Designing a hardware accelerator for a selective sweep detection algorithm using high-level synthesis

Niek Sterenborg
Faculty EEMCS, CAES, University of Twente
Bsc Thesis Committee: dr.ing. N. Alachiotis, S. Safapourhajari MSc, IR. E. Molenkamp
Enschede, The Netherlands - February 14, 2021
n.e.sterenborg@student.utwente.nl

***Abstract*—Genetic analysis is used widely by humanity. Use cases range from researching adaptive evolution to combating drug resistance among diseases. Since natural selection was discovered humans have been developing their understanding of genetic phenomena continuously. Detecting selective sweeps can give insight in mutations of a species or population and help to take action if required. To detect these sweeps, some software solutions exist that use the sweep signatures around a selective sweep to determine the position of the sweep. In this paper the OmegaPlus algorithm is analysed and accelerated with an FPGA. The core calculation of OmegaPlus is changed slightly so that it is suitable to be synthesized to HDL. Vivado High Level Synthesis is used to synthesize the code to HDL. OmegaPlus already has different versions that use multithreading to improve computation times. The throughput of these different versions is measured and compared with the throughput of the accelerator presented in this paper. The designed accelerator achieves speedups between 2.12x and 46.84x based on the version of OmegaPlus and the target FPGA.**

I. Introduction

Natural selection is the process with which species adapt and evolve. A genomic mutation in an individual organism can be positive, negative or have no impact, resulting in a different chance of survival and reproduction for the individual. This difference between individuals gives some an advantage over others. If the mutation is advantageous for the individual it results in positive selection, and a disadvantageous mutation leads to negative selection. Advantageous alleles increase in frequency over the population and eventually become fixed, while disadvantageous alleles decrease in frequency.

In population genetics, a selective sweep is known as the process in which a beneficial mutation increases in frequency throughout a population and becomes fixed (i.e. all individual organisms have the mutation after some time). [1] When such a sweep happens it leaves three types of signatures around the sweep location. The first is the locally reduced levels of polymorphism. The second is a shift in the site frequency spectrum (SFS) towards low- and high-frequency derived variants. The Third signature is a local linkage disequilibrium (LD) level pattern around the sweep that features high LD-levels on each side of the mutation, and low LD levels between loci on each side of the sweep. From these three types of signatures, the LD signature produces the most accurate results when detecting selective sweeps. [2] Detecting such a sweep in a genomic population data set can be done by using one of the available software solutions. These software solutions use the signatures that selective sweeps leave behind on the genomic data. The available software solutions all work slightly different and have their own advantages and disadvantages. Since the LD based method of locating selective sweeps was proven to be more accurate than other solutions, OmegaPlus was chosen as the baseline for acceleration. [3]

OmegaPlus uses the ω-statistic to find selective sweeps in DNA datasets. The omega statistic uses Linkage Disequilibrium (LD) patterns around the position of interest to determine the score of a sweep being present at the location. [4] A higher result of the calculation means a higher chance of a sweep at the location. The calculation is further discussed in the section II-C.

Depending on the size of the data set and genomic area of interest, computation of OmegaPlus can take quite a while. Especially when analyzing larger populations, or longer parts of the DNA of the individual organisms the number of calculations needed increases quickly. A solution to these long execution times is an accelerator for the calculation. This takes the same input as the calculation, but dedicated hardware calculates the output instead of the CPU. This relieves the processor running the program of the repeating core calculation because it only has to provide the inputs, and read the outputs of the accelerator. The accelerator design presented in this paper can achieve up to 46.84x faster calculation than the reference software depending on the size of the calculation and accelerator targeted FPGA board.

This paper first gives a bit more background information about selective sweeps, the ω-statistic and OmegaPlus in section II, and presents some related work on accelerating LD calculations and decreasing computation time of OmegaPlus in section III. After this the architecture of the accelerator is shown and explained in section IV, the implementation is explained in section V, and the results are shown and discussed in section VI. The paper ends with the conclusion and recommendation for future work in section VII.

II. Background

A. Selective Sweep theory

A selective sweep occurs when an organism in a certain population gets a beneficial mutation. Over the course of time, this mutation increases in frequency over the population because of the higher chance of reproduction of that specific organism. From a certain moment, all individuals in the population will have the mutation. Figure 1 shows a selective sweep.

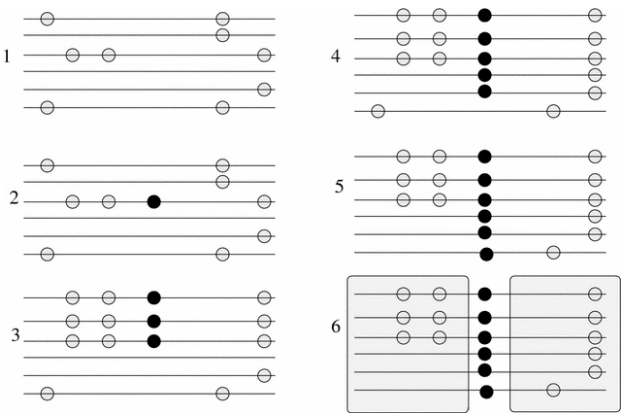

Fig. 1. Selective sweep visualized. Adapted from [1]
(1) A population, every line represents a single organism. The circles are mutations.
(2) A beneficial mutation (black circle) occurs.
(3-4) The beneficial mutation increases in frequency over the population. Note that also other mutations of the individual with the beneficial mutation see increased frequency.
(5-6) The selective sweep completes. The LD pattern that emerges on the sides of the location of the sweep can be used to detect the sweep.

The phenomena mentioned at point 3-4 is known as genetic hitchhiking. Other mutations around a selective sweep also increase in frequency because the individual with the beneficial mutation has an increased chance of reproduction.

B. Detecting a selective Sweep

A selective sweep can be detected by looking at the Linkage Disequilibrium (LD) pattern around the position of the sweep. LD is the non-random association of alleles at different locations in a population. As mentioned above, other mutations hitchhike along with the selective sweep. This causes elevated LD on each side of the beneficial mutation, and decreased LD on sites that are on opposite sides of the mutation. These varying LD levels can be calculated, and used to determine the location of a selective sweep.

C. The omega-statistic

The calculation that is used in OmegaPlus is first introduced by Yuseob Kim and Rasmus Nielsen in [4]. In this paper they construct the calculation based on the LD pattern around selective sweeps. The proposed measure is shown in equation 1.

$$\omega = \frac{\left(\binom{l}{2} + \binom{S-l}{2}\right)^{-1} \left(\sum_{i,j \in L} r_{ij}^2 + \sum_{i,j \in R} r_{ij}^2\right)}{\left(l\,(S-l)\right)^{-1} \sum_{i \in L, j \in R} r_{ij}^2} \quad (1)$$

In equation 1 a genomic window with width S SNPs is assumed. This window is split into a left and right sub-region with width l and $S - l$ respectively. r_{ij}^2 represents one LD measure, which is the squared correlation coefficient between sites i and j.
In the calculation, both properties of LD levels that are known to occur around selective sweeps are used. The elevated LD at either sides of the sweep location is present in the numerator, and the correlation between sites on opposite sides of the sweep is in the denominator of the equation. This makes the ω-statistic a measure for likeliness of selective sweep at the location it was calculated.

D. Calculation in C

The calculation is implemented in C in OmegaPlus as a nested for loop. The code is shown in appendix A for reference. It calculates the ω-statistic for a predefined range of left- and right-indexes. The LD correlations have been calculated before this point in the code, and are all stored in a lower triangular matrix from here on out called $correlationMatrix$. The code uses these already calculated correlation values to calculate the omega value for a specific location.

The number of times this calculation has to run increases with two factors. If the area of interest in the data set increases it is clear that more locations have to be checked thus more calculations have to be done, but there is a second variable that greatly increases the number of executions. The size of the windows on either side of the sweep in which the LD pattern is calculated significantly changes the produced value of the calculation. If the window size is too small it is possible that not all other mutations that hitchhiked along with the sweep get used in the calculation resulting in a much less pronounced result. However if the window size is too large the calculation includes genome locations that did not hitchhike with the beneficial mutation similarly decreasing the effectiveness of the calculation. Finding the selective sweep is dependant on calculating the ω-statistic with the correct window length. The OmegaPlus algorithm evaluates all possible window sizes and finds the best omega-score. This does mean that the omega-score has to be calculated multiple times per location, increasing the number of calculations. This behaviour of OmegaPlus is user controlled with the input parameters Minwin and Maxwin.

III. Related Work

Decreasing the computation time of DNA sequence analysis software by accelerating the calculation with FPGA's has been explored by for numerous implementations. Both [5] and [6] focus on accelerating the calculation of the LD,

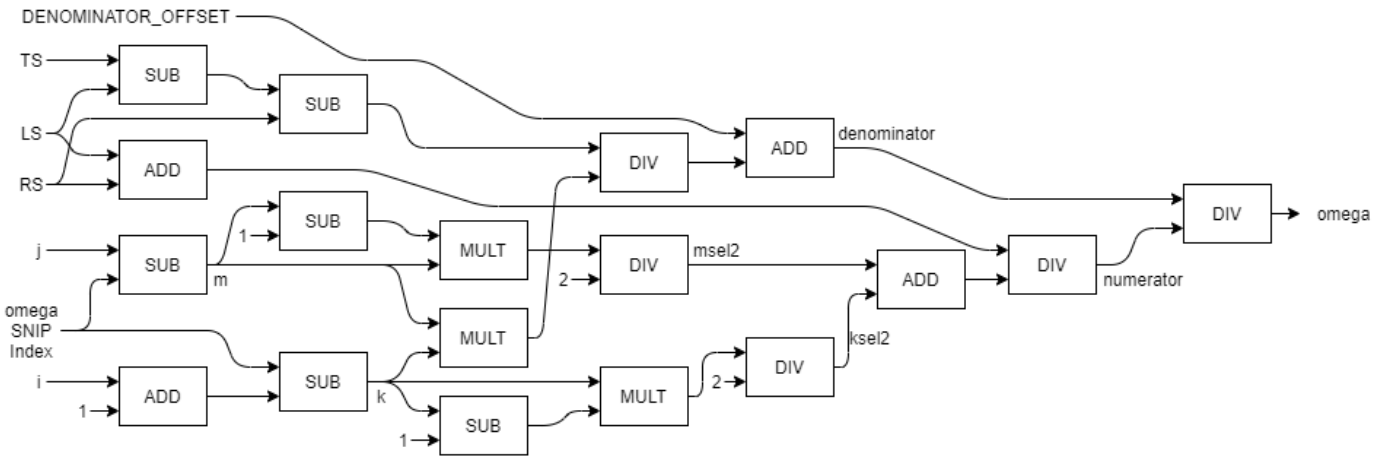

Fig. 2. Block diagram of the expected hardware for the calculation

which is assumed to be already calculated for the accelerator designed in this paper. In [5] it is proven that FPGA's can achieve 50x faster analysis compared to software. In [6] the memory layout is also discussed as an optimization for the accelerator, and the accelerator can achieve 12.7x to 134.9x faster performance compared to the reference software.

Decreasing the computation time of the program can be done using an FPGA, the method presented in this paper, but multi-threading the CPU processor is also an option. The OmegaPlus computational workflow allows for different kinds of multi-threading to be implemented. [7][8]. The OmegaPlus program structure is explained in these papers, and how to implement different kinds of multi-threading in an efficient way. Two different ways of multithreading are implemented in OmegaPlus, fine-grain and course-grain. Fine-grain multi-threading is much like the accelerator presented in this paper. It tries to run the calculations of one iteration of the calculation loop in parallel. Coarse-grain multithreading operates on a higher level, running multiple loop calls in parallel. A multi-grain implementation is also designed, which combines the two types of multithreading for even faster performance. In the multi-grain algorithm threads that finish the assigned loop quicker than others help the slower threads do the calculations.

IV. Architecture

A. Single instance hardware

The $omega$-calculation shown in appendix A is written in C. This code is optimized for synthesizing to HDL to achieve the fastest behaviour. Figure 2 shows the expected hardware for a single instance of the calculation. This accelerator produces an omega value for each location the software requests. This omega value is calculated with a few constants that stay the same for multiple locations, and three entries from the $correlationMatrix$ that change with the location. To store the matrix in an efficient way for the accelerator, it is investigated where in the matrix these three entries are, and how accessing these might be optimized How the matrix is accessed is shown in figure 3.

B. Matrix memory accesses

In the program, the index i selects a column in the matrix, and j selects a row. For the calculation, three arrays are needed: RS, LS, and TS. Suppose the code is at a certain i and j, in the case of figure 3 the darker red circle. The next iteration of j moves the location down one row, to the lighter red circle. The next iteration of i moves the location one column to the left, so from red to yellow. This means that for every i, the accelerator needs a new array of data for TS. The blue column however stays the same when i changes. This means that to save memory accesses, RS can be prefetched and reused throughout multiple iterations of i. Furthermore it makes sense to store the matrix in a column-major order since we need two columns per iteration of i. To explore the limitations of the accelerator the software includes the options to also prefetch TS and LS, but RS is always prefetched.

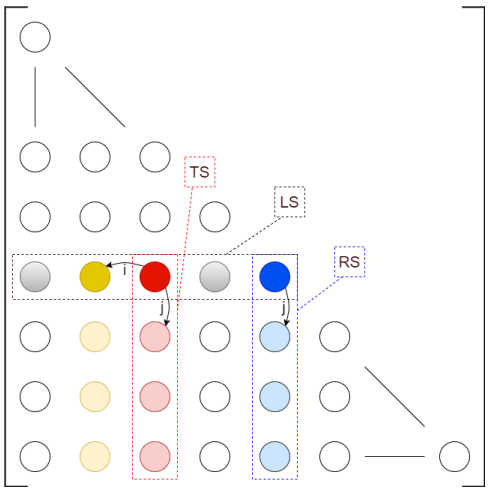

Fig. 3. Explanation on how the $correlationMatrix$ is accessed

C. Unrolling the right side for-loop

The calculation of OmegaPlus consist of a nested for loop, the outer loop looping over the left side values, and the inner loop looping over the right side values. To have control over the amount of hardware that is created for the accelerator, the inner for loop is partially unrolled.

```
//left side loop
for (i = leftMaxIndex; i > leftMinIndex; i--)
{
    //right side loop
    for (j = rightMinIndex; j < rightMaxIndex; j++)
    {
        //Calculation using i and j
        AcceleratorPipeline(i, j);
    }
}

//left side loop
for (i = leftMaxIndex; i >= leftMinIndex; i--)
{
    //right side loop
    for (j = rightMinIndex; j < rightMaxIndex; j += UNROLLFACTOR)
    {
        //unroll loop
        for (u = 0; u < UNROLLFACTOR; u++)
        {
            //Calculation using i and (j+u)
            AcceleratorPipeline(i, (j+u));
        }
    }
}
```

Fig. 4. Pseudo code for how the calculation is unrolled

Figure 4 shows how this change looks in the code. The top of the figure shows the original loop structure that loops over i and j and uses them in the calculation. The bottom shows the loops used in the accelerator. For the accelerator design the right side loop is unrolled with a certain UNROLLFACTOR. This creates a third nested for loop of which the bounds are always known. In the calculations, j has to be replaced with $j + u$ to match this change. When synthesizing the design in Vivado HLS the hardware for the calculation inside the unroll loop is created multiple times, equal to this unroll factor. This allows for the design to change its size based on the board it is going to run on. If the board has more FPGA resources available, the execution time can be reduced further.
The unroll factor does not always perfectly divide the number of right side loop iterations. The leftover iterations that remain after the right side loop iterations are divided by the unroll factorW are executed in software separate from the accelerator.

In the accelerator design, the order of the right side loop and the unroll loop is switched. This allows for the inner loop to be pipelined. In this case the unroll loop determines how many pipelines are created in parallel, still having control over how much data is computed in parallel and how much FPGA resources are used.

D. Processing schedule

The order in which the accelerator executes the calculations is shown in figure 5. The most inner part of the accelerator, the

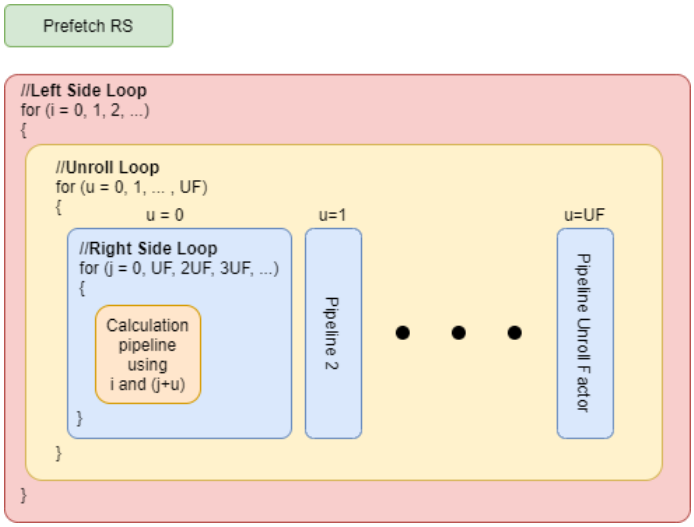

Fig. 5. Overview of the accelerator design

orange shape, is a single instance of the calculation hardware. The blue shapes in the figure are the right side loop, configured as a pipeline with a latency of 61 and an initiation interval of 1. This means that if data is applied to the pipeline, it takes 61 clock cycles to produce the output corresponding to that data, but after 1 clock cycle new data can be applied. The unroll loop, yellow shape in the figure, determines how many pipelines are created in parallel. The left side loop, red shape in the figure, still runs sequentially as it did in the code.

V. Implementation and Verification

Vivado HLS version 2020.1 is used to synthesize the optimized code to HDL. Vivado takes the code and create the accelerator logic based on the added pragma's such as pipeline, and unroll.

A. Test bench

An HLS project needs a source file with the code for the accelerator, and a test bench that compares the results of the code to the results of the hardware to verify correctness of the design. This test bench provides the function with its input and checks the output.
As test scenario, the sequential version of OmegaPlus was run with 3 different size examples. A small one with just under 6,000 iterations for the hardware, a medium with just under 500,000 and a large one with just under 5,000,000 iterations. During this the $correlationMatrix$ and all other function inputs that the hardware needs are stored in a text file. When OmegaPlus did the calculations, all its results are also stored in a text file. The test bench for the accelerator loads these function inputs and the results from OmegaPlus, and then calls the accelerator with the correct inputs. The output of the accelerator is then compared to the output of OmegaPlus to see if the accelerator functions correct.

B. Functional correctness

During testing the outputs of the accelerator might be slightly different to the software due to floating point rounding errors. The test bench checks if the result is within a certain epsilon band of the original OmegaPlus output. This band was defined as 0.001, and the accelerator has always produced an output within this bound.

C. Target devices

The accelerator design in this paper targets two FPGA boards. The Zync UltraScale+ ZCU102 Evaluation Board was chosen as a smaller development type FPGA board. To test the limits of the accelerator, it has also been synthesized to target the Alveo U200 Data Center Accelerator Card. This is a much bigger FPGA, so the design can be much more resource demanding.

VI. Results and Discussion

The throughput of the accelerator can be improved by tweaking the accelerator to the implementation. The unroll factor of the design can be changed, and TS and LS can be prefetched or not. The accelerator also behaves differently for different left- and right-side loop boundaries. This section focuses on changing these parameters and exploring the performance characteristics of the accelerator as a result of the changes. Then the throughput of different versions of OmegaPlus is measured and the performance of the accelerator is compared to the software implementation.

A. Accelerator

1) Left side loop iterations: First the behaviour of the accelerator with regards to the left side loop iterations is investigated. The results can be seen in table I. These results are obtained with a clock frequency of 100MHz and an unroll factor of 1. It can be seen that a change in number of iterations for the left side loop, regardless of the number of iterations for the right side loop, does not make any meaningful difference in terms of throughput. The accelerator unrolls and pipelines the right side loop and there are no optimizations in the left side loop so this behaviour is expected.

TABLE I
ACCELERATOR THROUGHPUT (IN MILLION SCORES PER SECOND) BASED ON DIFFERENT RIGHT- AND LEFT- SIDE LOOP ITERATIONS

Left side loop iterations	Right side loop iterations			
	100	1000	5000	10000
100	62.06	94.14	98.67	99.27
1000	62.11	94.24	98.78	99.38
5000	62.11	94.25	9879	99.39
10000	62.11	94.25	98.79	99.39

2) Unroll factor: Changing the unroll factor of the design changes the throughput because more calculations can be done in parallel. Increasing the unroll factor with a certain factor theoretically also increases the throughput with that same factor, however at a certain unroll factor the clock frequency of the FPGA has to be decreased for the design to be feasible. Figure 6 shows how the throughput changes with the unroll factor. The decrease in clock frequency is also taken into account in this figure. These results are obtained with one million iterations for the left- and right-side loop.

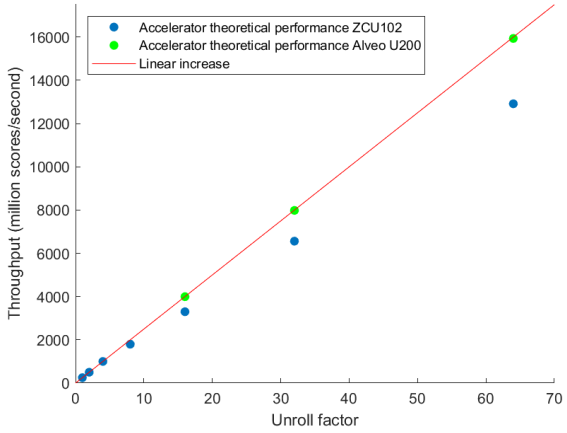

Fig. 6. Throughput for different accelerator platforms for different unroll factors

In figure 6, the blue data represents the accelerator running on the Zync UltraScale+ ZCU102 Evaluation Board. The green data represents the accelerator running on the Alveo U200 Data Center Accelerator Card. The red line represents a linear increase where the performance increases with the same factor as the unroll factor.

It can be seen that the accelerators throughput keeps increasing with a higher unroll factor. At lower unroll factors the performance increase seems linear. At higher unroll factors however the increase becomes slightly less. This is due to a few reasons. The clock frequency that the FPGA can achieve with the design starts dropping off at a certain unroll factor. Furthermore the calculations can run more in parallel with a higher unroll factor, but that also means that the pipeline latency plays a bigger part in the total latency compared to less parallel computation. The throughput for the ZCU102 board drops off quicker than the throughput for the Alveo U200. This is because the latter is able to reach higher clock frequencies at the same unroll factor due to its different design. At an unroll factor of 64 the U200 can operate on a 250MHz clock. In this situation, the performance decrease compared to the linear increase is 0.38%. The clock frequency of the ZCU102 however has to be reduced to 202.59MHz for an unroll factor of 64 according to the Vivado HLS synthesis results. This is a decrease of 18.96% and results in a decreased performance of 19.28% compared to the linear increase.

The highest unroll factor that was tested is 64, because the memory system needed for this unroll factor has to provide the design with 51.2GB/s. This is already quite substantial and not feasible on smaller development type

FPGA boards. Therefore, from here on this paper focuses on two design approaches for the accelerator. One design targets the ZCU102, the other targets the Alveo U200. For the ZCU102 the highest possible unroll factor is 4. This design has a throughput of 985.28 million scores per second and the memory subsystem needs to provide the accelerator with 3.2GB/s of data. For the design for the Alveo U200 platform an unroll factor of 32 is suitable. This gets the throughput of the accelerator up to 7762.23 million scores per second and the design needs 25.6GB/s data input from the memory system.

TABLE II
UTILIZATION OF FPGA RESOURCES FOR THE ZCU102 IMPLEMENTATION

	available	design	percentage
BRAM_8K	1824	36	1.97
DSP48E	2520	48	1.90
FF	548160	12003	2.19
LUT	274080	12847	4.69

TABLE III
UTILIZATION OF FPGA RESOURCES FOR THE U200 IMPLEMENTATION

	available	design	percentage
BRAM_8K	4320	40	0.93
DSP48E	6840	215	3.14
FF	2364480	50841	2.15
LUT	1182240	50584	4.28

3) Resource utilization: The FPGA resource utilization of the ZCU102 and U200 designs is shown in table II and III respectively. It can be seen that the utilization is very low, thus the performance bottleneck of the accelerator design is the memory system bandwidth.

4) Right side loop iterations: Lastly, the behaviour of the accelerator with regards to the right side loop iterations is investigated. This is done for the two unroll factors found in section VI-A2. The results are shown in figures 7 and 8 for the ZCU102 and Alveo U200 designs respectively.

It can be seen that the throughput of the accelerator converges to a maximum based on the number of right side loop iterations. The accelerator performs considerably worse for implementations with a smaller number of iterations for the right side loop, but increases its performance significantly for more iterations. The black dotted lines added in both figures represent 90% of the the theoretical maximum throughput of both designs. This theoretical maximum is the ideal assumed scenario with an infinite number of iterations for the right side loop. This would mean that the pipeline latency can be neglected, and that the accelerator can produce one output per pipeline on each clock cycle. In practice this maximum can never be reached because of the pipeline latency of the design, and the time needed to prefetch RS. The lines at 90% are a good point from which the accelerator works as intended. With fewer right loop iterations the performance of the accelerator is still acceptable in some cases, but the design of the accelerator

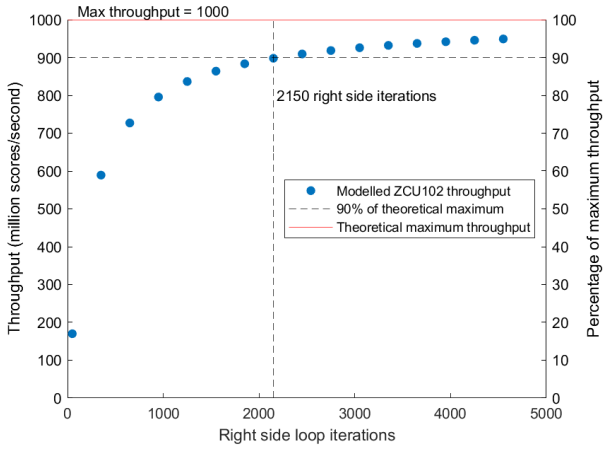

Fig. 7. Throughput as a function of right side loop iterations for the ZCU102 targeted implementation

could be improved for these situations. It should be noted that the maximum hard-coded number for right side loop iterations in OmegaPlus is 10000, so the design for the U200 accelerator can not reach its full potential.

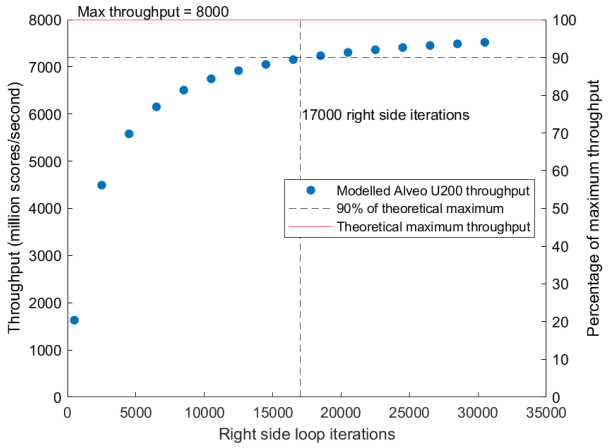

Fig. 8. Throughput as a function of right side loop iterations for the Alveo U200 targeted implementation

B. OmegaPlus results

To compare the accelerator performance with the software versions of OmegaPlus, a few test with the different versions of the software have been done. All these test were done with OmegaPlus v3.0.3 and run on an Intel(R) Core(TM) i7-6700HQ CPU at 2.6GHz.

1) Sequential version of OmegaPlus: The throughput of the sequential version of OmegaPlus was investigated with different calculation sizes. The results can be seen in table IV. These results are obtained with input parameters $Length = 100000$ and $Minwin = 10000$ The average measured throughput of the sequential version of OmegaPlus

TABLE IV
SEQUENTIAL OMEGAPLUS THROUGHPUT FOR DIFFERENT INPUT PARAMETERS

Grid	Maxwin	Total calculations	Right side iterations	Throughput (million scores/second)
100	25000	69,205,728	862	113.97
500	25000	348,791,291	858	113.71
1000	25000	698,272,448	858	118.38
100	50000	303,744,128	1924	97.50
500	50000	1,531,523,490	1920	100.77
1000	50000	3,066,265,911	1919	100.83
100	100000	404,062,473	2763	99.35
500	100000	2,037,721,845	2766	101.49
1000	100000	4,079,729,213	2766	101.74

is 105,303,707 scores per second. It can be observed in the table that the throughput of OmegaPlus is slightly higher when calling it with a smaller Maxwin parameter.

2) Multi-threaded version of OmegaPlus: To investigate the performance increase that OmegaPlus experiences when it is called with more threads, a large example was run and the time of calculations was measured. The input parameters that were used are $Grid = 1000$, $Length = 100000$, $Minwin = 10000$, and $Maxwin = 100000$. This resulted in $4,079,729,213$ calculations that needed to be done. These results can be seen in table V in the appendix and in figures 9 and 10. Note that the throughput scale for the second is logarithmic. The CPU on which this test was done has 4 cores. When OmegaPlus is called with up to 4 threads, each thread is run on its own core and has about 100 million scores per second throughput, similar to the results found for the sequential version of OmegaPlus. This results in linear performance increase for the software. When increasing the threads beyond the number of cores the performance increases slightly due to hyper-threading, but the behaviour is not linear anymore. Figures 9 and 10 both show the throughput of OmegaPlus compared to the two accelerator designs.

C. Compare results

In figures 9 and 10 it can be seen that the performance of the accelerators is better than OmegaPlus. The green line represents the throughput of the ZCU102 and the Alveo U200 in the case of figure 9 and 10 respectively. These throughput numbers for the accelerator are based on 2750 right side loop iterations. This is because the average iterations OmegaPlus had for this example is always above this bound. The performance of the accelerators should be compared to the multi-threaded version of OmegaPlus. The ZCU102 accelerator resulted in an 2.12x performance increase, and the Alveo U200 was a 10.80x faster. It should be noted however that the accelerator design for the Alveo U200 benefits a lot from more right side loop iterations, as can be seen in figure 8. As stated in section VI-A4 the maximum number of iterations for the right side loop is 10000. Taking this as the average number of right side loop iterations increases the performance of the U200 accelerator to 26.53x faster than OmegaPlus. Increasing this number even further to 50000 iterations results in an 34.20x performance increase.

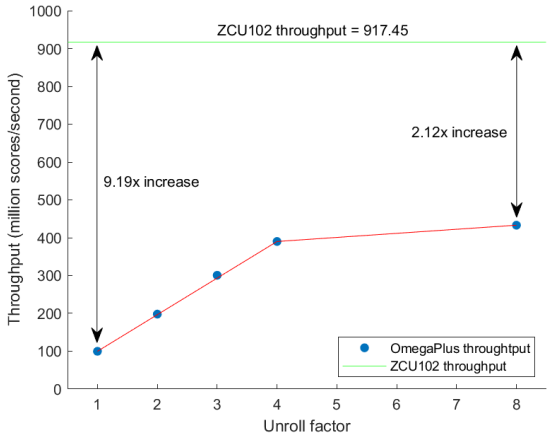

Fig. 9. Throughput of OmegaPlus compared with the ZCU102 accelerator design

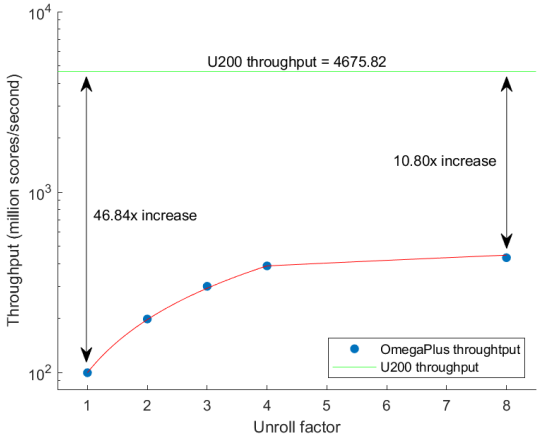

Fig. 10. Throughput of OmegaPlus compared with the Alveo U200 accelerator design

VII. CONCLUSION AND FUTURE WORK

From the results it becomes clear that the throughput of the accelerator increases with the size of the calculation, specifically the right side loop iterations. At the same time if the calculation size gets bigger the throughput of OmegaPlus decreases a bit. This means that using one of the designed accelerators becomes more interesting when the calculation size increases. The designed accelerators can finish the calculations up to 10.80x faster than OmegaPlus.

Since the performance of the accelerator increases a lot with number of right side loop iterations, it could be investigated if the hard-coded limit of 10000 that OmegaPlus currently features can be increased. This would be especially useful for larger implementations with a high unroll factor.

REFERENCES

[1] Pavlidis, P., Alachiotis, N. A survey of methods and tools to detect recent and strong positive selection. J of Biol Res-Thessaloniki 24, 7 (2017). https://doi.org/10.1186/s40709-017-0064-0

[2] Alachiotis, N., Pavlidis, P. RAiSD detects positive selection based on multiple signatures of a selective sweep and SNP vectors. Commun Biol 1, 79 (2018). https://doi.org/10.1038/s42003-018-0085-8
[3] Crisci Jessica, Poh Yu-Ping, Mahajan Shivani, Jensen Jeffrey, The impact of equilibrium assumptions on tests of selection, Frontiers in Genetics, 2013, https://www.frontiersin.org/article/10.3389/fgene.2013.00235, DOI:10.3389/fgene.2013.00235
[4] Yuseob Kim and Rasmus Nielsen GENETICS July 1, 2004 vol. 167 no. 3 1513-1524; https://doi.org/10.1534/genetics.103.025387
[5] Nikolaos Alachiotis and Gabriel Weisz. 2016. High Performance Linkage Disequilibrium: FPGAs Hold the Key. In Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '16). Association for Computing Machinery, New York, NY, USA, 118–127. DOI:https://doi.org/10.1145/2847263.2847271
[6] D. Bozikas, N. Alachiotis, P. Pavlidis, E. Sotiriades and A. Dollas, "Deploying FPGAs to future-proof genome-wide analyses based on linkage disequilibrium," 2017 27th International Conference on Field Programmable Logic and Applications (FPL), Ghent, 2017, pp. 1-8, doi: 10.23919/FPL.2017.8056814.
[7] Alachiotis N, Pavlidis P. Scalable linkage-disequilibrium-based selective sweep detection: a performance guide. Gigascience. 2016;5:7. Published 2016 Feb 8. doi:10.1186/s13742-016-0114-9
[8] Alachiotis N, Pavlidis P, Stamatakis A, Exploiting Multi-grain Parallelism for Efficient Selective Sweep Detection (2012). http://popgen.eu/wordpress/wp-content/uploads/2019/07/Alachiotis-Pavlidis-Stamatakis-2012-Exploiting-multi-grain-parallelism-for-efficient-selective-sweep-detection.pdf
[9] Alachiotis N, Pavlidis P, Stamatakis A, OmegaPlus v3.0.3 (2018) [Computer software], https://cme.h-its.org/exelixis/web/software/OmegaPlus/index.html
[10] XILINX VITIS, https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html

APPENDIX

A. Omega-statistic calculation in code

```
for (int i = leftMinIndex; i >= leftMaxIndex; i--) // Left Side
{
    float LS = correlationMatrix[omegaSNIPIndex][i];

    int k = omegaSNIPIndex - i + 1;

    int ksel2 = (k * (k - 1)) / 2;


    for (int j = rightMinIndex; j <= rightMaxIndex; j++) // Right Side
    {
        float RS = correlationMatrix[j][omegaSNIPIndex + 1];

        int m = j - omegaSNIPIndex;

        int msel2 = (m * (m - 1)) / 2;

        float TS = correlationMatrix[j][i];

        float tmpW = computeOmega(LS, RS, TS, k, ksel2, m, msel2);

        if (tmpW > maxW)
        {
            maxW = tmpW;
            maxLeftIndex = i + omega[omegaIndex].leftIndex;
            maxRightIndex = j + omega[omegaIndex].leftIndex;
        }
    }
}
```

```
float computeOmega(float LS, float RS, float TS, int k, int ksel2, int m, int msel2)
{
    float   numerator = (LS + RS) / (ksel2 + msel2);

    float denominator = (TS - LS - RS) / (k * m) + DENOMINATOR_OFFSET;

    float omega = numerator / denominator;

    return omega;
}
```

B. Multithreaded OmegaPlus throughput

TABLE V
THROUGHPUT OF OMEGAPLUS WITH MULTIPLE THREADS

	Threads	Right side loop iterations	Throughput (million scores/second)
1 thread total	**1**	**2766**	**99.83**
2 thread total	**2**	2766	**198.05**
thread 1	2.1	2766	103.07
thread 2	2.2	2766	99.03
3 thread total	**3**	**2766**	**300.70**
thread 1	3.1	2763	100.22
thread 2	3.2	2764	100.43
thread 3	3.3	2762	100.44
4 thread total	**4**	**2766**	**390.00**
thread 1	4.1	2764	97.71
thread 2	4.2	2769	98.41
thread 3	4.3	2772	98.37
thread 4	4.4	2761	97.52
8 thread total	**8**	**2766**	**433.08**
thread 1	8.1	2766	54.90
thread 2	8.2	2770	54.94
thread 3	8.3	2764	54.67
thread 4	8.4	2757	54.66
thread 5	8.5	2751	54.31
thread 6	8.6	2767	54.75
thread 7	8.7	2761	54.17
thread 8	8.8	2782	54.67