Technion – Israel institute of technology

Faculty of electrical and computer engineering

VLSI Lab

Project A book

Subject:

LSH minhash hardware accelerator for DNA classification

Students:

Boran Swaid – 211516836

Dima Ali-Salih – 211840889

Supervisor:

Robert Hanhan

Winter semester 2023

Table of content

List of illustrations

Abstract

We propose a novel LSH-Minhash accelerator a novel hardware for read classification using the big data technique minhashing. Our approach performs context-aware classification of reads by computing representative subsamples of k-mers within both, probed reads and their reversed reads and locally constrained regions of the reference genomes. As a result, the accelerator consumes significantly less memory compared to the state-of-the-art read classifiers Kraken and CLARK while achieving highly competitive sensitivity and precision at comparable speed. For example, using NCBI RefSeq draft and completed genomes with a total length of around 140 billion bases as reference, our accelerator's database consumes only 17.5 GB of memory while both Kraken and CLARK fail to construct their respective databases on a workstation with 512 GB RAM and MetaCache consumes 62GB.

Motivation

The motivation for building a hardware accelerator for LSH MinHash is the need for faster, more scalable, and energy-efficient solutions in data processing.

Due to the large number of reads produced by modern high-throughput sequencing technologies and the rapidly increasing number of available reference genomes corresponding software tools suffer from either long runtime, large memory requirements or low accuracy such as Kraken and CLARK.

Background

DNA sequence classification

What is DNA? A DNA sequence contains only the letters A, C, G and T. (Each letter represents a small molecule, and a DNA sequence is a 'macromolecular' chain of them.) Each letter in a DNA sequence is called a base, basepair, or nucleotide. Normally, DNA occurs as a double strand where each A is paired with a T and vice versa, and each C is paired with a G and vice versa. The reverse complement of a DNA sequence is formed by reversing the letters, interchanging A and T and interchanging C and G. Thus, the reverse complement of ACCTGAG is CTCAGGT.

Classification is used to predict the category of items with unknown labels based on data derived from a training set. They work well with discrete variables or categorical data, acting as the perfect solution for analysing DNA data. Sequence classification can be used to analyse DNA sequences for sequence similarity (of structure or function) and predict a category or class (in terms of function and relationship) for other sequences. Classification can help assist in gene identification in DNA molecules.

DNA classification is performed as follows: a sample potentially containing DNA of multiple organisms is obtained and prepared. The sample is sequenced; the sequencer outputs DNA reads; DNA sequencers are prone to sequencing errors, such as indels and replacements.

DNA reads (soiled with sequencing errors) are processed by a classification application that potentially associates each DNA read with a certain species in its database and returns the window index of the matched species.

1. The Process of DNA sequence classification

Despite the significant advancements in sequencing technologies, there are several challenges and limitations that create bottlenecks in the sequencing workflow:

1. Throughput and speed: even with high-throughput sequencing machines, the speed at which genomes can be sequenced might not be sufficient to keep up with the demands, especially in clinical settings where rapid results are crucial. This limitation hampers the quick analysis of large volumes of genetic data.
2. Data Processing and Analysis: the generation of massive amounts of raw sequencing data requires substantial computational power and sophisticated bioinformatics tools for data processing, alignment, and interpretation. Analyzing and making sense of this data can be time-consuming and computationally intensive, creating a bottleneck in the sequencing pipeline.
3. Accuracy and Error Correction: Despite the accuracy of modern sequencing technologies, errors can still occur, and correcting these errors or distinguishing between real variations and technical artifacts can be challenging and time-consuming.

Minhash Algorithm and Murmur3 function

Minhash algorithm or locality sensitive hashing is a technique for quickly estimating how similar two sets are using Jaccard similarity coefficient which is a commonly used indicator of the similarity between two sets. Let U be a set and A and B be subsets of U, then the Jaccard index is defined to be the ratio of the number of elements of their intersection and the number of elements of their union: $J(A,B) = \frac{|A \cap B|}{|A \cup B|}$

This value is 0 when the two sets are disjoint, 1 when they are equal, and strictly between 0 and 1 otherwise. Two sets are more similar (i.e., have relatively more members in common) when their Jaccard index is closer to 1. The goal of MinHash is to estimate J(A, B) quickly, without explicitly computing the intersection and union.

In this project we used Murmur3 Minhash function to implement the Algorithm. Murmu3 is widely used in streaming algorithms because it is fast to compute and achieves low collision rates. it computes the hash value of the input using a sequence of multiplications, additions, and bitwise logic operations such as or, xor and bit shifts.

Algorithm of murmur3 hash function:

```
algorithm Murmur3_32 is
   // Note: In this version, all arithmetic is performed with unsigned 32-bit integers.
   //      In the case of overflow, the result is reduced modulo 2^32.
   input: key, len, seed

   c1 ← 0xcc9e2d51
   c2 ← 0x1b873593
   r1 ← 15
   r2 ← 13
   m ← 5
   n ← 0xe6546b64

   hash ← seed

   for each fourByteChunk of key do
      k ← fourByteChunk

      k ← k × c1
      k ← k ROL r1
      k ← k × c2

      hash ← hash XOR k
      hash ← hash ROL r2
      hash ← (hash × m) + n

   with any remainingBytesInKey do
      remainingBytes ← SwapToLittleEndian(remainingBytesInKey)
      // Note: Endian swapping is only necessary on big-endian machines.
      //      The purpose is to place the meaningful digits towards the low end of the value,
      //      so that these digits have the greatest potential to affect the low range digits
      //      in the subsequent multiplication.  Consider that locating the meaningful digits
      //      in the high range would produce a greater effect upon the high digits of the
```

```
        //      multiplication, and notably, that such high digits are likely to be discarded
        //      by the modulo arithmetic under overflow.  We don't want that.

        remainingBytes ← remainingBytes × c1
        remainingBytes ← remainingBytes ROL r1
        remainingBytes ← remainingBytes × c2

        hash ← hash XOR remainingBytes

    hash ← hash XOR len

    hash ← hash XOR (hash >> 16)
    hash ← hash × 0x85ebca6b
    hash ← hash XOR (hash >> 13)
    hash ← hash × 0xc2b2ae35
    hash ← hash XOR (hash >> 16)
```

(from Wikipedia)

Project Goals and Requirements

DNA pre-alignment is a process used in DNA genome analysis pipeline to filter our reads which do not belong to a DNA reference. In this project we will design a hardware accelerator for DNA pre-alignment used to detect and classify DNA reads.

The main goals of this project are:

1. Design a novel accelerator for genomic applications: The core innovation of this project lies in the hardware accelerator specifically tailored to address the computational demands of genomic applications.
2. Gain speedup over commercial software-based tools: achieve a substantial speedup when compared to existing commercial software-based tools commonly employed for genomic data analysis such as MetaCache, Kraken and CLARK.
3. To learn digital VLSI design tools and flow such as Innovus and Synopsys using Systemverilog language.

Alternative Solutions

1- MetaCache: a novel software for read classification using the big data technique minhashing. The approach performs context-aware classification of reads by computing representative subsamples of k-mers within both probed reads and locally constrained regions of the reference genomes. As a result, MetaCache consumes significantly less memory compared to the state-of-the-art read classifiers Kraken and CLARK while achieving highly competitive sensitivity and precision at comparable speed.
MetaCache's database consumes only 62 GB of memory while both Kraken and CLARK fail to construct their respective databases on a workstation with 512 GB RAM. The experimental results further show that classification accuracy continuously improves when increasing the amount of utilized reference genome data.

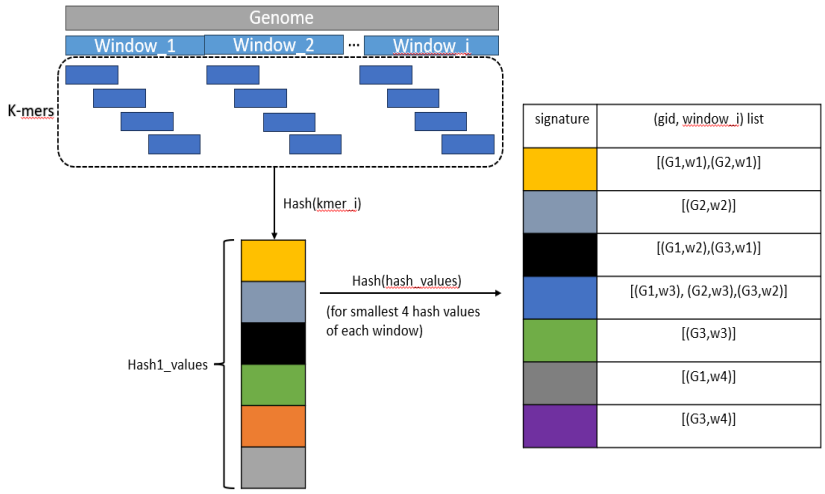
2- EDAM: Unlike state-of-the-art approximate search solutions that tolerate certain Hamming distance between the query pattern and the stored data, EDAM tolerates edit distance, which makes it especially efficient in applications such as text processing and genome analysis. EDAM was designed using a commercial 65 nm 1.2 V CMOS technology and evaluated through extensive Monte Carlo simulations, while considering different process corners. Simulation results show that EDAM can achieve robust approximate search operation with a wide range of edit distance threshold levels. EDAM is functionally evaluated as a pathogen DNA detection and classification accelerator. EDAM achieves up to 1.7× higher F1 score for high-quality DNA reads and up to 19.55× higher F1 score for DNA reads with 15% error rate, compared to state-of-the-art DNA classification tool Kraken2. Simulated at 667 MHz, EDAM provides 1, 214× average speedup over Kraken2. This makes EDAM suitable for hardware acceleration of genomic surveillance of outbreaks, such as the ongoing Covid-19 pandemic...

We chose to apply murmur hash function and MetaCache algorithm in hardware as our solution mainly for speed, simplicity, and low collision probability.

The Description of the Algorithm

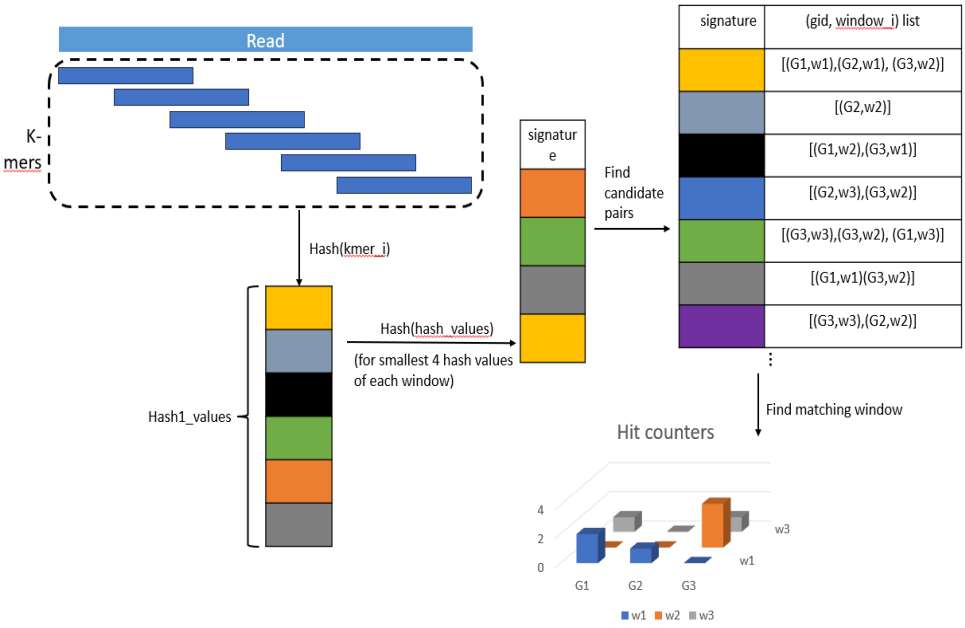
The algorithm can be divided into two sections: offline that contains the database construction process and online that contains the classification process.

- Offline:
  1- Pick 3 reference genomes (the code can be updated for more).
  2- For each reference genome, place it in memory as follows:
     i. Divide ref genome into windows of length 128 byte.
     ii. Divide every window into k-mers of length 16 byte with overlap of 8 byte.
     iii. For every window, find its signature using murmur function.
     iv. Hash the smallest 4 signature values using murmur function into the table that represents the memory.



2. offline algorithm description

- Online:
  1- Input: read of length 128 byte.
  2- Create an empty list called hash2_values.
  3- Hash the read into memory as described before in the offline section and save its hash values in hash2_values list.
  4- Find candidate pairs by going on the hash2_values list and for each value go over the list of windows that matches this value and increment the counter of this window and this genome id.
  5- Do the same steps as before but for the reversed read.
  6- Return the window index and genome id which has the highest count of hits from both the read and the reversed read, if there isn't, return (-1, -1).



3. online algorithm description

So, at the end of the classification process if the read belongs to one of our reference genomes, we'll get the matching window into the matching reference genome, and if the read doesn't match any genome, we'll get (-1, -1) that indicates that there is no matching.

Software Implementation

Researching parameters using software is essential for a multitude of reasons. Firstly, it allows scientists and engineers to gain a deeper understanding of complex systems and phenomena by simulating and manipulating various variables in a controlled environment.

The algorithm was implemented in python, and it was used to research the parameters that gives us the best results, which means the highest accuracy rate of classifying reads that will be explained later.

The project includes two files: LSH.py that includes the LSH Minhash class and the algorithm implementation and test_bench.py that includes the tests that research the algorithm parameters.

Content of LSH.py file:

The main class called MinHash, and its parameters are:

- G: length of reference genome
- L: length of window
- K: length of kmer
- S: number of smallest s values
- OV_L: length of overlap
- sigTable: table that saves signatures (hash values)

main functions of the algorithm:

- murmurFunc (key, len, seed): calculates hash value of key that represents kmer in our algorithm, len represents the len of the key and seed is a random number, we chose the number 123.
- Build windows(g): g represents a given genome, the function divides the genome into windows of length L and returns a list of the windows.
- hashKmers (windows, gID, list): this function hashs kmers of every window in windows onto sigTable by calling murmur3 function twice for every kmer.
- findcandidatePairs(read): this function finds candidate pairs of read and returns list of candidatepairs.
- findMaxCount(candidates): returns max count window index and genome id.

This code was implemented in two versions, one version has the manual implementation of murmur3 function, and other version uses murmur3 function from the library mmh3 of python. That helped us validate our implementation of murmur algorithm.

Content of test_bench.py file:

The test includes two stages, preparing the database for the classifier and testing the classification parameters and accuracy.

We used two reference genomes to do so, the first genome is COVID19, this genome is used as the base reference genome of the algorithm and to generate the reads for testing the true positive and false negative cases, the second genome is Bacteria and it's used to generate reads for testing the true negative and false positive cases.

Stage 1:

1. We defined a list of MinHash instances for several computations of parameters, called mh_list.
2. Generate reads from COVID19 reference genome with length of window, this list called readsA.
3. Generate reads from Bacteria reference genome with length of window, this list called readsB.
4. Map COVID19 reference genome to memory by running an offline algorithm.

Stage 2:

There are two major tests:

- Low_error_rate test:
  1. goes over readsA and readsB and adds errors to the reads with 0.2% deletion, 0.2% insertion, 3.6% replacements of letters in the genomes, randomly.
  2. Makes classification by running online algorithm on the reads with errors and calculates the TP, FN, FP, TN.
  3. Calculates the accuracy percent of the algorithm by:

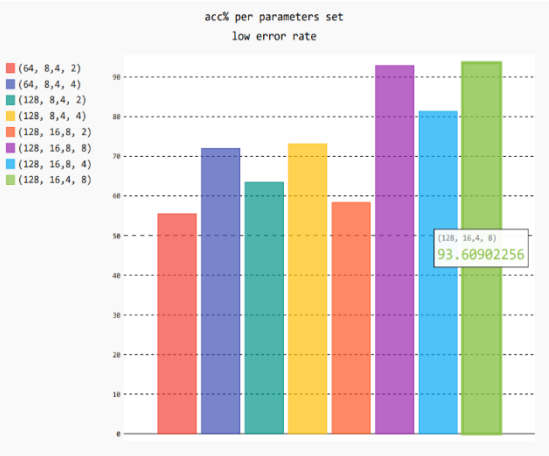
$$accuracy = \frac{TP + TN}{TP + FN + TN + FP}$$

- High_error_rate test:
  1. goes over readsA and readsB and adds errors to the reads with 7% deletion, 7% insertion, 1% replacements of letters in the genomes, randomly.

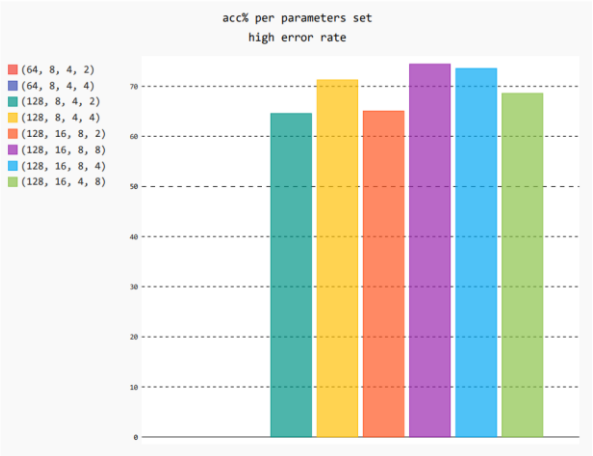
  2 + 3 +4 same as before.

(The error rates were taken from EDAM paper)

Results graphs of the tests:

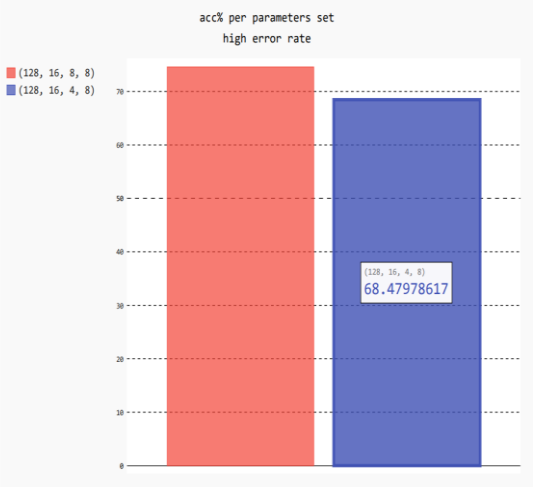


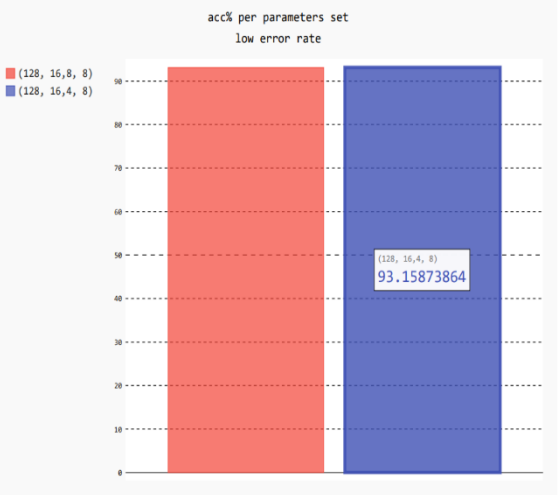
4. low error rate histogram



5. high error rate histogram

Zoom in:





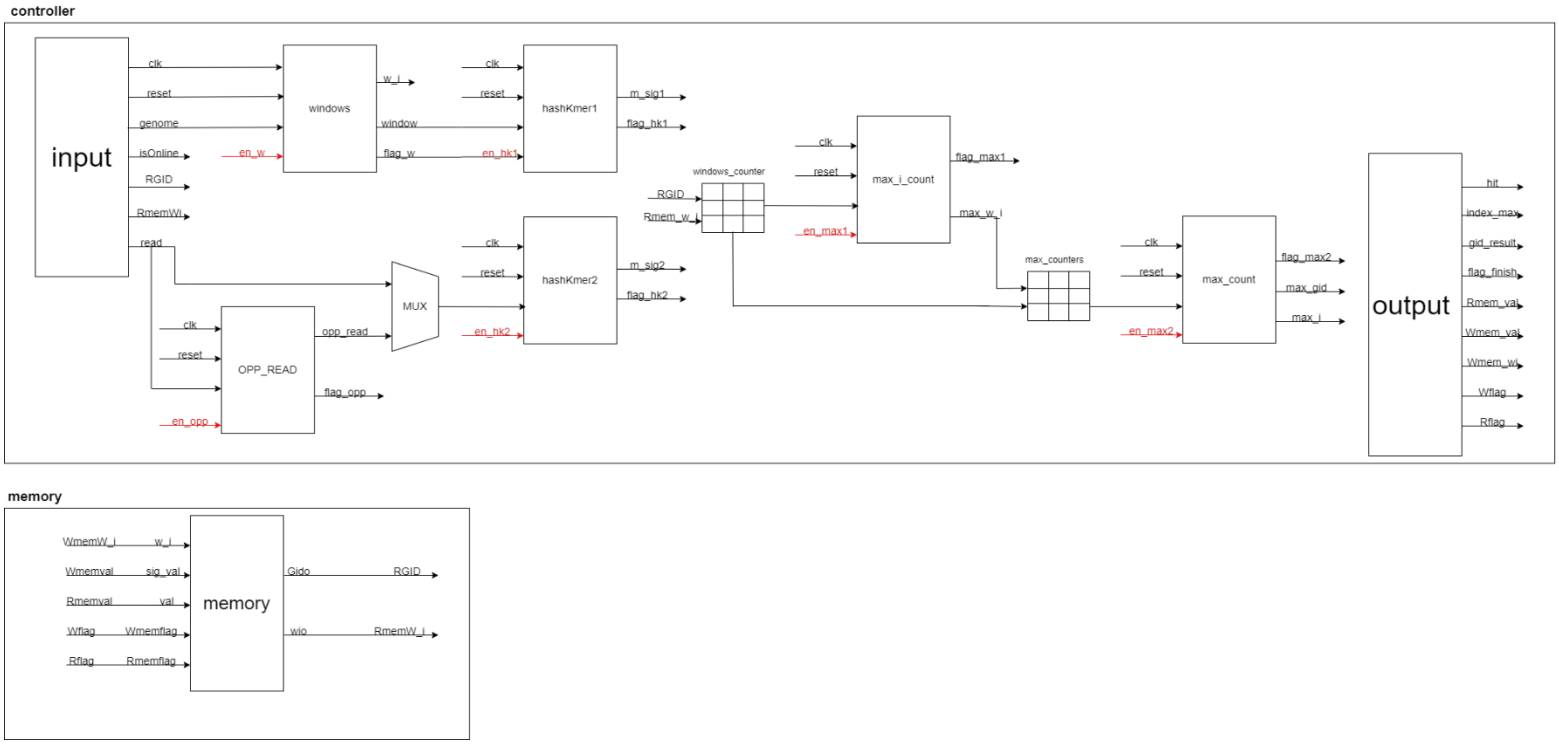
6. error rate histograms, zoom in

As we see in the histograms above, although the red bar gives better results than the blue one, we chose to use the blue bar parameters because of the tradeoff of memory and time consumption against accuracy.

Hardware part

Architecture

Here is an upper view of the architecture:

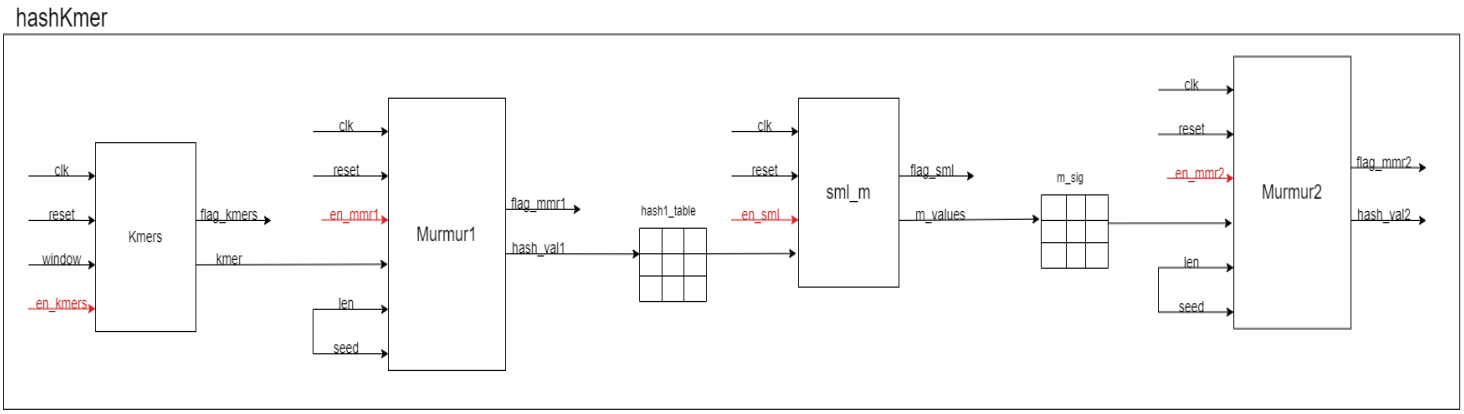


7. project Top view

Our algorithem covers several main modules:

1. Windows: divides the genome into windows of length *l*(*l*=128byte).
2. OPP_READ: calculates the reversed DNA read.
3. Max_i_count: returns the window index which has the highest count of hits in one genome.
4. Max_count: returns the window index and its genome id which has the highest count of hits in all the reference genomes.
5. The data structure of the memory is an associative array and it's built in the testbench.

6. HashKmer1,HashKmer2 modules also contain several modules:



8. HashKmers top view

i- Kmers: divide the window into kmers of length k(k=16byte) with overlapping of 8bytes.

ii- Murmur1: hash function- murmur3 is applied to each *k*-mer within a window and returns hash1_table.

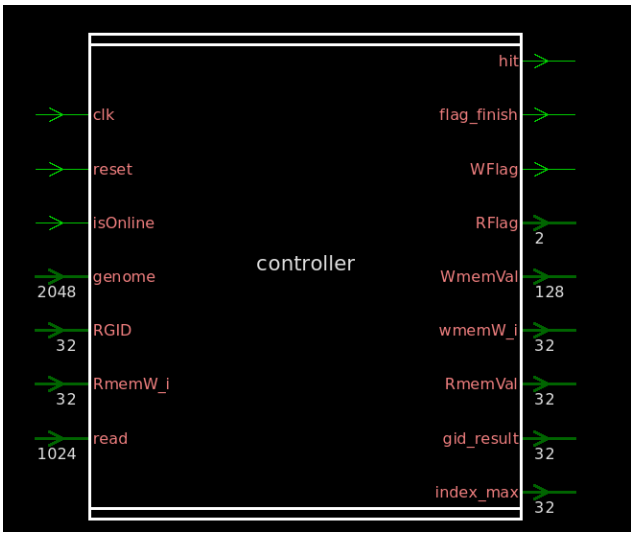
iii- Sml_m: returns the smallest values of hash1_table (S=4).

iv- Murmur2:again hash function- murmur3 is applied to each value of the hash1_table and returns hash_val2 of length 32-bit.

RTL

The controller module and all the other modules were designed in system Verilog programming language.

The interface of the controller:



9. controller module interface

Description of ports:

Input/output	Port name	Description
Input	Clk	The clock of the controller
Input	Reset	The reset of the controller
Input	isOnline	Indicates if we're in the online part (read/oppread) or the offline pare (ref)
Input	genome	The reference genome
Input	RGID	A read request to read genome id from memory
Input	RmemW_i	A read request to read window index from memory
Input	read	The input read we want to classify
output	hit	Indicates that we found candidate pairs
output	Flag_finish	Flag: the controller is done
output	Wflag	Send flag to test_bench that hash2Val is ready to be added to sigTable-write in memory
output	Rflag	flag to read from memory

output	WmemVal	A write request to write val in memory
output	WmemW_i	A write request to write window index in memory
output	RmemVal	A read request to read key val from memory
output	Gid_result	gid of the matching genome
output	Index_max	the final result- the matching window index

Interfaces of the modules:

Windows module: with each rise of the en (signal from the controller) we receive the window and its index as an output.

10. Windows module interface

OPP_READ module: it calculates the reversed read by replacing the letters: $A \leftrightarrow T, C \leftrightarrow G$ in the original read and outputs opp_read.

11. OPP_READ module interface

Max_i_count module: returns the max value in myTable which contains the windows counters of one genome - the window index which has the highest count of hits in one genome.

12. MAX1 module interface

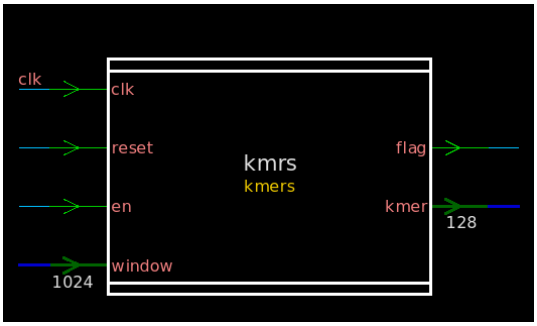
Max_count module: returns the max value in myTable which contains max window counter of each genome and returns the max between all of them.

13. MAX2 module interface

HashKmer module: divides each window into kmers of length 16bytes. A hash function-mumur is applied to each kmer within a window. The *s* smallest values of the resulting hash values are selected. All s values are inserted into a hash table using murmur hash function again.

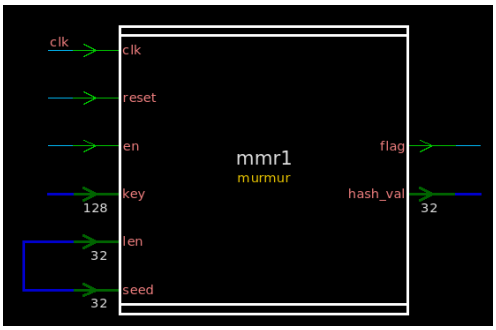
14. hashKmers module interface

Kmers module:



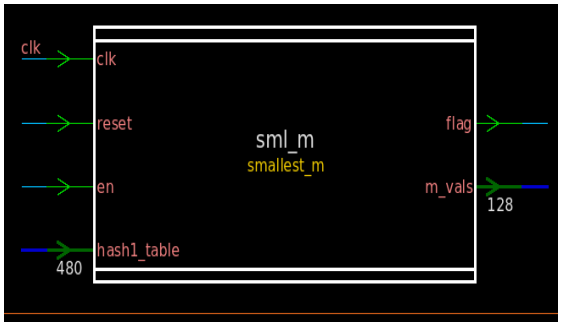
16. Kmers module interface

Murmur1 module:



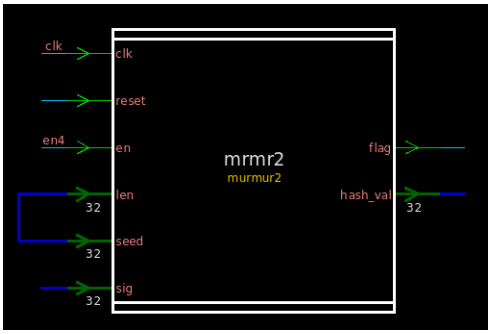
15. murmur module interface

sml_m module:



18. smallest_m module interface

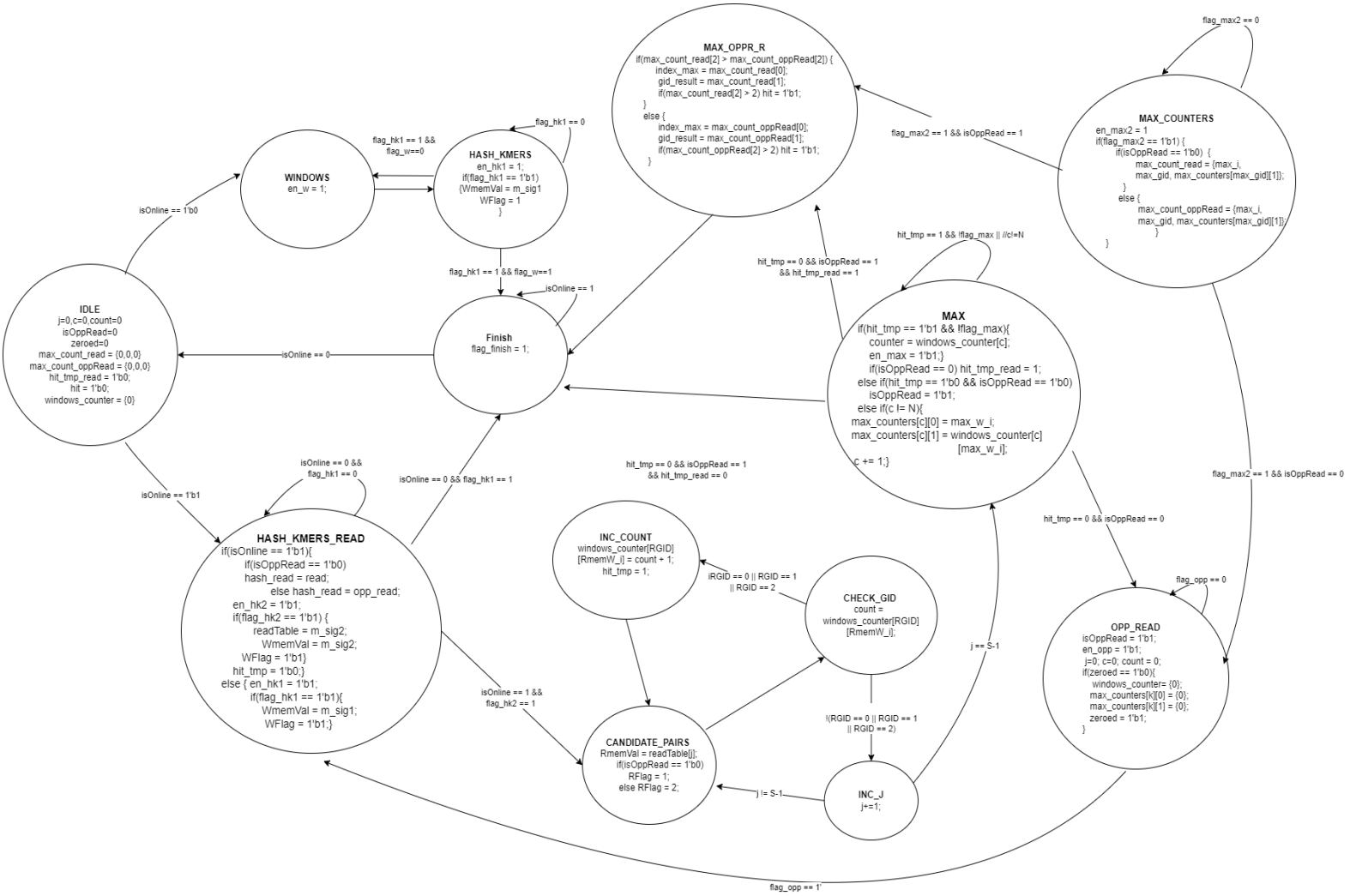
Murmur2 module:



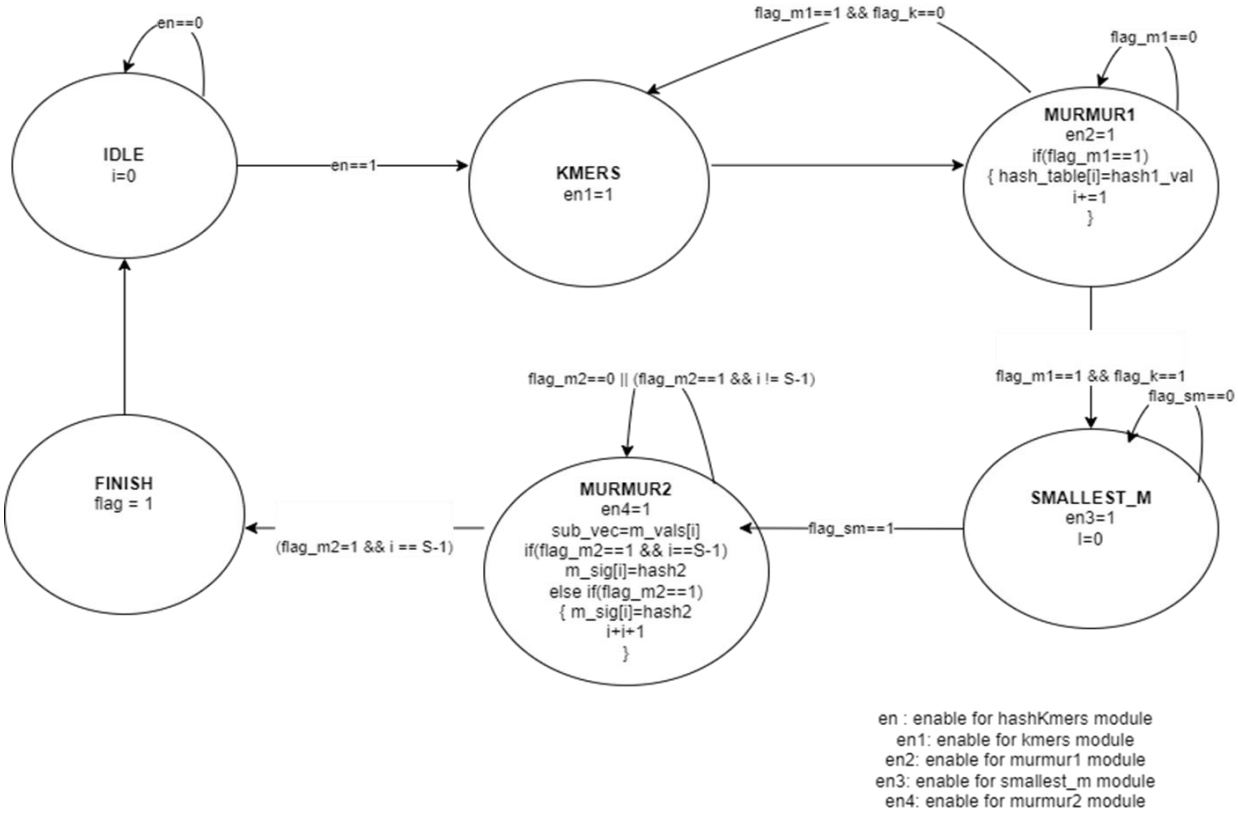
17. murmur2 interface

FSM

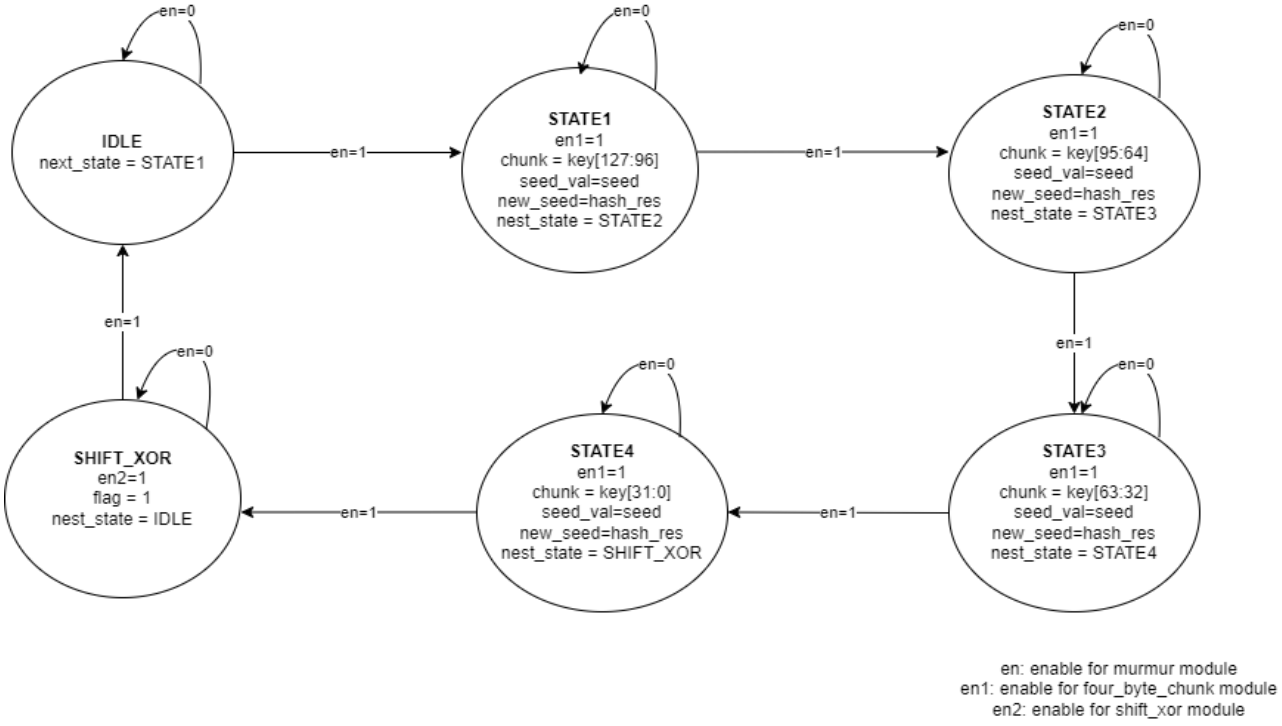
Cosntroller:



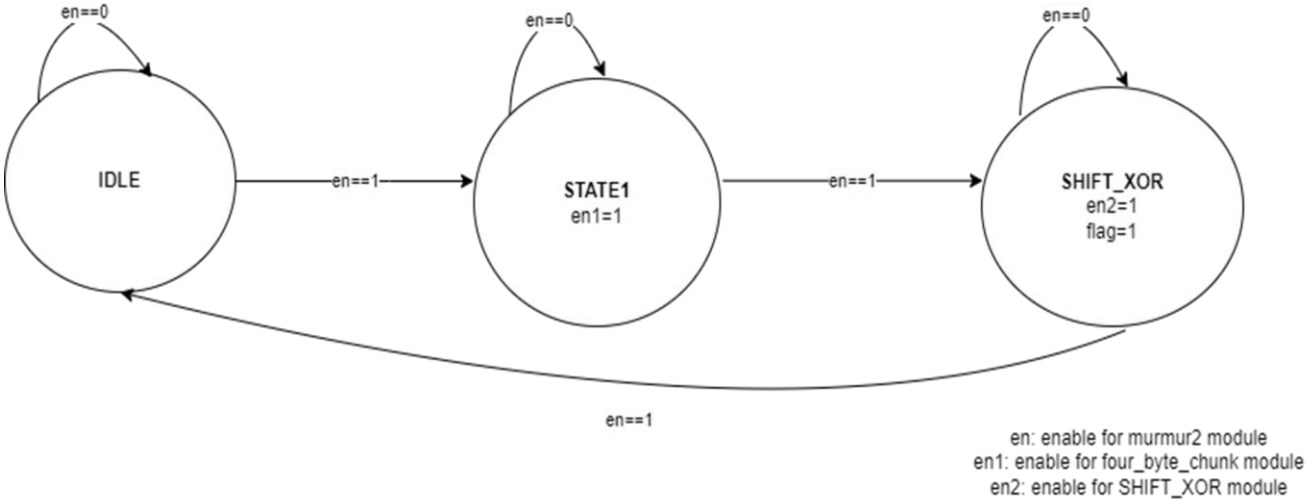
HashKmer:



Murmur3 for 128bit input:

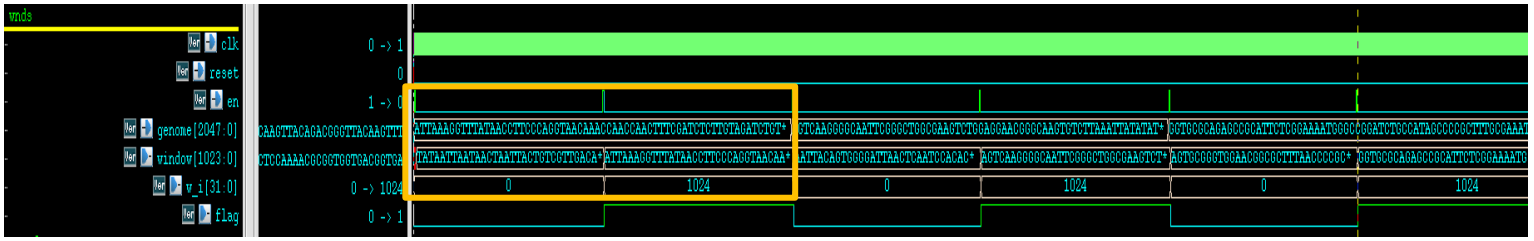


Murmur for 32bit input:



Simulation

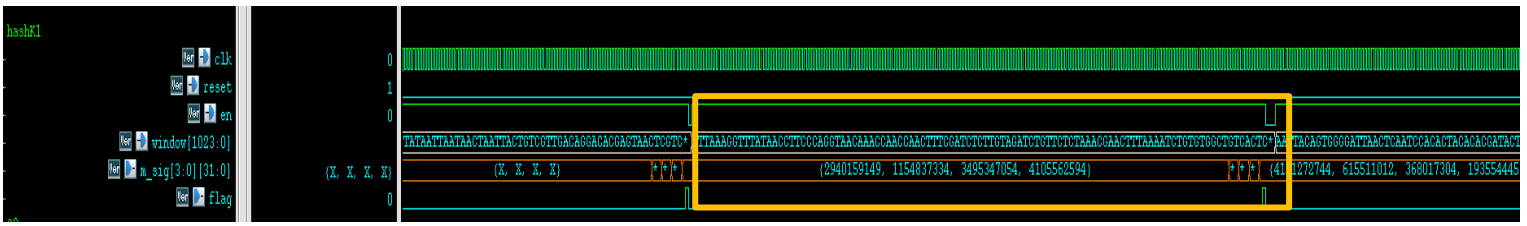
Simulation1: windows division



19. windows division simulation waveform

We can see that when the en rises to 1 we receive one window. The algorithm starts dividing the genome from the right then applying shift to the left to calculate the next window.
In the waveform above we have genome that contains two windows, that's why we have two indexes in w_i, w_i=0 indicates the first window and w_i=1024 indicates the second window because each window is 128byte → $128 \cdot 8 = 1024$.

Simulation2: hashkmer



20. hashkmers simulation waveform

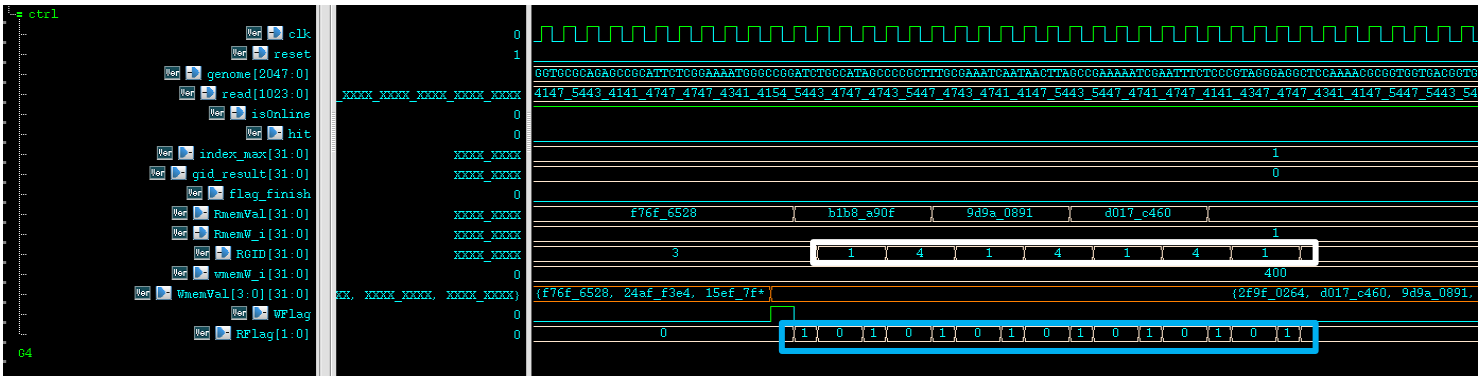
While en=1 we apply the operations that were explained before on one window: dividing the window into 16 kmers => applying murum function => smallest 4 values => applying murmur function again. In the output m_sig we can see 4 values for each window. And each time we finish working on one window, the flag rises to 1.

Simulation3: controller: the test has 3 different reference genomes (RGID 0:2) and 5 different reads (RGID 3:7)

21. controller simulation waveform

- Yellow boxes: while isOnline = 0, we map the reference genomes to the memory in the test bench, we can see 6 writes to memory, each write is a list of 4 hash values for each window.

Zoom in to the green circle:

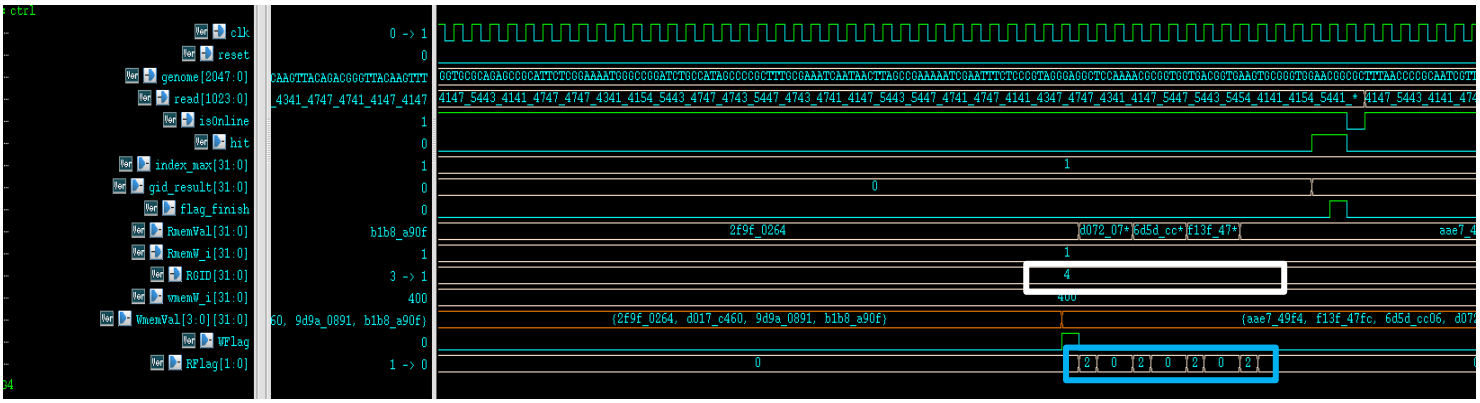


22. controller simulation waveform, zoom in 1

As we can see in the waveform above, after WFlag rises to 1, which indicates that we finished mapping the read to memory, we start finding hits with hash values of the read, so every time RFlag rises to 1 we read one pair of (w_i, gid) from the list that belongs to the hash value which is sent in RmemVal.

In this example, because the read is an exact window of the genome G1, we will get hit for every hash value, that's why we have 8 read cases (RFlag = 1), for 4 hash values, in each list the first pair belongs to the genome (gid = 1) and the second pair belongs to the read (gid = 4).

Zoom in to the purple circle:



23. controller simulation waveform, zoom in 2

As explained before, in this waveform we applied the same process, but we read from the memory when RFlag rises to 2, because we are in opp_read case.
In this example, because there are no hits, we can see 4 read cases, and gid is always 4.

The screen result of the test:

```
srcTBInvokeSim
Verdi>run
read GID = 3 -> found match at window index =          1 and in genome id =          0
read GID = 4 ->found match at window index =          1 and in genome id =          1
read GID = 5 -> no match!
read GID = 6 -> found match at window index =          1 and in genome id =          1
read GID = 7 -> found match at window index =          0 and in genome id =          2
$finish called from file "test_bench.sv", line 272.
$finish at simulation time              366450
Simulation complete, time is 366450.
test_bench.sv, 272 :                                                          #100$finish;
```

24. simulation output

The results matched our expectations.

Runtime

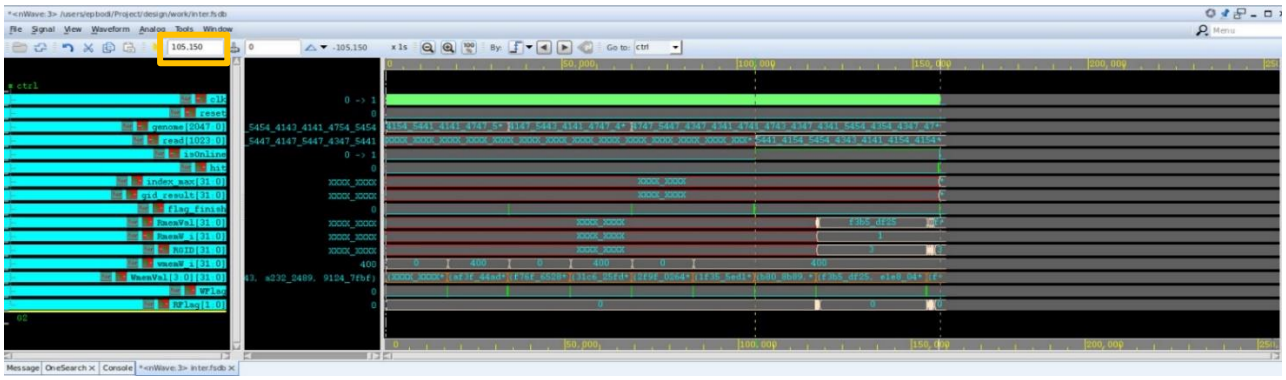
Calculation of simulation runtime to classify one read:

```
clock CLK (rise edge)                          10.00      10.00
clock network delay (ideal)                     0.00      10.00
opp_inst/opp_read_reg_747_/CP (dfnrn1)          0.00      10.00 r
library setup time                             -0.20       9.80
data required time                                         9.80

data required time                                         9.80
data arrival time                                         -9.78

slack (MET)                                                0.02
```

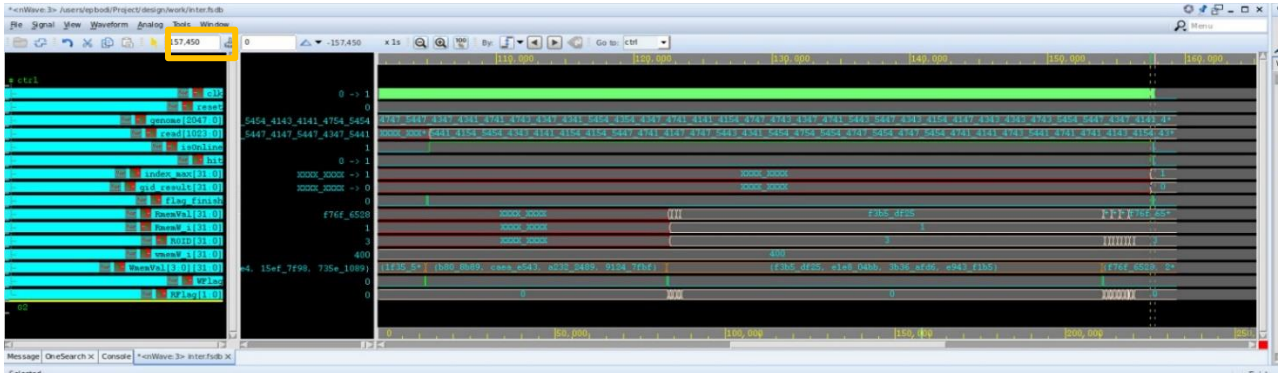
→ T=9.8ns

25. time report

From the waveform we calculated the cycles we need to classify one read, from when isOnline rises to 1 until hit rises to 1.



26. start time of classifying one read



27. end time of classifying one read

$$number\ of\ cycles = \frac{cycles}{one\ cycle} = 523$$

$$Time = number\ of\ cycles \cdot T = 523 \cdot 9.8nsec = 5.12 \cdot 10^{-6} sec$$

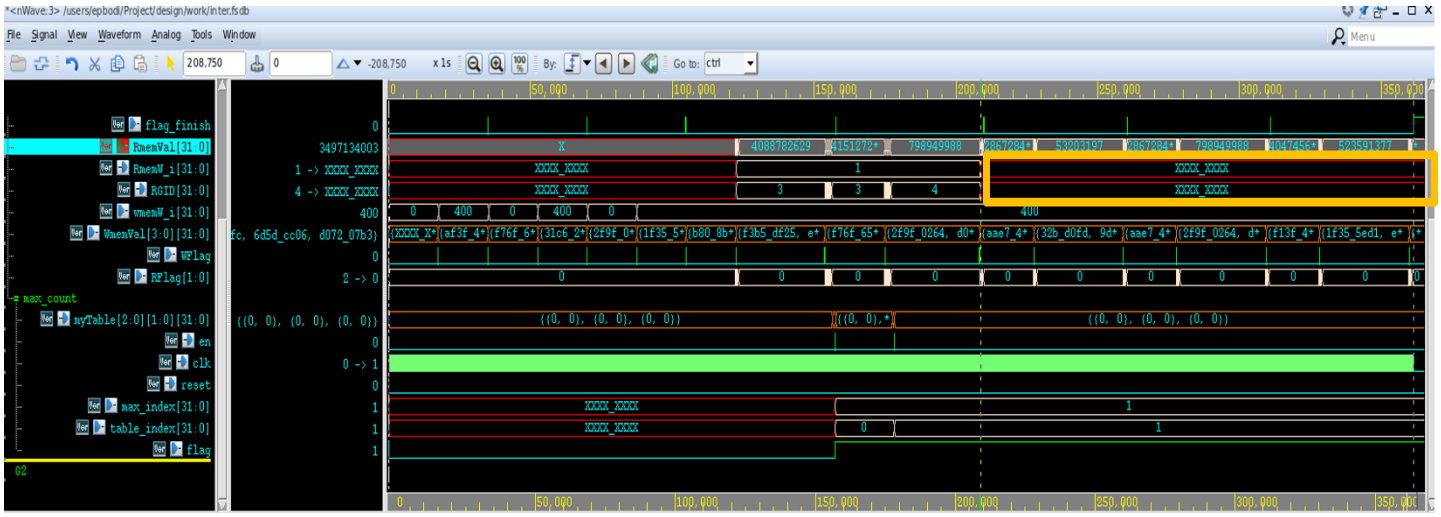
Comparison between Runtimes:

Algorithm	**Runtime (mrpm-million reads per minute)**
Our python simulation	0.15mrpm
Our hardware simulation	11.7mrpm
Kraken	1.4mrpm
MetaCache	1.3mrpm
CLARK	1mrpm

As we can see from the schedule above, we succeeded in building the accelerator with an impressive result, we gained 78x speedup over our software tool, 8x over Kraken, 9x over MetaCache and 11x over CLARK.

Problems, Challenges & Solutions

As it can be seen in the figure below, the output signals: RmemW_i[31:0] and RGID[31:0] return XXXX_XXXX at the end of the simulation which is wrong, it should remain a valid value like the others signals**.**



28. waveform of the error: invalid output

By running tests, the error was found in the test bench. We had to change the if condition marked in the code below to solve the problem:

Before:

```
always@(posedge clk) begin
    if(RmemFlag != 2'b0) begin
        w_i0 = (memory[val][t].w_index)/1024;
        G_id0 = memory[val][t].g_id;
        t+=1;
        if(t == memory[val].size()) t=0;
    end
    else begin
        w_i0 = w_i0;
        G_id0 = G_id0;
    end
end
```

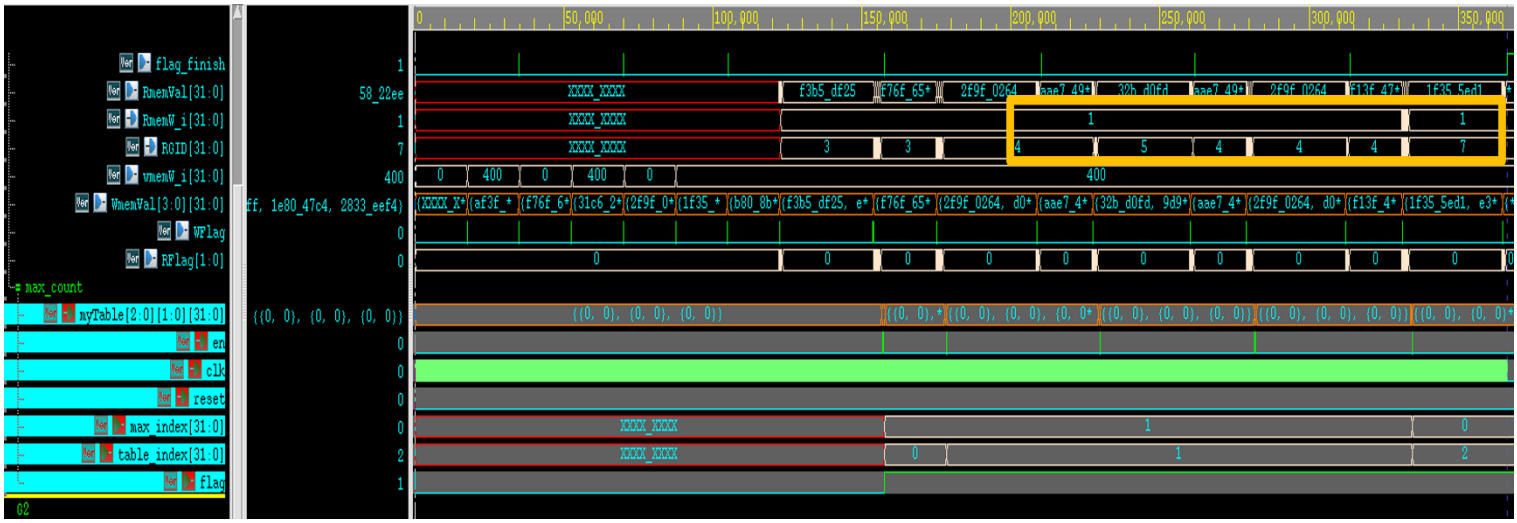
29. code with error

after:

```
always@(posedge clk) begin
    if(RmemFlag != 2'b0) begin
        w_i0 = (memory[val][t].w_index)/1024;
        G_id0 = memory[val][t].g_id;
        t+=1;
        if(G_id0 != 0 && G_id0 != 1 && G_id0 != 2) t=0;
    end
    else begin
        w_i0 = w_i0;
        G_id0 = G_id0;
    end
end
```

30. corrected code

After solving the problem:

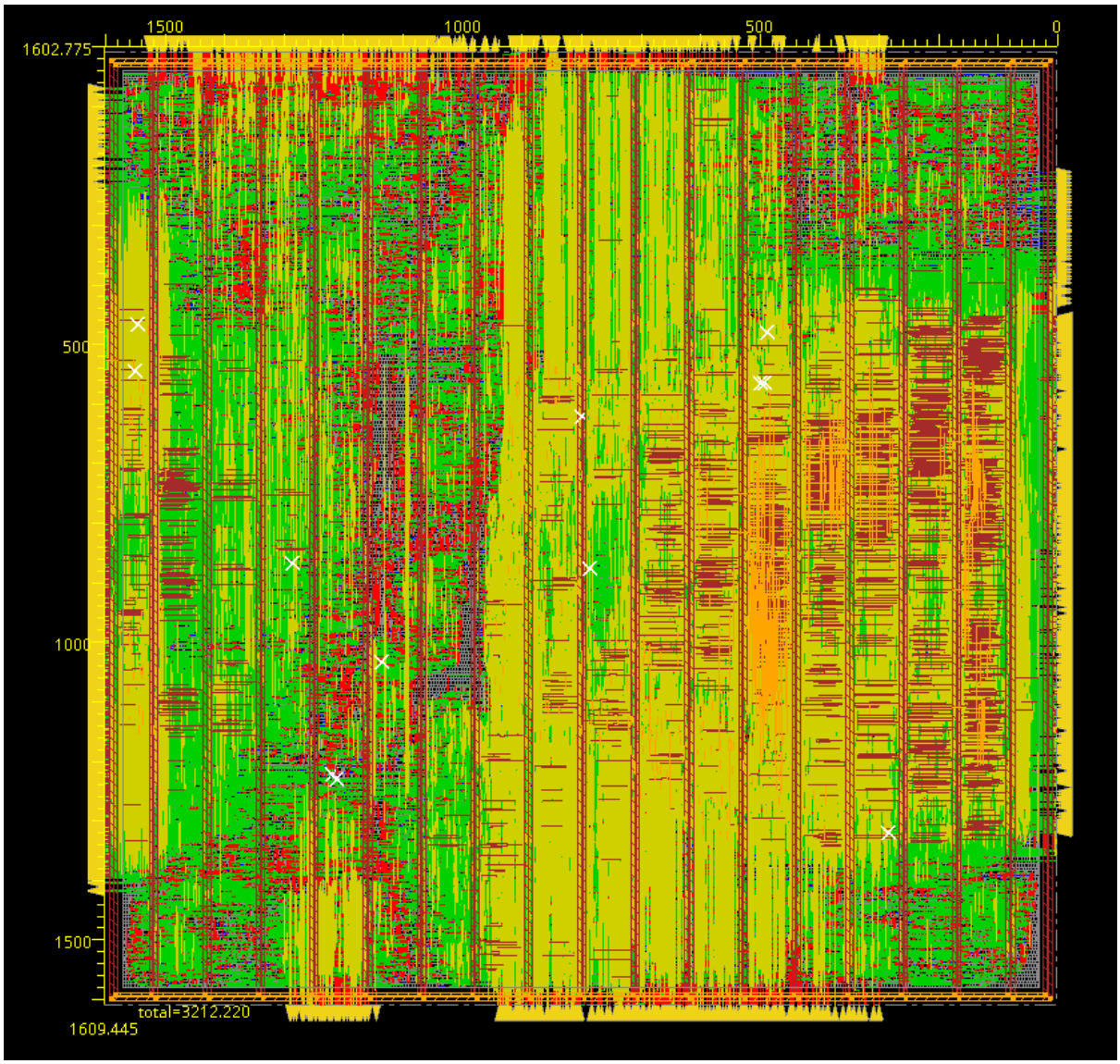


31. simulation waveform after solving the problem

Synthesis and Layout

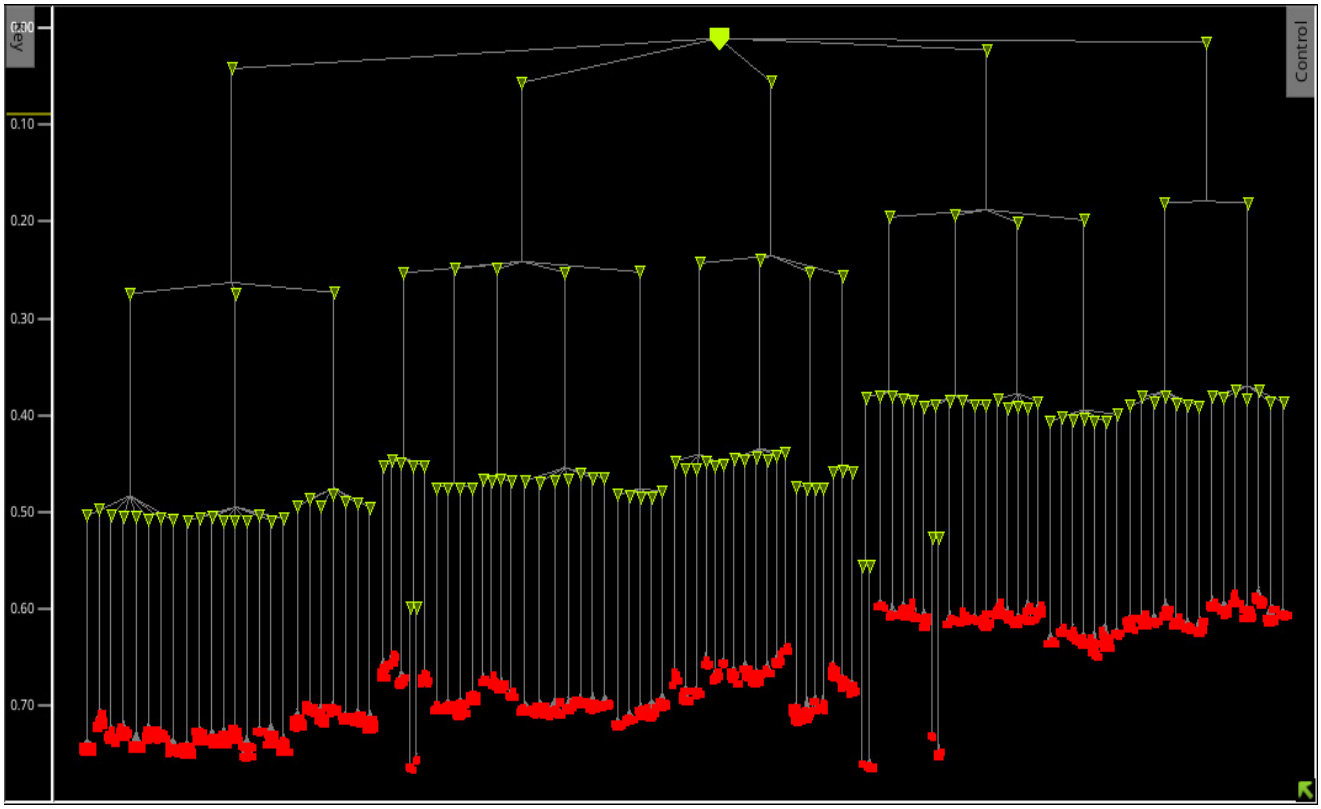
The synthesis and layout processes were performed using the tools Synopsys and Cadence (Design Vision and Innovus). The layout result was constructed using the Innovus software (Cadence) with the Tower 0.18u design kit.

The layout result: the chip and its dimensions 1602.775um:1609.445um:



32. chip layout

Here we can see the clock tree synthesis of our design which is a technique for distributing the clock equally among all sequential parts of a VLSI design. The purpose of Clock Tree Synthesis is to reduce skew and delay.



33. clock tree synthesis

Summary and Conclusions

In our project, we have developed a classification scheme that is finely tuned to classify reads considering the reverse read based on the number of matching k-mers they have with a reference genome window of a similar size to the read. This approach has proven effective and has allowed us to optimize various parameters to create a hardware accelerator for LSH MinHash.

The pre-work research focuses on formulating the algorithm, based on the MinHash algorithm using Murmur hash function. Also utilizing software to investigate parameters crucial for understanding complex systems and phenomena, specifically in the context of genomic classification.
The optimal parameters we have determined through our software search are as follows:

1. l (length of window): 128 bytes
2. k (k-mer size): 16 bytes
3. s (number of smallest hash values): 4
4. OL (overlap length): 8 bytes

In the hardware part, we leveraged the optimal parameters identified in the software research. This involved implementing the hardware accelerator in System Verilog. To ensure its functionality, we constructed a comprehensive test bench, utilizing waveforms for debugging purposes and calculating runtime. Additionally, we performed synthesis and layout processes to further refine and optimize the hardware accelerator's design.

Through extensive testing and optimization, we have achieved a remarkable improvement in the hardware implementation of our LSH MinHash accelerator. Specifically, we have achieved a significant acceleration, with a throughput of 11.7 million reads per minute (mrpm), compared to the initial software-based search approach which only managed 0.15mrpm.

To summarize, we gained 78x speedup over our software tool, 8x over Kraken, 9x over MetaCache and 11x over CLARK. This indicates the efficiency and effectiveness of our hardware accelerator in speeding up the process of read classification.

Future work

For future work and expansion of this project, we have identified several key areas:

1. Memory Utilization: One of our primary objectives is to leverage Random Access Memory (RAM) in the test bench instead of building memory structures for the hardware accelerator. This change would not only optimize the use of resources but also improve the scalability and efficiency of our hardware solution.
2. Scaling to Larger Databases: Our current implementation has been tested with a limited number of reference genomes and relatively small datasets. To further enhance the practical applicability of our system, we intend to scale up by incorporating larger reference genomes and accommodating more than three reference genomes in the classification process. This expansion will be crucial for handling more diverse and extensive genetic data.

This ongoing work reflects our commitment to advancing the field of genomic sequence classification and hardware acceleration. By implementing these future goals, we aim to make our system even more robust and capable of handling the demands of increasingly large and complex genomic datasets, which are crucial for various applications in genomics and bioinformatics.

References

[1] MetaCache: context-aware classification of metagenomic reads using minhashing, https://academic.oup.com/bioinformatics/article/33/23/3740/4083578#394530984

[2] EDAM: edit distance tolerant approximate matching content addressable memory, https://dl-acm-org.ezlibrary.technion.ac.il/doi/abs/10.1145/3470496.3527424

[3] MurmurHash, https://en.wikipedia.org/wiki/MurmurHash

[4] Minhash, https://en.wikipedia.org/wiki/MinHash

[5] A Computer Scientist's Dictionary for Genomics, https://www.bx.psu.edu/old/courses/bx-fall08/definitions.html

[6] National Library of Medicine, https://www.ncbi.nlm.nih.gov/datasets/taxonomy/454194/

GitHub link for the project

https://github.com/BoranSwaid/LSH-minhash-accelerator-project/tree/main