UNIVERSITY OF CALIFORNIA
SANTA CRUZ

HARDWARE ACCELERATOR FOR MINIMAP-2 KERNEL

A thesis submitted in partial satisfaction of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Kartik Rajesh Jain

December 2020

The Thesis of Kartik Rajesh Jain
is approved:

Professor Heiner Litz, Chair

Professor Jose Renau

Professor Scott Beamer

Quentin Williams
Interim Vice Provost and Dean of Graduate Studies

Table of Contents

List of Figures

List of Tables

Abstract

Hardware Accelerator for Minimap-2 Kernel

by

Kartik Rajesh Jain

Translating the computation-critical part of an application to special-purpose hardware is a standard practice to achieve performance gain and speed-up in execution time. In this thesis, we chose a gene sequence aligner tool: Minimap2 as a workload. This thesis implements the kernel of Minimap2 in hardware using Verilog and parameterizes the design to support parallelism. We subsequently evaluate the design's performance on timing, speed-up, and area utilization.

Acknowledgments

I would like to thank my teachers for their guidance and feedback. I also thank my family and friends for their constant support and encouragement.

Chapter 1

Introduction

General-purpose microprocessor systems are often not suitable for meeting the demands of many compute-heavy applications. Modern workloads require a large amount of compute resources and data movement, which limits performance. One solution is to offload the processing of such applications to dedicated hardware working jointly with the general-purpose processors. Hardware accelerators are the best example of such compute logic that utilizes the inherent parallelism in workloads to provide speed-ups, resulting in lower power consumption and higher performance compared to the traditional approach. Figure 1.1 shows how a compute intensive part of the application is transferred to a hardware accelerator. This thesis explores the design and verification of an FPGA-based accelerator specialized for a genomics workload –

Minimap2. We implement Minimap2 in Verilog and perform experiments to observe the gains achieved by offloading the application to a Field Programmable Gate Array (FPGA).

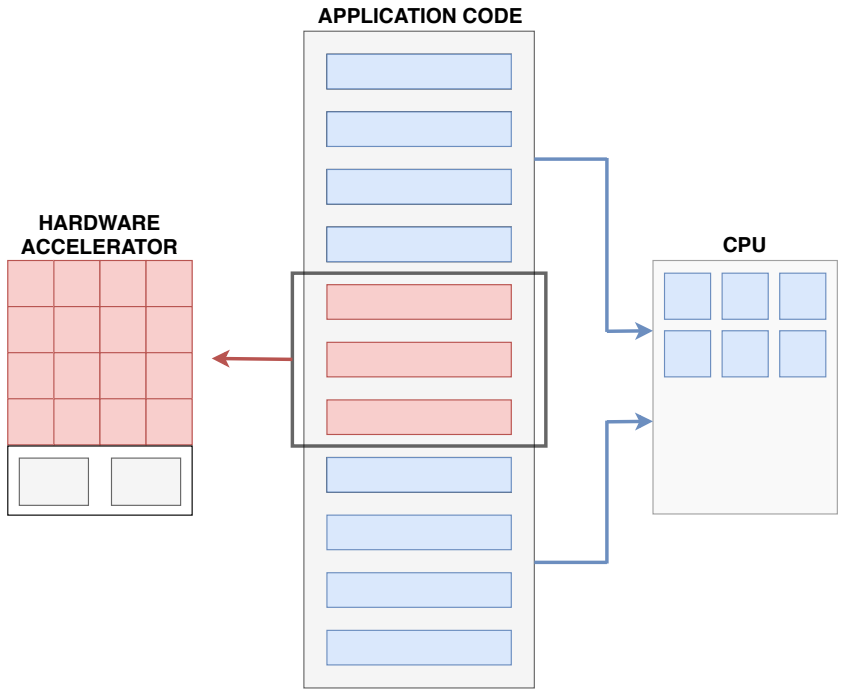

Figure 1.1: Offloading compute heavy part to a hardware accelerator

The field of genomics has expanded by leaps and bounds in the past decade and has been at the heart of promising scientific discoveries in bioinformatics. Genome research involves dealing with huge volumes of data, which can take hours or days to process, even on state-of-the-art systems. With several genomics databases being developed and the low cost of sequencers, data volumes

are expected to reach exabyte scales by 2025. Thus, genome workloads stand to benefit immensely from acceleration [1], which can help researchers draw valuable insights. Popular applications have already given rise to domain-specific accelerators such as Darwin[1], GenAx[2] to name a few.

Minimap2 [3] is a nucleotide sequence alignment tool, which can find regions of similarity between pairs of DNA or m-RNA sequences. It compares input sequences against extensive reference databases to produce biologically meaningful outputs. Since Minimap2 is designed as a software library in C, our main task is to implement its functionality in hardware. However, transforming a software kernel to hardware is challenging, since many popular software primitives cannot be directly described using hardware description languages (HDL's). Even if we are able to represent software constructs in HDL's, they are not always mapped efficiently to hardware.

In this thesis, our first focus will be to outline challenges faced in kernel acceleration and to propose techniques to overcome them. Next, we use the techniques to build a hardware architecture for Minimap 2. We also modify the kernel to adapt it towards a hardware implementation, for example, by reducing repetitive library functions to unique design modules. Next, we verify the design by modeling testbenches and tally all results against the C code. Once verified, we synthesize the design and make some optimizations to ensure

maximum utilization of parallelism. We achieved a maximum speed-up of 22% for the optimized HDL code with minor increase in the FPGA resources utilized. The HDL code for the Minimap2 kernel will be made open source upon completion of future works.

This thesis is laid out as follows. Chapter 2 discusses kernel acceleration challenges for code conversion and verification and enumerates the proposed techniques to tackle them. Chapter 3 discusses the selection of Genomics as a workload and details the RTL design. Finally, Chapter 4 provides results and conclusion.

Chapter 2

Kernel Acceleration

In this chapter, we focus on techniques for translating software kernels to hardware. We discuss challenges faced while converting software constructs (C/C++ constructs) to HDL representations. Furthermore, we propose techniques to overcome these limitations.

2.1 Overview

Applications can be modeled by software libraries [3] or purely represented by custom-made hardware. Programming languages such as C/C++ have been extensively used as they provide rapid development, high portability, and ease of updating features. However, software constructs are not adequate to represent the process, timing, and power variations observable in physical circuits.

Moreover, they do not provide the parallelism and speed up offered by hardware implementation. The efficiency of the application increases by orders of magnitude when it is implemented in hardware directly. We use hardware description languages (HDL's) to design the application's logic in hardware. HDL semantics allow us to express concurrency in the code. Most importantly, HDL's lets us simulate the progress of the time, allowing us to model the hardware before it is physically created. Converting software to hardware is supported by several tools and has become a field of wide interest. However, such tools are very recent, and most of them have not been validated on large production designs yet. We tried Vivado High-Level Synthesis (HLS) tool to map the software code to hardware, but it was incapable of efficiently representing the SIMD instructions and the pointers used in the code. Thus, in this work, we design a custom accelerator architecture by manually writing Verilog HDL. Custom design helps us validate our design easily and gives us flexibility for further optimizations.

2.2 Code Conversion

Implementing hardware for software kernels has several challenges. Few software constructs can be accurately represented in hardware. Constructs such as if-else conditions and case statements map easily in hardware as Verilog pro-

vides semantics to represent them. But, the biggest problem is faced while converting software language constructs like loops, dynamic memory allocation, pointers, since it is challenging to synthesize them efficiently.
This section discusses the challenges involved with these constructs and provides solutions to synthesize them into hardware efficiently.

2.2.1 Pointers

In software, pointers represent the address of an element in memory; they are used to pass parameters by reference, access array elements, or address dynamically allocated memory. They are an important construct of software languages that allow direct access and management of memory. In order to translate a program involving pointers into HDL code, it is necessary to have information about what each pointer points to. The hardware design should consist of appropriate logic to access and modify the data referenced by pointers. We dereference the pointers used in the kernel code and map the data stored in each location it points to register arrays or memory blocks. Now, this data can be accessed at any time of the execution. Manually implementing pointers in hardware gives designers control over the locality of the data used. An example in Figure 2.1 explains how it is done.

```
1 int *ptr[5];
2 int i;
3 for (i = 0; i < 5; ++i)
4     ptr[i] = i;
```

Figure 2.1: C code for pointers

In this example 'ptr' is an integer pointer pointing to an array of size five. The code is filling the integer array with some value. While converting this code to HDL we need to keep the size and the type of the array in mind. The above-mentioned code will be converted to HDL as described in Figure 2.2.

Here we have used a 2-D register array to represent the pointer array. The size of each element needs to be equal to that of the size of the pointer type which is an integer in this case and the length of the array should be equal to the size of the pointer array. If the size of the array is huge it is better to represent it in a memory block.

```
1 reg [31:0] ptr [4:0];
2 assign ptr[0] = 32'd0;
3 assign ptr[1] = 32'd1;
4 assign ptr[2] = 32'd2;
5 assign ptr[3] = 32'd3;
6 assign ptr[4] = 32'd4;
```

Figure 2.2: Verilog code for pointers

2.2.2 Dynamic Memory Allocation

Dynamic Memory Allocation is a software technique enabling dynamic memory management. C provides various inbuilt library functions like malloc, calloc and realloc to allocate and deallocate memory from the heap dynamically. The malloc function dynamically allocates a single large block of memory by taking data size as a parameter and returns a void pointer pointing to the block's first location.

ptr = (cast-type*) malloc(byte-size)

The calloc function is very similar to the malloc function, but it also initializes each memory block to value zero. realloc function deallocates the previously allocated memory block and reallocates it with the new size specified.

ptr = realloc(ptr, new-size);

C/C++ has a dedicated memory allocator which performs the task of allocating and deallocating memory in a heap, and it also manages the free list in the memory. Supporting dynamic memory management in hardware is complex and unfeasible for most hardware designs. Furthermore, the lack of virtual memory limits the physical memory resources, and hence every hardware design is intrinsically bounded in the amount of memory it can use. We manually represent memory using register arrays or memory blocks (block RAMs). The problem with dynamic memory allocation is the size of the memory required for an application is unknown. The size depends on the parameters used by the dynamic memory allocation functions. We can determine the maximum value of the functions' parameters during compile time to estimate the size of memory utilized. Then we can initialize a register array or block memory accordingly to map the memory of the same size used by the code.

Let us consider the code in Figure 2.3. At first, a single block of memory of size n * sizeof(int) is dynamically allocated using the 'malloc' function. 'malloc' returns a pointer pointing at the first address of this memory block. In line '6', the pointer is reallocated using the function 'realloc' to a size of n * 2 * sizeof(int). To translate the following code to HDL, we need to know the maximum possible value of n can take in this code. After analyzing the code

```c
int * ptr;
int n, i;
//...
ptr = (int*)malloc(n * sizeof(int));
//...
ptr = (int*)realloc(ptr, n * 2 * sizeof(int));
```

Figure 2.3: C code for dynamic memory allocation

and learning the greatest value of 'n', we can get an estimate about the size of memory required in hardware to execute this code. Supposing the maximum value of n to be 10, we infer from the code that we will need a block of size 20 * sizeof(int). When integrating the FPGA kernel into the host program, the host program is extended with exception handling routines to handle the case where the host allocates memory greater than reserved in hardware. In this case, the kernel cannot be offloaded to the FPGA and needs to be executed in software.

```verilog
reg [31:0] ptr [19:0];
```

Figure 2.4: Verilog code for dynamic memory allocation

We represent memory block as a 2-D register array of size 32 X 20 as shown in

Figure 2.4.

2.2.3 Functions

A software function is a code set that performs a specific task and is packaged as a unit. This unit can now be used in the program wherever this particular task needs to be performed. This feature makes the code modular, reduces redundancy, and makes the code easy to understand and maintain. Functions can be defined in the application source code or may be directly accessed from inbuilt libraries. To translate functions into RTL, the designer must know what parameters the function takes, what computations occur within them, and what output is produced. The Verilog language allows designers to use functions in a similar way as software languages. Verilog functions are synthesizable and generate combinational circuits, although a function can only have a single output. A way around this constraint is to write Verilog modules performing the same task as a function. Verilog modules can have multiple outputs which can be then linked in a separate top module.

The C code in Figure 2.5 takes two 128 bits wide vectors as inputs and performs a Single Instruction Multiple Data (SIMD) operation on them to produce another 128-bit vector output. The set of SIMD instructions is also referred to as SSE (Streaming SIMD Extensions) instructions. To translate this code we need to

```
include <emmintrin.h>
void main(){
__m128i a, b, c;
// variable assignment
a = _mm_or_si128 (b, c);
}
```

Figure 2.5: C code for function

understand how the used SIMD function works. The function _mm_or_si128 is described in the library emmintrin.h and is used for computing bitwise OR.

The function written in C is translated into a Verilog module, described in Figure 2.6 which takes two 128 bits inputs, performs bitwise OR on them, and returns a 128-bit output. This module now can be called anywhere in the main HDL code to perform this operation. Similarly, we can write a Verilog function in place of the module and both will perform the same.

```
1 module _mm_or_si128 (input [127:0] a,
2                       input [127:0] b,
3                       output [127:0] dst);
4 assign dst = a | b;
5 endmodule
```

Figure 2.6: Verilog code for functions

2.2.4 Loops

Loops are a programming construct that helps execute a sequence of instructions multiple times until a certain condition is met. Generally, loop iterations take up most of the execution time, so their hardware translation has higher chances of providing a significant speedup. Moreover, there are various optimization techniques to speed loop execution time. Verilog gives designers the option of using *generate for* to represent loops into hardware. Generate statements create several instances by replicating the logic of the body of the loop. Figure 2.7 shows how generate-for is represented in hardware.

Just like software for loops, *generate for* loops also requires an entry statement:

for (initialization expression; test expression; update expression)

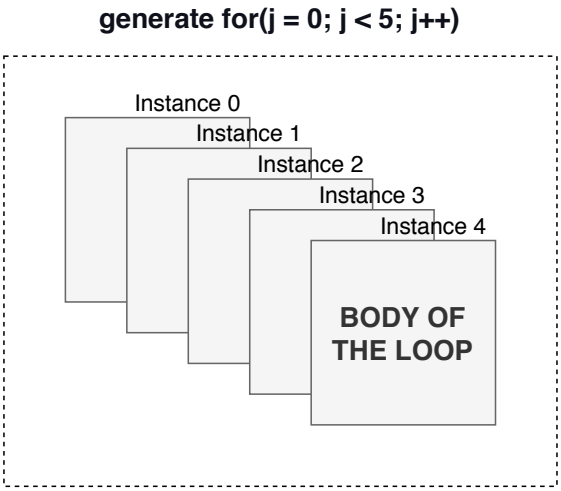

Figure 2.7: Testing a Circuit

The loop boundaries must be defined for generate loops; a dynamic variable cannot control them. Kernel loops can be optimized to gain a speedup on the process. There are two important techniques for loop optimization 1. Loop unrolling, and 2. Pipelining explained below:

1. Loop unrolling

Loop unrolling is a technique to unwind the loop, enabling instructions to execute in parallel. This method provides higher throughput by supporting concurrency hence accelerating the process.

```
1 for(j = 0; j < 3; j++)
2 begin
3     A[j] = B[i] + (C[j]*D[j]);
4 end
```

Figure 2.8: Verilog code for for loop

```
1 assign A[0] = B[0] +  4*D[0];
2 assign A[1] = B[1] +  8*D[1];
3 assign A[2] = B[2] + 16*D[2];
4 assign A[3] = B[3] + 32*D[3];
```

Figure 2.9: Verilog code for loop unrolling

In Figure 2.8 the for loop is not unrolled and 'C' is a constant array holding values {4, 8, 16, 32}. Figure 2.9 shows the modified code where the loop is unrolled. Unrolling loop enables the series of assignment statements to run in parallel. The concurrent computation reduces the execution time of the design, hence providing us with a speed-up.

2. Pipelining:

Loops in software languages are sequential, which means the next iteration of the loop begins executing only when the previous iteration is completed.

Loop pipelining in hardware allows the instructions in the loop to run concurrently. This technique enables instruction-level parallelism thus increasing the throughput and accelerating the execution process. Figure 2.10 explains how loop pipelining works.

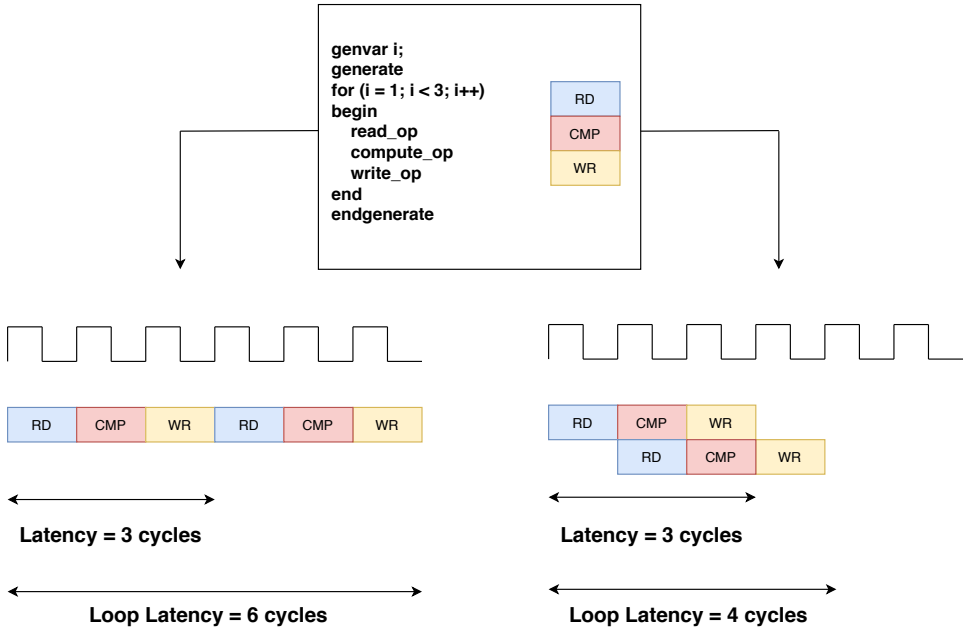

Figure 2.10: Pipelining

As you can see in Figure 2.10, without pipelining it takes 6 cycles for the loop to completely execute, while it takes only 4 cycles when the loop is pipelined. We can apply the same technique to accelerate codes consisting of multiple loops that execute one after another. We can pipeline these loops in a similar fashion to run concurrently.

There are a few limiting factors to these techniques:

1. Data dependency: If the data between subsequent iterations are dependent, the execution can not proceed until the previous iteration is computed and updated. Data dependency nullifies the performance improvement provided by loop unrolling and pipelining.

2. Limited hardware resources: Unrolling loops creates copies of the loop body in order to exploit parallelism and provide high throughput. If the unrolling factor i.e., the number of times a loop is unrolled, is high, there can be a shortage of hardware resources limiting the parallelization of code.

Chapter 3

Genomics Workload

In this chapter, we explain why we selected Genomics as a workload for our research and discuss the Minimap2 algorithm. This chapter also covers an overview of how to analyze the software kernel before translating it into hardware. Lastly, we present how the Minimap2 code is translated into hardware and describe the optimization done on the design.

3.1 Why Genomics

Genomics is an interdisciplinary field of biology that concerns the sequencing and analysis of an organism's genome. It is an indispensable part of medical research and has various practical applications such as disease prevention, pharmaceutical development, and criminal forensics.

Gene sequencing is a vital component of genomics studies. It is the study of determining the exact order of the bases in a strand of DNA. With the help of gene sequencing, we can better understand the source of various diseases ranging from cancer [4], Alzheimer's [5] to rare genetic disorders [6].

Sequence alignment is another important study in the field of genomics. It involves arranging DNA or RNA sequences to identify regions of similarity. The similarities between the sequences suggest functional or evolutionary relationships.

The extraordinary rate in improvements in genomics has led to generating an increasing amount of genomic data. The exponential growth [7] in the genomic databases has caused the utilization of substantial computational resources. Processing data at such huge scales require platforms that can accurately represent sophisticated algorithms used by gene computing tools and save us time and resources. FPGAs very well suit this description; they provide an opportunity to parallelize the algorithm that helps in improving computational time, and they also reduce area and power consumption. In this thesis, we convert the software kernel of Minimap2 into hardware and parallelize it to achieve speedups.

3.2 Minimap2 Algorithm

Before discussing the algorithm listed below are some common terminology used in this section:

1. Read: Read refers to the sequence of base pairs of the input genome sequence.
2. Seed: Seed is a subset of the read sequence.
3. Anchor: A short match of DNA sequence between the two reads.
4. Chain: A list of anchors that have the same position in both the reads.
5. Overlap: A sequence match between two reads.

Minimap 2 [3] is a multi-purpose gene sequence alignment tool that can map DNA and long mRNAs against an extensive reference database. It follows a typical seed-chain-align procedure as most full-genome aligners [8]. The first step of the Minimap2 algorithm involves collecting short seeds from the reference sequences and indexing them in a hash table. With the help of this hash table, reads that share a large number of seeds are selected. Then the Minimap2 tool performs a chaining algorithm on these selected reads to compare seed locations in these reads. The chaining algorithm performs dynamic programming to group together seeds which have a relatively equal distance from each

other. The similarity in the sequence shows that the selected seeds are related. The last stage is the base level alignment, where the tool extends the matches of seed and fills the gaps between adjacent reads in the chains with approximate matches. Figure 3.1 represents the working of the algorithm pictorially.

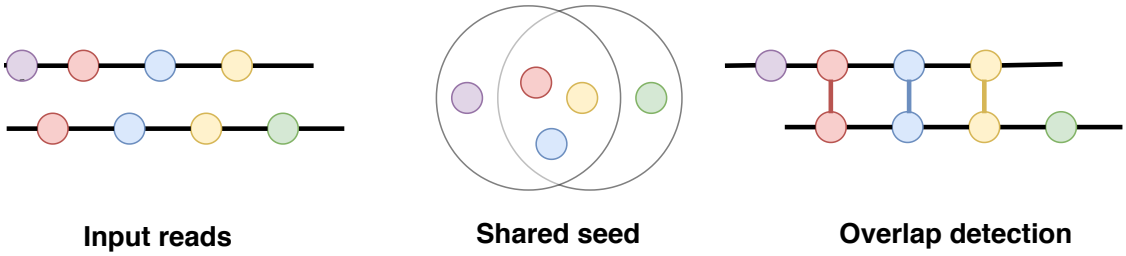

Figure 3.1: Minimap2 algorithm

Minimap2 is a versatile tool that can work on long reads as well as short reads. This tool's chaining algorithm is high-speed and accurate, making the tool 3–4 times faster than the mainstream short-read mappers and 30 times faster than other long-read mappers.

3.3 Kernel Analysis

Writing hardware for software kernels enhances the performance of the workload. Analyzing the kernel before converting it into hardware helps us understand the workload better and hence will accelerate and increase the

accuracy of the conversion process. Some characteristics for kernel analysis are discussed below.

3.3.1 Data movement

Understanding the data flow in the code helps us underline data dependencies, which plays an important role in deciding the extent of the parallelization of the kernel. Once we understand the data dependencies in the code, we can implement the concurrent computation of the tasks, allowing them to overlap in their operation, thus increasing the overall throughput.

3.3.2 Scope of Parallelization

In software languages like C/C++, instructions are executed sequentially. In the absence of any directives that limit resources, the hardware provides the opportunity of running them concurrently, minimizing the latencies. There are many techniques for optimizing the logic; some of the important ones are pipelining and loop unrolling that have been discussed in detail in Section 2.4.

3.3.3 Resource Utilization

The judicious use of resources while writing HDL code is crucial for translating software kernels. It governs the area utilization of the design. Verilog

lets us control how to represent a variable in hardware, we can implement it using a register or a wire. If the variable represents a large array it is better to use block rams. Also, there are few techniques to improve resource utilization but are out of the scope of this thesis.

3.3.4 Library functions

C/C++, allow us to use predefined library functions, but there is no such feature provided by HDL languages to use C library functions directly. We need to identify such functions in the kernel code and write separate logic to represent them in hardware. One advantage of writing HDL code for library functions is that we can modify them to support parallelism.

3.4 Design Methodology

In this section, we present an overview of our hardware design and verification methodology. We use techniques described in Chapter 2 to convert the software kernel of Minimap2 into hardware. Figure 3.2 shows the steps involved in translating C code to Verilog. The first step includes analyzing the software kernel based on characteristics discussed in Section 3.3. Once we understand the workload, we identify the major C constructs used in the code and convert them individually into synthesizable hardware. Then, we test the

functionality of these hardware designs. Section 3.5.2 explains in detail the verification process of the design. We combine and connect the verified HDL codes into a single code, maintaining the kernel's data flow. The last step of the conversion process is to verify the functionality of the whole HDL code. Once the functionality is tested, we optimize the design further. We try to parallelize the design for better performance before unloading it onto the FPGA.

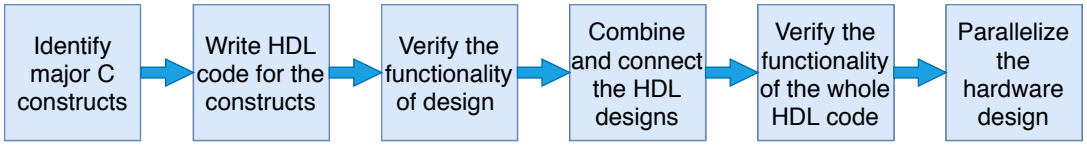

Figure 3.2: Design methodology

3.5 My Design

3.5.1 Block Diagram

Figure 3.4 shows a top-level block diagram of Minimap2 architecture. We use the design methodology explained in Section 3.4 to design the kernel of Minimap2. The Minimap2 code consists of one main loop, with three loops nested inside it. Figure 3.3 shows a high level pseudocode of the Minimap2

```
#include <emmintrin.h>  //for SIMD instructions
void main(){
//variable declaration and initialization
    for (i = 0; i < qlen + tlen; ++i){
    //inner loop boundaries are created.
        for(){/*fission loop*/}
        for(){/*core loop*/}
        for(){/*third loop*/}
    }
}
```

Figure 3.3: Pseudo code for Minimap2 kernel

kernel. The code's primary inputs are the target and reference genome sequences, lengths tlen, and qlen. This data gets stored in the register array when the reset is set high. We represent each loop by a separate Verilog module called in the top module. Each SIMD instruction is translated to the Verilog module and instantiated wherever they are used in the code. The main loop executes for qlen + tlen iterations and in every iteration generates loop boundary conditions for the three nested loops. The three loops also have data dependencies amongst them.

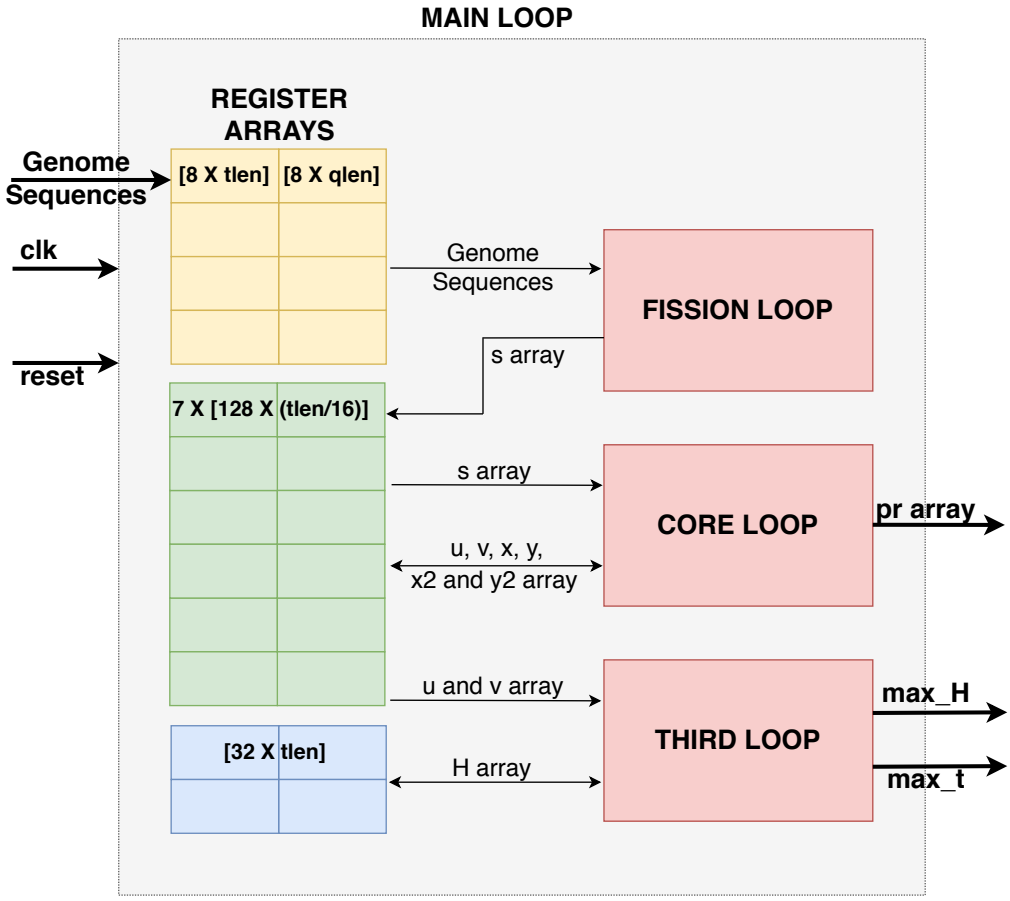

Figure 3.4: Block diagram of Minimap2 kernel

The first loop is called the fission loop, it takes genome references as inputs and performs SIMD operations on them to generate score data which is then stored in a pointer array 's'. Figure 3.5 shows a top-level block diagram of how the fission loop generates output and stores it in 's' array. *m1_*, *sc_mch*, *sc_N*, and *sc_mis* are code parameters whose values are derived from the C code.

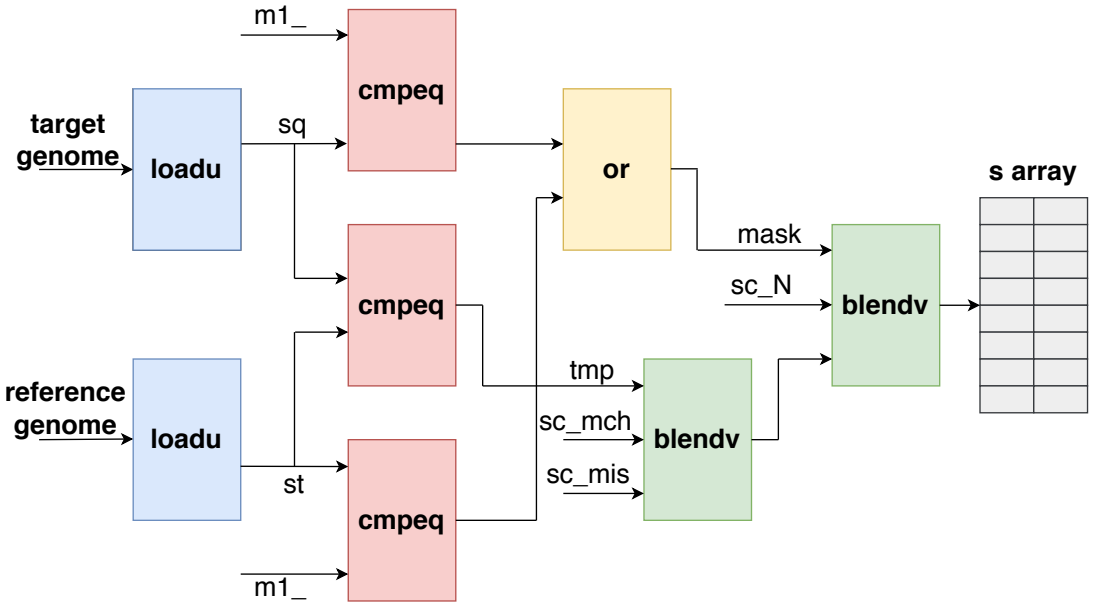

Figure 3.5: Block diagram of fission loop

The next loop is the core loop, which is the significant part of the code where the genomes' chaining occurs. The core loop generates a pointer array 'pr' as an output. The core loop in the process of computing output also stores data in 'u' and 'v' pointer arrays, which are used by the third loop. The third loop computes 'max_H' and 'max_t' values representing the number of matches between two reads. Apart from these significant constructs, there are some minor pointer and variable declarations in the HDL code.

3.5.2 Verification Integration

Design verification is one of the crucial steps performed to ensure the correctness of hardware logic. We use a bottom-up approach to test the hardware design of the Minimap2 kernel. First, we verify the individual C construct designs such as functions, pointers. Then, we test each loop module formed by merging the C construct designs. Lastly, we instantiate the loop modules in a top module and verify the whole design. The steps for testing a design remains the same at every level.

1. Identify the part of the C code that is being verified.

2. Annotate the C code so that all inputs and outputs to a C function can be captured into a file. Several input and output sets can be captured by just executing the regular C code. The captured data is then leveraged to automatically generate the Verilog unit tests.

3. Design a testbench wrapper for the Verilog code such that it provides the design with input data collected from C code. Use file handling constructs in Verilog to read the input data file.

4. Save the output data from the Verilog code into a file.

5. Finally, compare the output collected from C code and the Verilog code. If the files match, the hardware conversion is accurate.

3.5.3 Design Optimization

One significant advantage of writing hardware logic for complex software kernels is that it provides the opportunity to parallelize operations increasing the throughput by computing on multiple data concurrently. Hardware designs can leverage vector (SIMD) instructions, pipelining, and loop unrolling to implement parallel execution. This section discusses the scope of parallelization in the HDL code of the Minimap2 kernel and the measures taken to optimize it. The Minimap2 kernel consists of one main loop, that contains the majority of the computation. This loop has three nested loops, each performing individual tasks. The loop optimization techniques discussed in Section 2.4 can be used here. We followed a similar approach to accelerate the execution time of the code.

In every iteration, the nested loops in the code work on 128 bits of data to compute the output. The size of data worked upon is fixed because the SSE instructions used operate only on 128 bits of data at once. While translating the code into hardware, we parameterized the SSE functions to perform the same functionality on larger data size. The Verilog code has no restrictions of SIMD width, functions can support a more flexible range of data sizes [128, 256, 512, 1024, and 2048].

Suppose that the nested loops are iterating N times and data is set to 128 bits,

total bits computed at the end of iteration will be N x 128 bits. Now If we double the data size to 256 bits, to compute the same amount of data we will require N/2 iterations (N/2 + 1 if N is odd). Hence, the number of iterations will be approximately reduced by half every time the data size is doubled, thus decreasing the execution time of the process.

We wrote Verilog code for the main loop and the three nested loops of the Minimap2 kernel. But we leveraged the optimization technique to parameterize only the fission loop and the core loop because they have similar loop boundary conditions. We didn't optimize the third loop because it has a different loop boundary and update condition than the other two loops in the kernel. We can also pipeline the three nested loops and execute them concurrently to accelerate the process. We leave the optimization of the third loop and pipelining of the nested loops for future work.

The parameterized design require additional logic to mask the unwanted bits of the output data if needed. The following example explains the need for masking logic: Let's assume a loop working on 128 bits of data takes 7 iterations to compute the output. If we double the data size to 256, the number of iterations will be reduced to 4. In the last iteration, only the first 128 bits of the output are valid; the rest of the bits are invalid. Similarly, for the parameter value 512, only 2 iterations are required. The last 128 bits in the last iteration are invalid.

To remove the invalid data we mask the invalid bits.

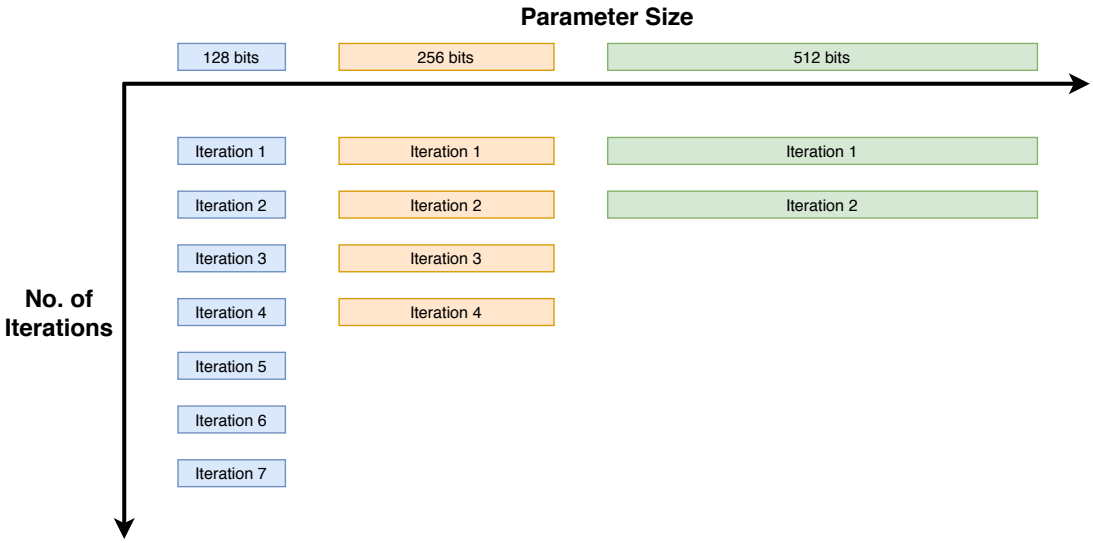

Figure 3.6: No. of iterations vs Parameter size

Chapter 4

Results

In this thesis, we discussed the techniques for offloading software code (Minimap 2) to hardware and how to optimize the design further for better performance. In this chapter, we analyze results obtained by synthesizing the hardware design of Minimap2.

4.1 Experimental Setup

The logic design is synthesized using the Xilinx Vivado tool using the device Xilinx Virtex-7. For the experiment, we selected a unit test for C code and used the same unit test to simulate the HDL code. This ensured that both the C and HDL code received the same input data to work on.

As discussed in section 3.5 III, we have only parallelized the fission loop and the core loop of the kernel. We have taken the following parameter (data-path width) values: 256, 512, 1024, and 2048 for this experiment. The following sections discuss the results obtained by synthesizing the code of minimap2 for various parameter values.

4.2 Timing evaluation

We split the Verilog code into two parts: the optimized part (fission loop and core loop) and the non-optimized part (third loop). We then simulate these Verilog codes at a clock period of 20ns to calculate the execution time. Table: 4.1 shows the value of execution time in 'μs'. The data-path width of the code is set to 128 (base).

	fission and core loop [t]	third loop	Total
Time(μs)	161.3	635	796.3

Table 4.1: Execution time for optimized and non optimized hardware design

Table 4.2 and Figure 4.1 shows the execution time to compute the output for various data-path width values by the optimized hardware design (fission loop and core loop).

Parameter	128 (base) [t]	256 [t′]	512 [t′]	1024 [t′]	2048 [t′]
Time(μs)	161.3	83.2	44.2	24.6	14.9

Table 4.2: Execution time of optimized design for different data-path width

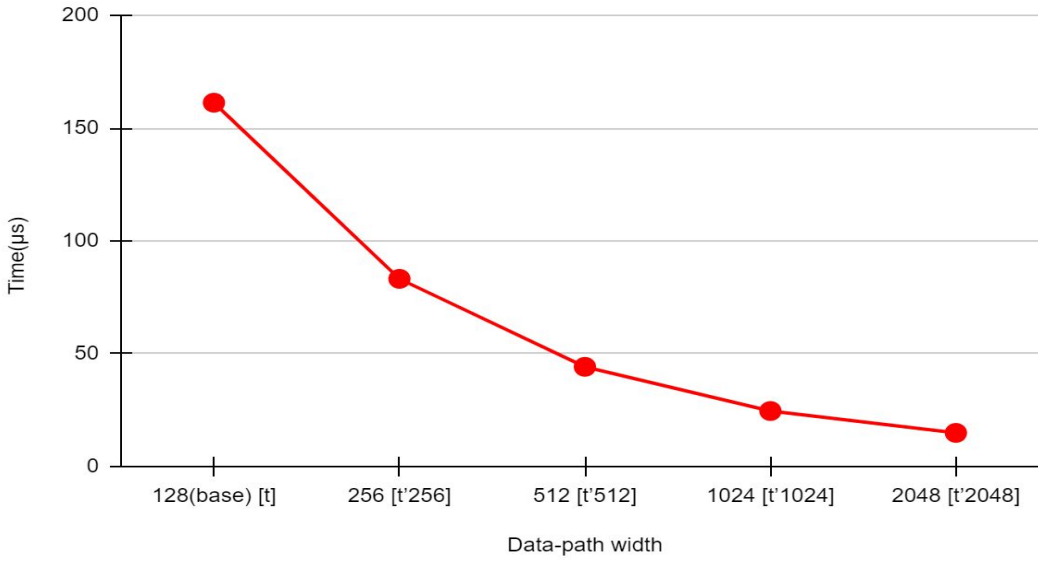

Figure 4.1: Execution time vs Data-path width graph

We can infer from the table and the graph that there is a speedup of 2X every time we double the parameter for optimized design. The speedup is because an increase in data size leads to an increase in concurrent computation, which reduces the design's execution time. But we will achieve this speedup only to a certain parameter value. After a point, any further parallelization of code won't

affect the speed-up of the design. The kernel computes 128 bits per iteration for the fission loop and the core loop. When the code is parallelized, we increase the number of bits worked upon per iteration. Every time we double the number of bits, the number of loop iterations required to compute the same amount of data reduces by half. We will reach a limit of speedup when the number of loop iterations does not reduce any further on the increase in the parameter value.

For the unit test, we are running on the design, the maximum number of iterations is 54. The number of iterations will reduce to 1 when we increase the parameter value by 64 times (closest power of 2 to 54), which equals 8192 bits per iteration. We will reach a speed-up limit when we parallelize the code to compute 8192 bits in a single iteration. Any further parallelization will not affect the timing of the design. The parallelization limit of the design differs for different test cases. There is a possibility of further parameterizing the code for test cases with larger read segments (where the maximum number of loop iterations exceeds 54) to achieve more speed-up.

4.3 Speed-up evaluation

We use Amdahl's law to calculate how much the computation time can be accelerated by running a part of it parallelly. According to Amdahl's law, the maximum theoretical speed-up achieved is given by the formula:

$$S = \frac{1}{(1 - Fe) + (Fe/Se)}$$

where:

Fraction enhanced (Fe) is computation time taken by parallelizable part of design (with no parallelization) [t] over computation time taken by the whole design. [T]

$$Fe = \frac{t}{T}$$

Speed-up enhanced (Se) is computation time taken by parallelizable part of design (with no parallelization) [t] over computation time taken by parallelized design [t']

$$Fe = \frac{t}{t'}$$

We can compute speed up in execution time by using values of t, t' and T from the Table 4.1 and 4.2.

Table 4.3 shows speed- up of the design in percentage for different data-path width.

Parameter	256	512	1024	2048
Speed-up (in %)	10.8	17.3	20.7	22.5

Table 4.3: Speed-up in percentage for different data-path width

4.4 Area Evaluation

We evaluate the area in terms of device utilization numbers which are represented by the number of look up tables (LUT's) and number of registers as a flip flop (FF) used by the design. Table 4.4 and Table 4.5 shows the number of LUT's used and the number of registers as a flip flops used by the optimized design respectively.

Parameter	128	256	512	1024	2048
Number of LUT's	45500	56735	56539	84080	141511

Table 4.4: Number of LUT's in optimized design for different data-path width

Parameter	128	256	512	1024	2048
Number of registers as FF's	51091	52412	55398	61694	73984

Table 4.5: Number of reg as FF's in optimized design for different data-path width

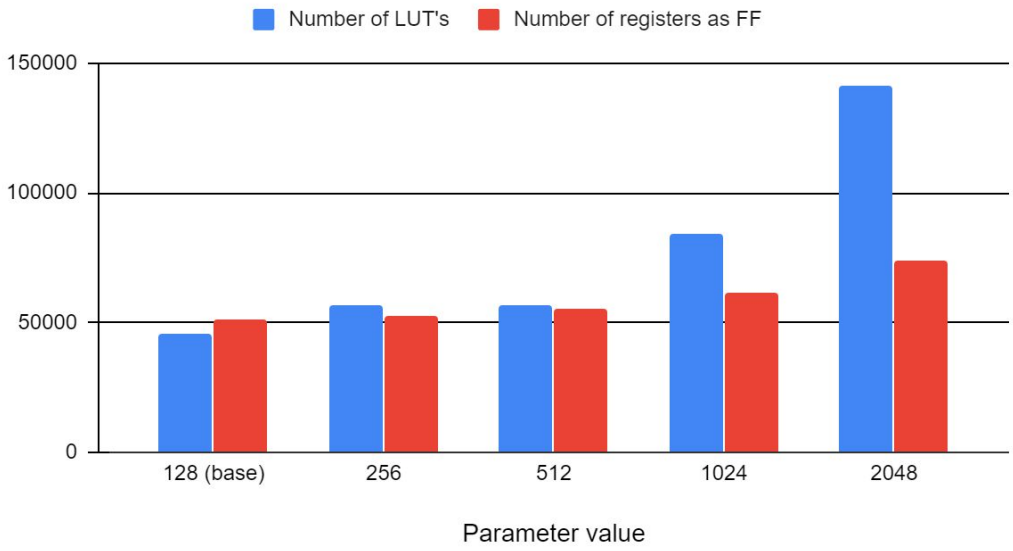

Figure 4.2: Graph showing number of LUT's and register as FF's used by different data-path width

The reference genomes and the arrays formed after dereferencing dynamically allocated memory are represented using register arrays. The huge size of these register arrays is why the number of resources utilized by design is very high. We plan on replacing register arrays with block RAM in the future to reduce the resources used. The size of these register arrays is unaffected by the bit size value. Hence, there is only a marginal increase in the number of resources utilized, depicted in Figure 4.2. We can summarize from the achieved area results that the small increase in the number of LUT's and registers as a FF's is because the design requires additional logic to compute data concurrently, doubling every time we double the parameter value. The optimized design also requires extra logic (for, e.g., masking logic), which also adds on the number of

resources.

4.5 Maximum Frequency

We synthesized the Verilog code to calculate the maximum frequency (F_max) the design can operate on without violating any setup and hold time conditions. To measure the value of F_max we provide the design with a clock whose period value is set such that the Total Negative Slack (TNS) is a positive value. Then we decrease the clock period until we get a value of TNS, which is close to zero. F_max is given by reciprocal of the clock period where TNS approaches zero. For our design when parameter value is 128 (base), the value of the F_max is **30 MHz**. There is a scope of improvement in the maximum frequency value if we pipeline the three loops in the design.

4.6 Conclusion

In this thesis, we discussed techniques to design hardware logic for software kernels. We selected the Minimap 2 kernel as a workload and wrote synthesizable Verilog code for it. A verification environment was created to test the functionality of the hardware design. The design was then parallelized to achieve speed-up in execution time. Upon evaluating the design's performance on time and area, we can conclude that there is speed-up with minimal increase in the area of the design when we double the data size parameter.

Future research will focus on further pipelining the nested loops to compute data concurrently, accelerating the design's computation speed. Moreover, representing input genome sequences and pointer arrays used in the kernel code by block RAM instead of register arrays will ensure that the increase in area is less as parallelization of the design increases.

Bibliography

[1] Y. Turakhia, G. Bejerano, and W. J. Dally, "Darwin: A genomics co-processor provides up to 15,000x acceleration on long read assembly," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, (New York, NY, USA), p. 199–213, Association for Computing Machinery, 2018.

[2] D. Fujiki, A. Subramaniyan, T. Zhang, Y. Zeng, R. Das, D. Blaauw, and S. Narayanasamy, "Genax: A genome sequencing accelerator," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pp. 69–82, 2018.

[3] H. Li, "Minimap2: Pairwise alignment for nucleotide sequences," *Bioinformatics*, vol. 34, pp. 3094–3100, 05 2018.

[4] E. D. Pleasance, R. K. Cheetham, P. J. Stephens, D. J. McBride, S. J. Humphray, C. D. Greenman, I. Varela, M.-L. Lin, G. R. Ordóñez, G. R. Bignell, K. Ye,

J. Alipaz, M. J. Bauer, D. Beare, A. Butler, R. J. Carter, L. Chen, A. J. Cox, S. Edkins, P. I. Kokko-Gonzales, N. A. Gormley, R. J. Grocock, C. D. Haudenschild, M. M. Hims, T. James, M. Jia, Z. Kingsbury, C. Leroy, J. Marshall, A. Menzies, L. J. Mudie, Z. Ning, T. Royce, O. B. Schulz-Trieglaff, A. Spiridou, L. A. Stebbings, L. Szajkowski, J. Teague, D. Williamson, L. Chin, M. T. Ross, P. J. Campbell, D. R. Bentley, P. A. Futreal, and M. R. Stratton, "A comprehensive catalogue of somatic mutations from a human cancer genome," *Nature*, vol. 463, p. 191—196, January 2010.

[5] A. Lacour, A. Espinosa, E. Louwersheimer, S. Heilmann, I. Hernández, S. Wolfsgruber, V. Fernández, H. Wagner, M. Rosende-Roca, A. Mauleón, S. Moreno-Grau, L. Vargas, Y. A. L. Pijnenburg, T. Koene, O. Rodríguez-Gómez, G. Ortega, S. Ruiz, H. Holstege, O. Sotolongo-Grau, J. Kornhuber, O. Peters, L. Frölich, M. Hüll, E. Rüther, J. Wiltfang, M. Scherer, S. Riedel-Heller, M. Alegret, M. M. Nöthen, P. Scheltens, M. Wagner, L. Tárraga, F. Jessen, M. Boada, W. Maier, W. M. van der Flier, T. Becker, A. Ramirez, and A. Ruiz, "Genome-wide significant risk factors for alzheimer's disease: role in progression to dementia due to alzheimer's disease among subjects with mild cognitive impairment," *Molecular psychiatry*, vol. 22, pp. 153–160, Jan 2017. 26976043[pmid].

[6] Y. Cho, C.-H. Lee, E.-G. Jeong, M.-H. Kim, J. H. Hong, Y. Ko, B. Lee, G. Yun,

B. J. Kim, J. Jung, J. Jung, and J.-S. Lee, "Prevalence of rare genetic variations and their implications in ngs-data interpretation," *Scientific Reports*, vol. 7, p. 9810, Aug 2017.

[7] D. A. Benson, I. Karsch-Mizrachi, D. J. Lipman, J. Ostell, and D. L. Wheeler, "GenBank," *Nucleic Acids Research*, vol. 33, pp. D34–D38, 01 2005.

[8] H. G. Suarez, B. E. Langer, P. Ladde, and M. Hiller, "chainCleaner improves genome alignment specificity and sensitivity," *Bioinformatics*, vol. 33, pp. 1596–1603, 01 2017.