencing process, along with genome analysis, is carried out in several steps: (1) sample preparation; (2) DNA sequencing that generates multiple DNA reads (i.e., sub-sequences of the DNA sample); and (3) DNA classification, DNA read alignment, genome assembly, variant analysis, etc. While variant refers to a single change in a genome (i.e., a single base change mutation), lineage is a collection of variants that help define a specific line of an organism.

1.3 ViRAL Algorithm

ViRAL is the platform for a quick and accurate identification, classification, and lineage assignment of SARS-CoV-2 variants. It accepts an assembled genome and outputs the ordered list of most probable lineages such genome belongs to. If the top probabilities output by ViRAL are similar, this might indicate that the new genome belongs to an unknown lineage, creating a new node on the phylogenetic tree.

The top-level overview of ViRAL algorithm is presented in Figure 1.2. The input to ViRAL is an assembled genome in FASTA file format. To convert the genome into a sequence of feature vectors, we build a pipeline of preprocessing steps, comprising the feature extractor and the embedding layer. The feature extractor chooses a set of fragments (i.e., genome subsequences of fixed length), represents each fragment as a two-dimensional matrix, and outputs those genome fragment matrices (GFMs). Each GFM is fed into an embedding layer that consists of one neuron. This layer converts the GFM into a genome fragment vector, designated the feature vector. The feature vectors are fed into the ViT, which outputs the most likely assignment candidates (by attaching probabilities to each lineage).

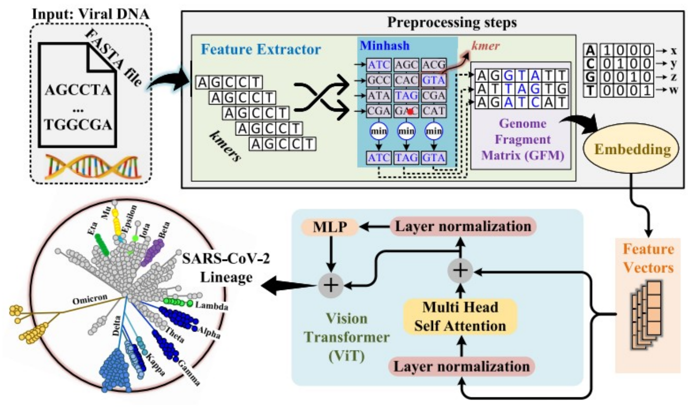

Figure 1.2: A general overview of the ViRAL algorithm.

The feature extractor together with the embedding layer transforms a genome into a sequence of representative feature vectors, which are the numerical representations of genome fragments. We implement the feature extractor using MinHash to preserve similarities in the genome. It allows the feature extractor to find features in the query genome that are shared by the known lineages. The embedding layer is used as a dimensionality reduction step. It allows for the reduction of ViRAL latency without affecting its assignment accuracy.

Chapter 2

Introduction

In this chapter, an extensive literature review was conducted in order to cover all possible subjects that affect the goals of this project. Understanding how the different components and architectures that are in use today and the modern solutions to the problem this work is trying to solve are important to better define the objectives of this research.

2.1 MinHash Accelerators

Several MinHash acceleration solutions have been proposed. MetaCache GPU uses MinHash for metagenomic classification and implements it on GPU. MSIM is a near-memory MinHash accelerator with a limited energy consumption reduction (26.4× vs. high-performance GPU), targeted for high-performance applications. In [64], MinHash is used for kmer clustering. An FPGA-based solution is limited to parallel calculation of 15 hash functions (whereas ViRAL calculates in parallel at least 256 hash functions). JACK-FPGA uses a cloud FPGA to accelerate MinHash. Scotch is an FPGA-based locality-sensitive hashing accelerator with a limited energy consumption gain (5× over high-performance GPU).

The purpose of MinHash acceleration in ViRAL framework is to balance the MinHash and Vision Transformer calculations while providing the highest energy efficiency, preferably sufficient for portable applications.

The MinHash Scheme

A modified MinHash is the second processing component of ViRAL, whose role is extracting the informative feature vectors from the genome. MinHash, or the min-wise independent permutations scheme is a technique for similarity estimation. MinHash is used in many tasks in computational biology, including genome assembly, metagenomic gene clustering, and genomic distance estimation. MinHash implements the following sketch function:

Given a set of characters A, a compression factor n, and a hash function h, the elements of the set A are hashed using function h to generate the set $H(A)$. Then the elements of $H(A)$ are sorted and the inputs to the smallest n elements are returned as described in Algorithm 1.

Algorithm 1: The Sketch Algorithm `sketch(A)`

```
Require: set $A = \{a_1, \dots, a_{|A|}\}$, compression parameter $n$ and a hash function $h$
sketch $\leftarrow \emptyset$
$H(A) \leftarrow (h(a_1), \dots h(a_{|A|}))$
Sort $H(A)$ to get $(h(a_{i1}), h(a_{i2}), \dots, h(a_{i|A|}))$
return $(a_{i1}, a_{i2}, \dots, a_{in})$
```

The subset obtained by applying the sketch algorithm provides a good estimate for the Jaccard index, defined as follows: Given two sets A, B, the Jaccard index is

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|}$$

Formally, given two sets, A and B,

$$\frac{|sketch(A) \cap sketch(B) \cap sketch(A \cup B)|}{|sketch(A \cup B)|} \approx J(A,B)$$

We use the sketch algorithm to extract feature vectors from a genome, as presented further.

Feature Extractor

The first part of the ViRAL pipeline receives the assembled genome and outputs matrices that represent genome fragments. Feature extractor employs MinHash to find similar fragments (i.e., features) in different genomes.

Algorithm 2: The Feature Extracto

```
Require: genome $g$, compression parameter $n$, fragment size $f$, kmer size $k$, hash function $h$
$G \leftarrow \{gi \equiv g[i : i+k]\}$ s.t. $i \in [0, n-k+1]$
kmers $\leftarrow$ sketch$(G, n, h)$
left $\leftarrow$ floor $\left(\frac{f-k}{2}\right)$
right $\leftarrow$ ceil $\left(\frac{f-k}{2}\right)$
fragments $\leftarrow \{g[i - \texttt{left}, i + \texttt{right}] : gi \in \texttt{kmers}\}$
return one-hot-encode(fragments)
```

One hot encoding of a fragment refers to the operation of transforming each base in the fragment as follows: $A \rightarrow [1,0,0,0]$; $C \rightarrow [0,1,0,0]$; $G \rightarrow [0,0,1,0]$; $T \rightarrow [0,0,0,1]$; $N \rightarrow [0,0,0,0]$.

For the example please refer to Figure 2.1(b).

An example is illustrated in Figure 2.1(a). Input is a genome sequence g of length $N = 19$. We generate all possible kmers (genome sub-sequences of length k) G (where $k = 4$), and extract ($n = 3$) 4mers from the genome using the MinHash scheme (i.e., `sketch`(G, n = 3, h)). The next step is to generate fragments of length $f = 8$ (by extending each of three 4mers by $\frac{f-k}{2} = \frac{8-4}{2} = 2$ bases in each direction). The last step of this workflow is to encode each basepair of the fragments using one-hot encoding.

For the genome sequence g of length N, the extractor first generates a set G of all possible kmers. It then sketches (i.e., applies MinHash sketch function to) the set of kmers, G, to extract n representative kmers of the genome (Algorithm 1). Those kmers are used as anchors to be expanded to generate fragments. The last part is transforming a fragment into a numerical matrix where the genome basepairs A, G, C, and T are encoded using one-hot encoding as described in Algorithm 2.

Embedding Layer

The input of the embedding layer is a genome fragment matrix of dimensions $f \times 4$ (i.e., f represents the fragment length and 4 is the dimensionality of the basepair one-hot encoding). It outputs a feature vector of size f. The embedding layer consists of one neuron which performs a base-wise linear transformation. Specifically, the embedding layer defines 5 learnable parameters, 4 weights w_A, w_C, w_G, w_T, and one bias term b_N. Given a genome fragment matrix, we transform each base (represented in one-hot encoding) to a numerical token as presented in Figure 2.1(b), which also contains an example in which we embed a fragmented matrix of dimensions 8×4.

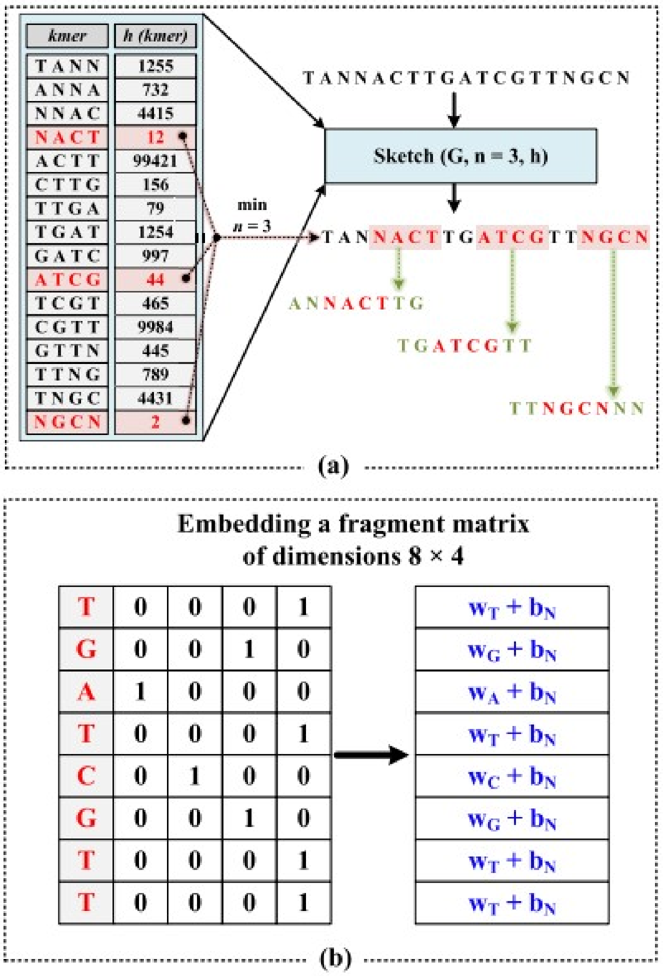

Figure 2.1: (a): Feature extractor workflow example. (b): Embedding a fragment matrix of dimensions 8×4.

2.2 System Architecture

Figure 2.2 illustrates the system view of the ViRAL platform. DNA samples are prepared and sequenced. The assembled genome material is fed into ViRAL, whose main components are MinHash and ViT accelerators, implemented by ACMI, a standalone ACcelerated MInHash ASIC, and NVidia GPU, respectively.

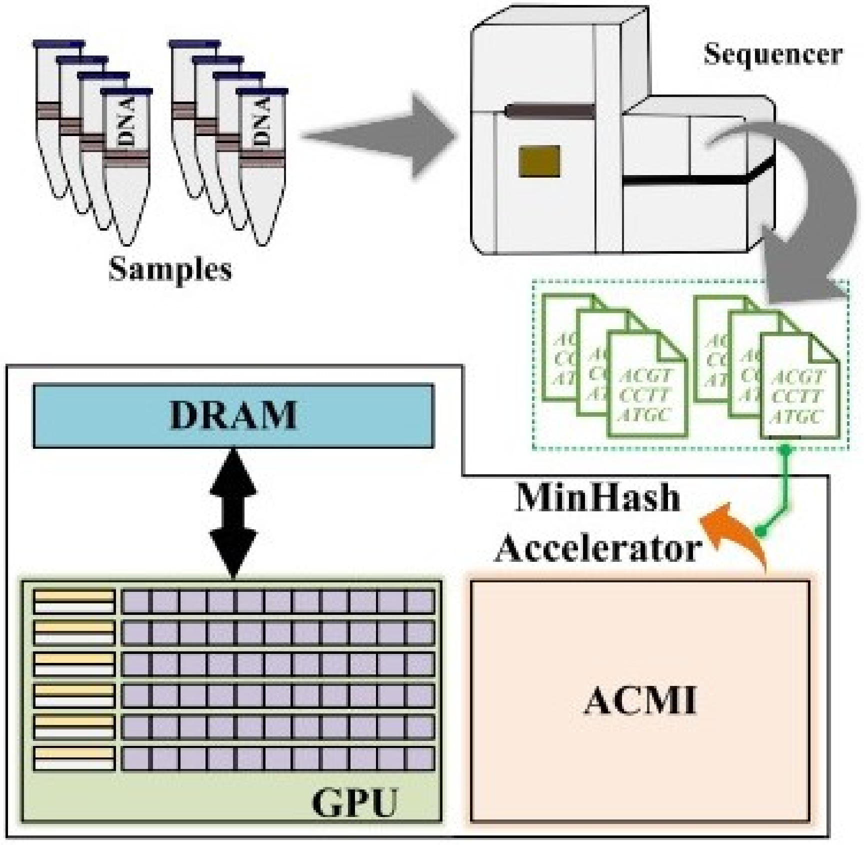

Figure 2.2: System view of the accelerated ViRAL platform featuring MinHash accelerator ACMI, combined with NVidia GPU.

ACMI is a throughput-oriented streaming platform. ACMI (1) inputs a genome $i + 1$, (2) processes both the previously received genome i and the genome $i + 1$, and (3) outputs the genome i related results, in parallel. ACMI inputs the accession genome one DNA base per cycle (on average). In the case of SARS-CoV-2 virus whose size is approximately 30K DNA bases, it takes approximately 30K cycles to upload a single genome. The result of ACMI is a Genome Fragment Matrix (GFM). Both the compression factor and the fragment_size are 256, hence the GFM is a 256×256 matrix of 4-byte elements. To balance the pipeline, ACMI is designed to output 8B/cycle, thus taking approximately $256 \times 256 \times 4/8 = 32K$ cycles to output the GFMs.

Since ACMI operating at 500 MHz requires I/O throughput of less than 5 GB/s to achieve its optimal performance, data connectivity can be implemented by PCIe 3.0 or higher.

FVThe ACMI architecture is presented in Figure 2.3. ACMI comprises the Fragment Memory (FM), the kmer buffer, the kmer index counter, Hasher, Sorter, and Extender units. Hasher performs hashing of the input kmer sets using MurmurHash3, which is a non-cryptographic hash function suitable for general hash-based lookup. Sorter sorts the hashed kmers (kmer signatures) and outputs 256 smallest signatures. Extender extends the signatures into larger DNA fragments and generates the GFMs.

While Hasher and Sorter process kmers in a pipelined fashion, "on the fly", the Extender requires access to the entire genome. Therefore, the Fragment Memory (FM) unit is organized as a double buffer, comprising buffers FM1 and FM2, operating in two phases: in phase 1, FM1 receives genome i, which is subsequently hashed and sorted. At the end of the phase, FM1 and FM2 are logically swapped. In phase 2, FM2 receives genome $i + 1$, which is being processed by the Hasher and Sorter. At the same time, FM1 serves the Extender, which generates the GFMs of the genome i. At the end of the phase, the buffers are swapped again, and so on. ACMI pipeline timing is shown in Figure 2.4.

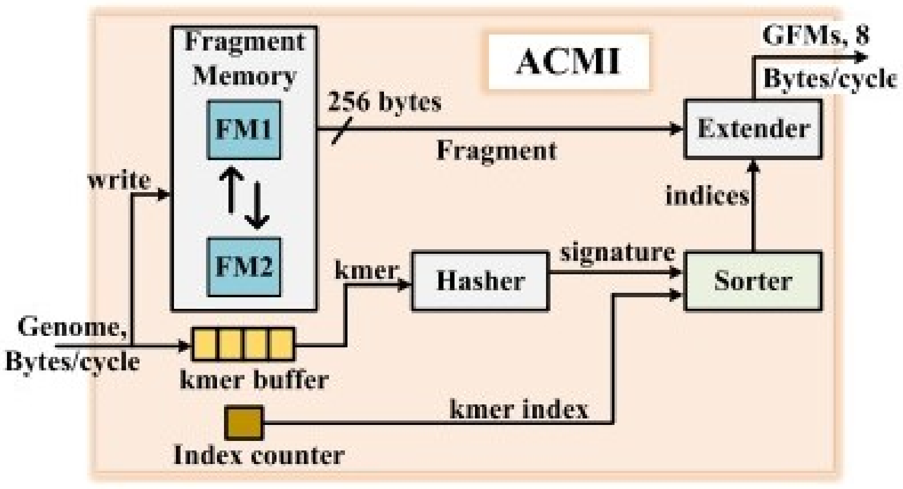

Figure 2.3: The top-level architecture.

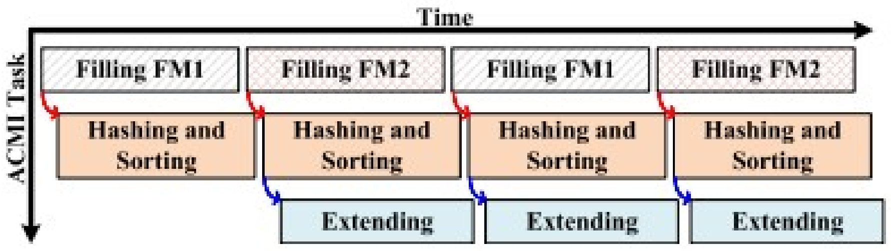

Figure 2.4: ACMI pipeline timing.

2.2.1 Kmer Buffer and Index Counter

The kmer buffer converts the DNA base stream received by ACMI into the stream of kmers (16mers). The index counter keeps track of the kmer index (the position of the kmer on the genome sequence).

2.2.2 Fragment Memory

The Fragment Memory buffers enable parallel write and read. The write operation occurs at a single-byte granularity. The read occurs in chunks of 256 bytes. As depicted in Figure 2.5, each FM buffer comprises eight 128×32 byte SRAM modules and can store up to 32KB of genomic information. FM address is composed of the row pointer (7 MS-bits), the SRAM module pointer (the following 3 bits), and the byte offset (the byte address within the row, 5 LS-bits). Since the read data granularity is 256 bytes, only the 7 MS-bits of the address are used in read access.

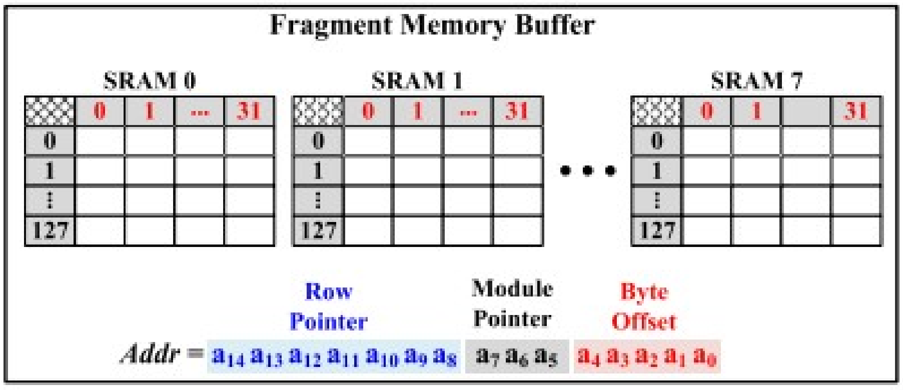

Figure 2.5: A fragment memory buffer comprising 8 128 $\times$ 32 bytes SRAM modules.

2.2.3 Sorter

The Sorter maintains a list of compression factor = 256 lexicographically smallest kmer signatures. It receives the signatures from the Hasher, along with their indices. After all kmers of the genome are hashed and sorted, Sorter transfers the entire list of 256 indices of the smallest signatures to the Extender (in parallel). At this point, the FM buffers are swapped, and the Extender begins processing the index data of the recently hashed and sorted genome, while a new genome is being written to the other FM buffer, and hashed and sorted by ACMI in a pipeline fashion.

The hashed kmers (signatures) are transferred sequentially from the Hasher to the Sorter. In each cycle, each comparator in the chain receives from its left neighbor a tuple comprising a kmer signature and its index, and compares such signature with the minimal signature value the comparator stores. If the new signature is smaller (lexicographically) than the stored one, the comparator saves the new tuple and outputs the old tuple, otherwise, it outputs the new tuple. After all, signatures are processed, the comparator chain retains the 256 smallest signatures and their indices (which are afterwards transferred to the Extender).

2.2.4 Extender

Extender converts each of the 256 kmers with the smallest signatures into a 256-base wide DNA fragment, by extending such kmer left and right. To accomplish that, Extender reads the DNA fragments directly from one of the FM buffers, using the index it receives from the Sorter to calculate the FM row pointer.

1. Calculate the fragment position on the genome sequence frag_idx $= \left\lfloor \frac{(kmer\_idx - \lfloor \frac{\text{frag\_len} - \text{kmer\_len}}{2} \rfloor)}{\text{frag\_len}} \right\rfloor$.
2. Read the DNA fragment pointed to by frag_idx.
3. Read the DNA fragment from the next FM buffer row, addressed by frag_idx $+ 1$.
4. Render the extended DNA fragment by concatenating the relevant parts of those two consecutive DNA fragments.

The 256 extended DNA fragments comprise the GFMs that become the output of ACMI.

An example of extension is shown in Figure 2.6. Suppose that the kmer length is 4 (a 4-mer), the fragment length is 8, and the size of the genome is 32. The Extender has to extend the 4-mer with index 9 (i.e., the 4-mer TAAG marked in red in Figure 2.6). In this case,

$$\text{frag\_idx} = \left\lfloor \frac{(kmer\_idx - \frac{\text{frag\_len}-\text{kmer\_len}}{2})}{\text{frag\_len}} \right\rfloor = \left\lfloor \frac{(9 - \frac{8-4}{2})}{8} \right\rfloor = \left\lfloor \frac{7}{8} \right\rfloor = 0.$$

Hence the Extender will read from FM buffer fragments 0 and 1. It will then concatenate the relevant bases using shift operations, render a single 8-base wide DNA fragment, encode the bases using one-hot encoding, and append the encoded extended DNA fragment to the GFMs.

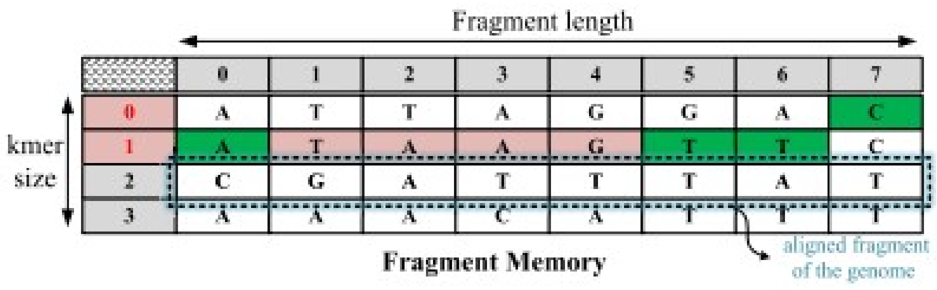

Figure 2.6: Example of extension. The fragment length is 8, the kmer size is 4. Each line represents an aligned fragment of the genome. The bases in pink are the bases of the 4mer at position 9. The extended fragment includes two bases (marked in green) to the left and to the right of the original 4mer.

Chapter 3

Major Modules

3.1 Fragment Memory

3.1.1 Component Implementation

The structure of the component

Ports and Subcomponents

Computational Logic

3.1.2 Codes & Tests

3.2 Hasher

Hasher is a hardware component that implements a single algorithm. According to the definitions of the article, the implementation hash function is Murmur3.

MurmurHash, a non-cryptographic hash function, is distinguished by its expeditious and efficacious computation of hash values, which are instrumental for general lookup operations. This algorithm is characterized by its operational simplicity and rapid data processing capabilities, rendering it particularly suitable for hash-based structures such as hash tables, caches, and checksums. The nomenclature 'MurmurHash' is derived from the algorithm's fundamental operations—'multiply' and 'rotate'—which are pivotal to its functionality. The most recent iteration, MurmurHash3, introduces enhancements that include the provision of both 32-bit and 128-bit hash values, along with optimizations for various computing platforms, thereby affirming its adaptability and extensive applicability in various domains.

3.2.1 Component Implementation

In each clock cycle, the hasher receives a single k-mer value represented by 4 bytes. Additionally, it maintains a fixed SEED value, also encoded as 4 bytes. Guided by the fixed constants, the hasher performs multiplications and rotations on the input k-mer. These operations transform the k-mer into a hash value. Finally, an approver issues a single signature based on the resulting hash value.

The structure of the component

Ports and Subcomponents

Computational Logic

3.2.2 Codes & Tests

Listing 3.1: hasher.sv

```
module murmur_4bytes
  #(
    parameter HASHER_DATA_BITS = 32
  )(
    input wire [HASHER_DATA_BITS-1:0]  seed,
    input wire [HASHER_DATA_BITS-1:0]  kmer,
    output wire [HASHER_DATA_BITS-1:0] signature
  );

  assign signature = hasher(seed, kmer);

  function [HASHER_DATA_BITS-1:0] hasher(input [HASHER_DATA_BITS-1:0] seed, kmer
    logic [HASHER_DATA_BITS:0]          k, key;
    localparam [HASHER_DATA_BITS:0]     c1 = 'hcc9e2d51;
        localparam [HASHER_DATA_BITS:0]     c2 = 'h1b873593;
    localparam [HASHER_DATA_BITS:0]     m =    5;
    localparam [HASHER_DATA_BITS:0]     n =   'he6546b64;
        begin
      k = kmer;
      k = k * c1;
      k = {k[HASHER_DATA_BITS-16:0], k[HASHER_DATA_BITS-1:HASHER_DATA_BITS-15]};
          k = k * c2;
          key = seed;
      key = key ^ k;
      key = {key[HASHER_DATA_BITS-14:0], key[HASHER_DATA_BITS-1:HASHER_DATA_BITS
      hasher = key * m + n;
    end
  endfunction

endmodule

module ROL13
  #(
    parameter HASHER_DATA_BITS = 32
  )(
    input wire [HASHER_DATA_BITS-1:0] roll_in ,
    input wire [HASHER_DATA_BITS-1:0] roll_out
  );

  assign roll_out[HASHER_DATA_BITS-1:0] = {roll_in[HASHER_DATA_BITS-14:0], roll_

endmodule

module ROL15
  #(
    parameter HASHER_DATA_BITS = 32
  )(
    input wire [HASHER_DATA_BITS-1:0] roll_in ,
    input wire [HASHER_DATA_BITS-1:0] roll_out
  );

  assign roll_out[HASHER_DATA_BITS-1:0] = {roll_in[HASHER_DATA_BITS-16:0], roll_

endmodule
```

3.3 Sorter

The sorter, part of the ACMI system, follows the hasher in the processing sequence. It receives both the signature and its corresponding index from the hasher and the index counter. The sorter begins its task upon receiving a completion signal from the hasher, allowing it to process the result immediately within the same clock cycle.

3.3.1 Component Implementation

The structure of the component

The component maintains a table when, in each clock cycle, the component receives a new value and updates the table according to logic. The table contains 256 rows. Each row consists of three values: signature, index and position. The following is an example for three values stored in the table:

Signature [32bits]	Index [10bits]	Position [8bits]
F2045678 h	0101010101	FF h
02045678 h	0000011111	2C h
12045678 h	1100010000	88 h

As we explained earlier, the signature and index are data received in each clock cycle. The values' position and entry into the table are determined according to logic. Next to this table there are 256 comparators, a comparator for each row. All the comparisons will perform a value comparison between the new signature and the signature that the row contains.

Ports and Subcomponents

The component has 4 inputs. The component receives a clock and reset from the top model (ACMI). From the hasher component it receives the generated signature. In addition, the index corresponding to the received signature is obtained from the index counter. The component outputs to the extender the updated list of indexes found in the table.

Computational Logic

Each clock cycle receives a signature (32 bits) and an index (10 bits) The sorter has access to a table containing 256 rows. Each line has: a signature, an index of the signature, and a position (8 bits). Position, describes the order of the signatures by value so that in the row of the biggest signature FFh appears and in the smallest 00h The sorter works as follows:

Clock Rise the sorter checks whether the new signature is greater than all the signatures in the table by 256 equals in parallel. For each comparator there is a signal indicating the result of the holocaust. The new position is calculated for all places in the table.

Clock Fall For each signature smaller than the new signature, the position decreases by one as it was calculated. For the signature whose position is FFh - the new signature overrides the old signature and the position also receives the sum of the results of the equations.

While the clock is up, the expander can read the indexes in the table without fear that we will get a meta-stable value there or that is not updated for the last iteration.

3.3.2 Codes & Tests

3.4 Extender

The Extender is the final component of the ACMI. It uses the information from the FM and it performs the expansion operation on the values that the sorter found to be the smallest. The extender will extract an 8-base wide DNA fragment for the relevant bases

3.4.1 Component Implementation

The structure of the component

The module uses generate blocks to create parallel logic for each k-mer index. For each index:

1. It calculates the start position by subtracting half the difference between fragment length and k-mer length from the k-mer index.
2. For each base in the fragment: It calculates the current position. If the position is within the valid memory range, it extracts the base from the fragment memory. If the position is out of range, it inserts a placeholder base ('N', represented as 0x0000).

Ports and Subcomponents

The Extender receives two inputs and outputs two outputs. The extender receives the indexes of the 256 smallest signatures that maintained in the Sorter component. Addresses the fragment memory with the calculated new indexes and receives as input the stored bytes accordingly.

Computational Logic

According to the definition we presented in the introduction, we implemented the calculation of the index to memory access. As mentioned in the fragment memory section, the access to read from the fragment memory is done by only 7 bits in order to import 256 bytes of genome.

3.4.2 Codes & Tests

Chapter 4

Integration & Submodules

Chapter 5

Results & Performance

Chapter 6

Summary & Discussion