Chapter 7

Accelerating the software implementation using HLS

7.1 Analysing software performance

In the SDSoC environment, it is possible to analyze the software running on the SoC chip using a TCF profiler. After running this analysis, the TCF profiler returns an overview of the functions, sorted on the amount of time spent when running. The analysis is shown in figure 7.1

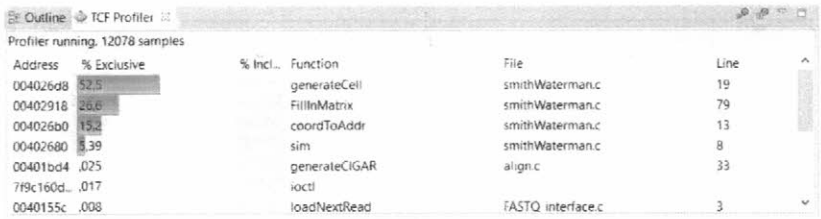


Figure 7.1: a TCF profile of the software implementation

When examining this analysis, we should keep in mind that the generateCell, coordToAddr, and sim functions are inline functions used in the FillInMatrix method. Just as suspected the software spends almost all of its time in these methods, so it's worth it to try to accelerate these functions.

7.2 Recoding parts of the software to be more hardware friendly

7.2.1 Recoding the Cell generation layer

Back in 2011, Vermij E. did a thesis [22] on RVE (recursive variable expansion). He discusses the most efficient ways to program a processing element to generate one value in the alignment matrix. His results can be found in figure 7.2.

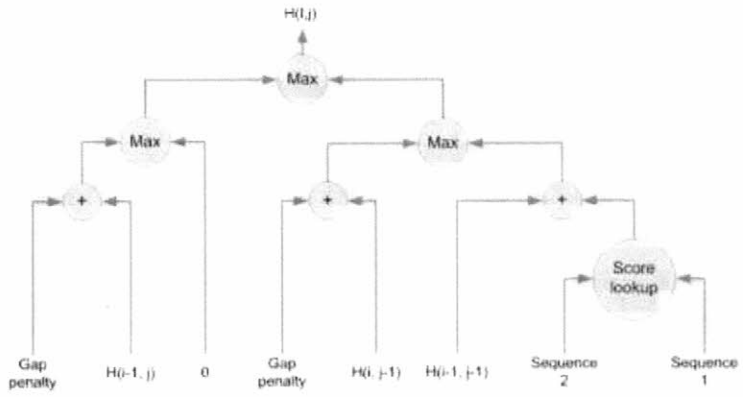


Figure 7.2: The optimal processing found in the thesis from Vermij E.[22]

It seemed like a good idea to reimplement the generatecell function, using this newly found scheme. However, it is also important to keep track of where the value comes from. Therefore, the following (new) code was adopted for generating a cell:

```
//calculate the possible  values
CELL diagonalCELL = { diagonal.value + sim(refVal, seqVal), 1 };
CELL leftCELL = { left.value - gp, 2 };
CELL upCELL = { up.value - gp, 3 };
CELL zeroCELL = { 0, 0 };

CELL upstreamA = (leftCELL.value > upCELL.value) ? leftCELL : upCELL;
CELL upstreamB = (diagonalCELL.value > zeroCELL.value) ?
diagonalCELL : zeroCELL;

CELL newCell = (upstreamA.value > upstreamB.value) ? upstreamA : upstreamB;

//Return the cell:
return newCell;
```

Where the second attribute in the CELL type is the direction.

7.2.2 Recoding the FillIn layer

HLS does not support input and output from the same memory locations in hardware, therefore, the Fillin layer also has to be recoded. We will take a look at the data dependencies again:

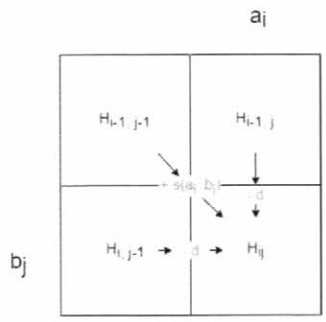


Figure 7.3: Data dependencies for generating a cell in the alignment matrix

Since the current cell only depends on the left-up 3 cells, we can compute every cell on the diagonal in parallel. Therefore, 3 arrays where created: one contains the current diagonal being generated, one contains the previous diagonal for the cell above and the cell to the left of the current generated cell, and one for the diagonal before that. This last array will house the left-up cell also needed for the new cell generation. A schematic of the data structures used can be found in table 7.1

	ref	T	G	T	T	A	C	G	G
seq	0	0	0	0	0	0	0	0	0
G	0	0	3	1	0	0	?		
G	0	0	3	1	0	?			
T	0	3	1	6	?				
T	0	3	1	?					
G	0	1	?						

=>

ppD	pD	cD
0	0	0
0	0	?
1	0	?
1	6	?
3	1	?
0	1	?

Table 7.1 The 3 newly created arrays to house the data needed for generating the next diagonal of cells. The '?' in the cD array (red) represents cells that are currently being generated. Notice that all the information needed to generate the new cell is present in the other 2 arrays

Also, keep note that the FillIn layer should keep track of the maximum location in the matrix. Having gained this new information, we can recode the FillInlayer as follows:

1. Create the arrays ppD, pD and cD. They should be the length of the sequence. All are initialized on zeros.
2. Start the current diagonal at the second column.
3. For every cell at the diagonal, do the following:
   (a) Check for edge cases. The row has to be smaller than the length of the sequence, the column can't be 0 or smaller and the column can't be bigger than the length of the reference.
   (b) Generate the new cell using the data in the ppD and pD arrays.
   (c) Write this new cell to memory, and its value to the cD array.

(d) Check if it is bigger than the current maximum. If so, the current maximum should be this new cell.

4. Set current diagonal to the next one.
5. Shift the values ($cD \rightarrow pD$ and $pD \rightarrow ppD$) and repeat from step 3, until the full matrix is filled.
6. Return the position of the maximum to the alignment level.

Note that by programming the Fill in Layer this way, there is no need to read cells from memory, since they can be generated using the arrays. Only the reference and read sequences should be read from memory.

7.3 Hardware acceleration

When accelerating the application, it was chosen to implement the FillIn and cell generation layer in hardware, since they are the functions where most computational time is spent (see 7.1). Therefore, they were merged into 1 convenient function, which will be transferred to the programmable hardware.

7.3.1 DMA

DMA or Direct Memory Access allows certain hardware systems in the computer to access the main system memory independent of the CPU. This can be of importance to the performance since the hardware system requires a lot of data movement between to and from the memory. The arrays (ref, seq, and matrix) are available in the memory, which means the programmable hardware should be able to access these arrays using DMA.

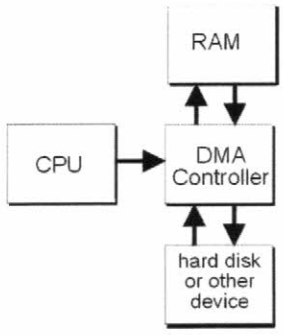


Figure 7.4: Schematic of a system using DMA

The following pragmas were used for the DMA:

```
#pragma SDS data access_pattern(ref:SEQUENTIAL, seq:SEQUENTIAL,
        matrix:SEQUENTIAL)
#pragma SDS data sys_port(ref:AFI, seq:AFI, matrix:AFI)
#pragma SDS data mem_attribute(ref:PHYSICAL_CONTIGUOUS,
        seq:PHYSICAL_CONTIGUOUS, matrix:PHYSICAL_CONTIGUOUS)
#pragma SDS data zero_copy(ref[0:refMax], seq[0:seqMax],
        matrix[0:refMax*seqMax])
```

- **access_pattern** can be either SEQUENTIAL or RANDOM. It specifies the data access pattern by the hardware to the memory. If SEQUENTIAL is used, the interface will be a stream (e.g. ap_fifo). On the other hand, when RANDOM is used, a RAM interface will be generated.
- **sys_port** can be ACP, AFI, GP or MIG. It specifies on which level the memory is interfaces.
  - ACP: Lets hardware functions have cache-coherent access to DDR using the PS L2 cache.
  - AFI: Hardware functions have fast non-cache coherent access to DDR via the PS memory controller.
  - GP: The processor directly writes/reads data to/from hardware function. This would be inefficient for large data transfers.
  - MIG: Hardware functions access DDR from PL via a MIG IP memory controller.

  We have quite a large amount of data, so GP doesn't seem like a good option. Likewise, the memory will be accessed uniform, we have no use for a cache as this would just slow it down. Therefore the AFI was selected.
- **mem_attribute** can be either PHYSICAL_CONTIGUOUS or NON_PHYSICAL_CONTIGUOUS. The default value is NON_PHYSICAL_CONTIGUOUS. PHYSICAL_CONTIGUOUS is used for memory allocated with sds_alloc. Likewise, NON_PHYSICAL_CONTIGUOUS is used with memory allocated using malloc.
- **zero_copy** means that the hardware function accesses the data directly from shared memory through an AXI master bus interface.

The data_pack pragma The data_pack pragma is used on the matrix since it is an array of structs (CELL type). This packs the data fields of a struct into a single scalar. The bitwidth of the packed struct is the sum of its attributes.

Note, the bitwidth of (packed) data on the AXI but must be a power of 2, to interface with the memory. Therefore, a choice was made to change the CELL_VALUE type to an int16_t and the direction to uint16_t. However, this is quite wasteful towards memory usage. It can be optimized by remodeling some of the code, but due to time constraints, this was not implemented.

7.3.2 Unrolling pragmas

Which loops are unrolled and how, is best described in pseudocode, which is found hereunder:

```
Create the current maximum (=0) and the position of the maximum;
Create the 3 arrays (pD, ppD, and cD) of size seqMax;

for i from 0 to seqMax: //initializing the arrays
   #pragma HLS UNROLL
   pD[i] = ppD[i] = cD[i] = 0;

for currentColumn from 2 to (refLength + seqLength): //The fillIn loop
   #pragma HLS loop_tripcount min=5000 avg=5150 max=5700
   #pragma HLS PIPELINE

   for i from 0 to seqMax: //the diagonal loop
      #pragma HLS UNROLL
      row = i;
      col = currentColumn - i;

      //edge cases
      if (i < seqLength && col != 0 && col <= refLength):
         generateCell(  //cell generation layer
                diagonal value = ppD[i-1],
                left value = pD[i],
                up value = pD[i-1],
                reference value = ref[col],
                sequence value = seq[row]
         );
         store the current cell at cD[i];
         store the current cell in memory;
         if (value of cell bigger than maximum):
            replace current max position with this cell;


   for i from 0 to seqMax: //the shifting loop
      #pragma HLS UNROLL
      ppD[i] = pD[i];
      pD[i] = cD[i];

return the position of the maximum;
```

The unroll pragma transforms the loop to multiples copies of the loop body in the FPGA hardware, which allows the loop iterations to be executed in parallel. Since no parameters are given to this pragma, the loops will be unrolled fully, so the whole loop will be executed in parallel. Note that the number of iterations of the loop should be known at compile time.

In the code, all loops are unrolled fully, except for the loop that runs over every diagonal (the fill-in loop). Unrolling this loop would not affect execution speed since all elements in the loop are dependent on the previous iteration of this loop.

The loop_tripcount pragma is applied to a loop to manually specify the total number of iterations performed by a loop. In our code, this is applied to the fill-in loop since the number of iterations is not known at compile time (refLength and seqLength are variables).

The pipeline pragma will try to transform the body of the loop to a pipeline, where every iteration of the loop has a given II or *Initialization Interval*. If no II is given, the default is 1. This means it will try to run one iteration of the loop body for every clock cycle.

7.3.3 Comparison with the software

If we examine execution time (using the built-in latency analyzers in SDSoC), we can see we achieved a speedup of 4.41. This means the hardware variant of the implementation runs 4.41 times faster than the software variant.

Performance estimates for 'FillInHW in align.c:250' funct ...

SW-only (Measured cycles)	94414155
Hardware accelerated (Estimated cycles)	21410374
Estimated speedup	4,41

Performance estimates for 'FillInHW in align.c:272' funct ...

SW-only (Measured cycles)	94414155
Hardware accelerated (Estimated cycles)	21410374
Estimated speedup	4,41

Resource utilization estimates for Hardware functions

Resource	Used	Total	% Utilization
DSP	0	1728	0
BRAM	4	312	1,28
LUT	125724	230400	54,57
FF	53266	460800	11,56

Figure 7.5: The latency analyzer in SDSoC. We can see the number of clock cycles spent in the software and the hardware variant of the implementation, as well as the speedup. The speedup is calculated 2 times, since we run the fillIn function twice, once for the forward and once for the reverse sequence. The resource utilization of the FPGA hardware is also visible in this picture

Chapter 8

Conclusion and future research

This chapter will contain some finals words on the problem definition and the proposed implementation, as well as some recommendations for future research.

8.1 Conclusion

Problem definition The first step in analysing DNA, is to sequence it. This will determine which bases and in which order they are occuring in the DNA. However, the technology of DNA sequencing machines limit us to reads of 75-300 bases long. Therefore, the DNA is multiplied and cut into short pieces before sequencing. If we want to make assumptions about the whole genome, we will need to input a lot of these reads.

Typically, the read is compared with the whole genome in a local alignment, for example with the Smith-Waterman algorithm. As an output, we would get the position in the human genome and an alignment with its score (how well the sequence fits in that spot). This practice is commonly referred to as *Mapping to reference genome*.

Since the reads from DNA sequencing machines are 75 to 300 bases long, and the whole human genome is approximately 3 billion bases, this comparison is computationally a very intensive task. If we analyze the S-W algorithm (as we have done in 3.3.2), we can see that the value of each cell in the matrix is only dependent on the left-upmost 3 cells. Therefore, it leads us to believe that this algorithm can be accelerated on other hardware solutions such as an FPGA (as discussed in 4) since S-W is heavily parallelizable.

In most clinical applications where mapping to a human reference genome is used, the number of reads to be compared with the genome is in the millions, which further increases the demand to speedup the process of mapping the reads in the genome, to decrease the time-to-result.

The idea of this thesis was to implement the Smith-Waterman alignment algorithm on an MPSoC, which contains both programmable FPGA hardware and an ARM processor in 1 chip. As a target board, the ZCU 104 evaluation kit was chosen.

Proposed solution and implementation As a starting point, a software implementation was implemented on the ARM processor. After running the software implementation overnight with the unmapped sequences from the coronavirus as a sample set, we obtained a dataset of mapped reads. The reads were also mapped by using the Galaxy online tool [13] and both were imported into the IGV analyzing software for comparison, which can be seen in figure 8.1

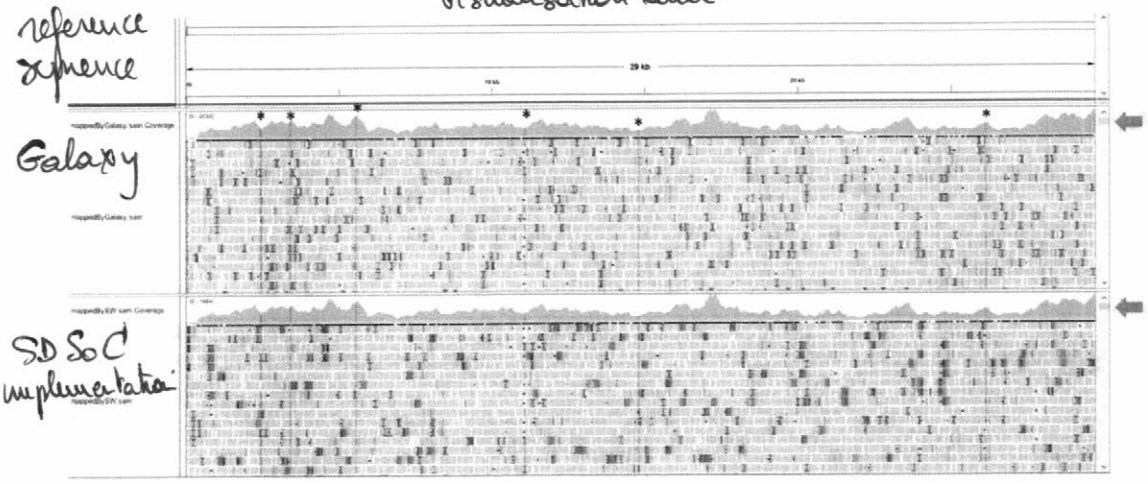
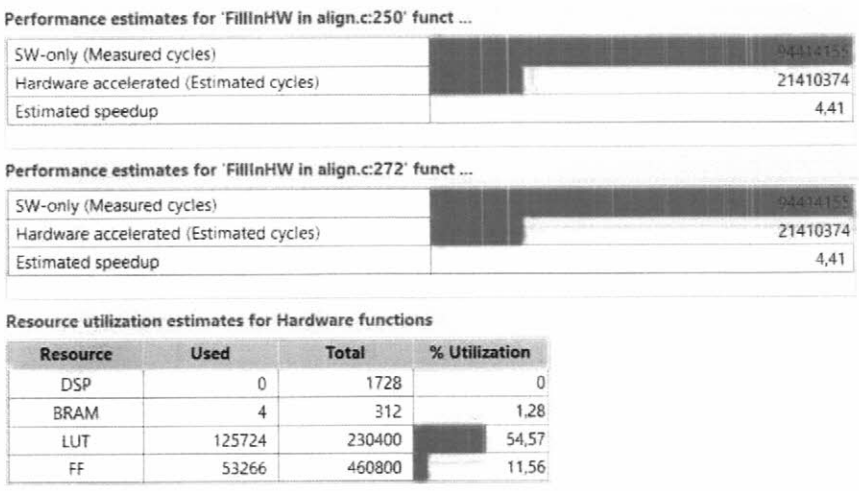


Figure 8.1: A comparison of the data obtained by Galaxy (top) and the data obtained by the implementation discussed in this chapter (bottom)

It can be observed that the implementation is working correctly since the reading depth graphs are approximately the same (indicated by an arrow). ~~Furthermore, IGV was able to detect and identify consistent substitutions or indels in the reads (indicated by an asterisk).~~

If you look closely to figure 8.1, some small differences between the exact reads are visible. This will probably be because both the algorithm and the parameters were a bit different. However, the important part is that the reading depths are the same, as well as the consistently mutated bases marked in the genome (indicated by an asterisk).

Hardware speedup After implementing the matrix fillIn in the FPGA hardware, we can examine execution time (using the built-in latency analyzers in SDSoC). The estimated achieved speedup ~~that is~~ 4.41X. This means the hardware variant of the implementation runs 4.41 times faster than the software variant.



Performance estimates for 'FillInHW in align.c:250' funct ...

SW-only (Measured cycles)	94414155
Hardware accelerated (Estimated cycles)	21410374
Estimated speedup	4,41

Performance estimates for 'FillInHW in align.c:272' funct ...

SW-only (Measured cycles)	94414155
Hardware accelerated (Estimated cycles)	21410374
Estimated speedup	4,41

Resource utilization estimates for Hardware functions

Resource	Used	Total	% Utilization
DSP	0	1728	0
BRAM	4	312	1,28
LUT	125724	230400	54,57
FF	53266	460800	11,56

Figure 8.2: ~~The~~ latency analyzer in SDSoC. We can see the number of clock cycles spent in the software and the hardware variant of the implementation, as well as the speedup. The speedup is calculated 2 times, since we run the fillIn function twice, once for the forward and once for the reverse sequence. The resource utilization of the FPGA hardware is also visible in this picture

8.2 Recommendations for future work

Naturally, we have several recommendations for the continuation of this research. They will be classified in improvements of the current implementation and re-evaluation of the used algorithm.

~~8.2.1~~ Recommendations for improving the current implementation

~~Before, during, and after implementing, some~~ optimizations ~~which~~ could improve the proposed implementation ~~came up. However, we did not have time to implement them, so hereunder they will be mentioned:~~

- Treat the gap opening and extension separately. In the current implementation, the gap penalty is treated linearly. However, the S-W algorithm could be refined to include an affine gap penalty. More info on the gap penalty can be found at 3.3.2.
- In a sequence, a base may be suddenly marked with an 'N' character, Which represents a base that was unidentified in the primary processing. In this thesis, it was decided to cut the sequence short at the moment an 'N' character is registered. However, a possible improvement might be to keep these unidentified bases in the alignment as a "generic" base and in this way improve the accuracy of the alignment. More info on the current implementation can be found at 6.3.1.2.
- Make some types more memory efficient, such as the BASE and the SEQ_INDEX type. Firstly, the BASE type only has 4 possible values (the 4 nucleotide bases). However, it is

defined as 1 byte in the current implementation, since it is the smallest type that C supports. However, storing a base in 1 byte is memory inefficient. Improvements could be made by defining an own type, by stacking 4 bases in 1 byte. Secondly, the SEQ_INDEX type is stored as 2 bytes in the current implementation, even though it only contains values up to 300, which makes it memory inefficient. This could also be improved.

- If this implementation would be used in practice, there would be a need to find an easier way to on and offload data to board. Currently, this is accomplished by changing out the SD card, which could easily be improved using FTP since an Ethernet stack is present due to the operating system. The speed the data would be transferred at is not important since we can assume the time it takes to map the sequence takes far longer.

8.2.2 Recommendations for re-evaluating the alignment method

In the field of alignment algorithms, a few alternatives exist to S-W since it takes a lot of computing power. In most cases, these algorithms will use a transformation or a lookup table to determine possible candidate locations, where a small regional S-W will be performed on a part of the genome. For example, The BFAST algorithm uses a hash table where candidate locations are stored for every sequence [17]. As a second example, in the Bowtie application, the candidate locations are determined using the Burrows-Wheeler Transform.

As future work, one of these more sophisticated algorithms can be selected. Some candidate locations can then be determined in a way defined by the algorithm. These candidate locations can then be mapped using the implementation of this thesis.

Summary

Tijdens de laatste decennia hebben biologen ~~steeds~~ grote stappen gezet in het begrijpen van het leven; dieren, mensen en planten. Sinds het opkomen van DNA sequencing technieken is genetica een ~~groot~~ onderdeel geworden van de biologie. Maar, door de hoeveelheid data die verwerkt moet worden, is het analyseren van genetische toepassingen erg rekenintensief voor computers, bijvoorbeeld bij het menselijk genoom, wat ~~een lengte heeft van~~ 3 miljard baseparen.

Een grote fundamentele applicatie binnen de genetica is het "short read genome mapping" of het "short read alignment", wat probeert de locatie van een kort stukje DNA terug te vinden in het volledige genoom. Als er genoeg van deze "reads" gealigneerd kunnen worden, kan hier veel interessante info uit worden afgeleidt. Bijvoorbeeld, bij voldoende aligneringen kunnen we het volledige genoom afleiden uit ons bemonsterd DNA. ~~Zelfs aan~~ het aantal reads die aligneren op 1 ~~plaats~~ (wat men ook wel de "reading depth" noemt) ~~kunnen we dingen afleiden, zoals bijvoorbeeld~~ een trisomy van chromosoom-21 ~~bij patiënten met het~~ (Down syndroom).

Typisch word een "read" vergeleken met het volledige genoom via het Smith-Waterman algoritme. Als resultaat krijgen we dan de positie van deze read terug in het genoom.

Een volledig genoom in 1 keer bepalen kan niet, vanwege de technieken die de sequencing machines gebruiken. Deze machines kunnen enkel reads van een korte lengte aan, met een maximum van een paar honderd baseparen ineens. Tegenwoordig wordt er meer en meer DNA gesequenced, en dit groeit exponentieel. Gezien er meer vraag is naar goede aligneringstechnieken, moeten deze ook worden verbeterd om bij te blijven met de stijgende vraag naar DNA sequencing. Maar, meestal worden deze nieuwe technieken enkel geprogrammeerd op de standaard processoren van een computer.

In deze thesis zullen we het Smith-Waterman algoritme bestuderen, en uit deze studie leren we dat een implementatie op een "gewone" processor niet de beste optie is. Er bestaan andere elektronische technologieën om dit algoritme op uit te voeren, zoals bijvoorbeeld een FPGA. We zullen een MPSoC gebruiken, die beide een stuk ARM (de "gewone" processor) en FPGA (de gespecialiseerde hardware) aan boord heeft.

In de meeste klinische toepassingen waar dit soort alignering gebruikt wordt, is het aantal reads die gealigneerd moeten worden in de miljoenen. Als ultiem resultaat willen we de "time-to-result" van een klinische test verkleinen, zodat de rekenkracht van de computers niet de bottleneck wordt van de tests.

Eerst werd een softwareversie van het algoritme geïmplementeerd op de ARM processor. Deze

hebben we ~~ook~~ getest met het SARS-CoV-2 (coronavirus) als een dataset, en vergeleken met resultaten bekomen via ~~traditionele methoden~~. Hier zagen we dat onze implementatie dezelfde resultaten ~~heeft~~. Verder waren we ook in staat om mutaties te ontdekken in het gemonsterde genoom in vergelijking met onze referentie.

Nadat we dit ook geïmplementeerd hebben op de FPGA gespecialiseerde hardware, hebben we een implementatie die dezelfde resultaten betoonde als onze softwareversie, maar deze resultaten 4 keer sneller kan bereiken.

Abstract

One fundamental application in genomics is short read genome mapping, which attempts to locate a short read of DNA on the full genome. If enough of such short reads are mapped, some interesting results can be achieved. For example, if enough reads are mapped, we can extrapolate the full genome of the organism. Even the amount of reads in one place (referred to as the reading depth) can give useful information, e.g. the presence of trisomy of chromosome-21 in Down syndrome patients.

Typically, the read is compared with the whole genome in a local alignment, for example with the Smith-Waterman algorithm. As an output, we would get the position in the human genome and alignment with its score (how well the sequence fits in that spot). This practice is commonly referred to as *Mapping to reference genome*.

A full genome cannot be determined immediately, because the machines that determine this sequence can only handle short reads consisting of a few hundred bases. These sequencing machines are currently capable of producing millions of reads per day, and their throughput is growing at an exponential rate. This exponential growth should be accompanied by an improvement in genome mapping techniques, to keep up with the throughput of these machines. However, most of the current software tools used for genome mapping are run on traditional CPUs.

If we analyze the S-W algorithm, we can see that the value of each cell in the matrix is only dependent on the left-upmost 3 cells. Therefore, it leads us to believe that this algorithm can be accelerated on other hardware solutions such as an FPGA since S-W is heavily parallelizable.

In most clinical applications where mapping to a human reference genome is used, the number of reads to be compared with the genome is in the millions, which further increases the demand to speedup the process of mapping the reads in the genome, to decrease the time-to-result.

The idea of this thesis was to implement the Smith-Waterman alignment algorithm on an MPSoC, which contains both programmable FPGA hardware and an ARM processor in 1 chip. As a target board, the ZCU 104 evaluation kit was chosen.

As a starting point, a software implementation was implemented on the ARM processor. After running the software implementation overnight with the unmapped sequences from the SARS-CoV-2 (coronavirus) as a sample set, we obtained a dataset of mapped reads. The reads were also mapped by using the Galaxy online tool. We observed that our implementation is working correctly since the reading depth graphs are approximately the same (indicated by an arrow). Furthermore, IGV was able to detect and identify consistent substitutions or indels in the reads.

The exact results of the Galaxy online tool were not achieved, but this will probably be because the parameters are a bit different. However, the important part is that the reading depths are the same, as well as the consistently mutated bases marked in the genome that were detected.

After implementing the Smith-waterman matrix fill-in in the FPGA hardware, we achieved speedup of 4.41 in comparison with an implementation running fully on the ARM processor.

Keywords: Smith-Waterman algorithm, short read genome mapping, accelerator, MPSoC, FPGA, HLS

Bibliography

[1] Bioinformatics company. http://www.clccell.com/download.html.

[2] Bioinformatics company. https://www.cray.com/.

[3] Bioinformatics company. http://www.timelogic.com/.

[4] figure: azure machine learning fpga comparison.
https://docs.microsoft.com/en-us/azure/machine-learning/service/media/concept-accelerate-with-fpgas/azure-machine-learning-fpga-comparison.png.

[5] figure: Dna chemical structure. figure: DNA chemical structure.

[6] figure: Double stranded dna with coloured bases.
https://upload.wikimedia.org/wikipedia/commons/thumb/1/14/Double_stranded_DNA_with_coloured_bases.png/10
Double_stranded_DNA_with_coloured_bases.png.

[7] figure: fpga diagram. https://evergreen.loyola.edu/dhhoe/www/FPGA_diagram.png.

[8] figure: Karyotype.
https://upload.wikimedia.org/wikipedia/commons/thumb/b/b2/Karyotype.png/800px-Karyotype.png.

[9] figure: sequential working of microprocessor.
http://www.lighterra.com/papers/modernmicroprocessors/sequential2.png.

[10] figure: Trisomy 21 male.
http://www.anatomybox.com/wp-content/uploads/2011/07/Trisomy-21-male.png.

[11] Genome of coronavirus, ncbi, https://www.ncbi.nlm.nih.gov/sra/SRX7852918.

[12] igv download page.
http://software.broadinstitute.org/software/igv/.

[13] Online bioinformatics tools.
https://usegalaxy.org/.

[14] (2020). *Sequence Alignment/Map Format Specification*. The SAM/BAM Format Specification Working Group.

[15] Alexandrov, V. (2006). General purpose computations on graphics hardware: methods, algorithms and applications. In *Computational science - ICCS 2006 : 6th international conference, Reading, UK, May 28-31, 2006 ; proceedings*. Berlin: Springer.

[16] Nollet, F. (2019). powerpoint: Howest 2019-2020 sanger and ngs.

[17] Olson, C. B. (2011). An fpga acceleration of short read human genome mapping. Master's thesis, University of Washington.

[18] O.O. Storaasli, D. S. (2011). Accelerating genome sequencing 100x with fpgas. http://ft.ornl.gov/ olaf/pubs/RSSIOlafDave.pdf.

[19] S. Drury, M. Hill, L. C. (2016). *Cell-Free Fetal DNA Testing for Prenatal Diagnosis*. Elsevier, Inc.

[20] Smith, T. F. and Waterman, M. S. (1981). Identification of common molecular subsequences. *Journal of Molecular Biology*, 147:195–197.

[21] Vergel, R. (2018). *Getting started with SDSoC*. Xilinx Inc. https://github.com/Xilinx/SDSoC-Tutorials/tree/master/getting-started-tutorial.

[22] Vermij, E. (2011). Genetic sequence alignment on a supercomputing platform. Master's thesis, Delft university of technology.