University of Twente

Master thesis

Hardware acceleration of sweep detection using Clash.

Computer Architecture for Embedded Systems

By Lucas Bollen

Graduation Committe:

Chairman	dr.ir. S.H. Gerez
Supervisor 1	dr.ir. N. Alachiotis
Supervisor 2	ir. H.H. Folmer
External Member	dr.ir. V. Zaytsev

August 27, 2021

Abstract

With technologies for whole genome sequencing becoming more affordable and common place in bio-informatics, there has been an increasing interest in processing genomes to extract valuable information. The hitchhiking effect is the tagging along of neighbouring mutations when a beneficial mutation is selected due to natural selection. This leads to a local reduction in genetic variation known as a selective sweep. Multiple tools have been developed that can process whole genome data to detect selective sweeps, each with their pros and cons. Sweep Detector [1] finds the location of the most likely recent selective sweep based on changes in the Site Frequency Spectrum (SFS). As shown during the SARS-CoV-2 pandemic, detecting and localizing selective sweeps can help us track the evolution of viruses and answer important questions regarding its evolution. One of the main problems with selective sweep detection is that depending on data set size and required precision, processing data sets is very time consuming. This thesis presents a flexible device independent hardware design for FPGAs that can aid in significantly accelerating selective sweep detection.

A combination of profiling and manual analysis was used to determine which part of Sweep Detector should be accelerated. Sweep Detector makes use of double precision floating point operators which require lot of resources on FPGA, we found that up to 64 bit fixed point values could not be used as replacement. However, Using single precision floating point operators did not significantly affect the precision of Sweep Detector. The accelerator heavily relies on data access to arrays that can not be stored on chip, for this thesis we assumed ideal memory to first focus on the design of the accelerator. The likelihood calculations performed by Sweep Detector contain four selected loops which contain loop carried dependencies and data dependent repetition. Loop carried dependencies make it impossible to concurrently process multiple iterations of the same loop and data dependent repetition makes batch processing of iterations inefficient. This has been solved by unrolling the outermost loop which contains neither of these properties and by combining custom variations of the merge and branch circuit presented by Styles et al. [2] with a decentralised control method that aims for maximum throughput of this accelerator regardless of configuration.

By utilizing the high level abstractions offered by Clash we created a parametrized design that allows us to easily control the level of parallelism in the design based on the resources available on the target device to maximize the performance. Synchronisation of data is ensured on a type level such that the user does not need to worry about synchronisation issues. Furthermore a decentralised control method is applied that aims to maximize the throughput of this accelerator regardless of configuration. The design was configured for a Cyclone V FPGA that allowed for 14 times parallelism, compared to the C implementation we found a maximum theoretical speedup of up to 93 times for the same functionality on a single core Xeon Gold 6126 processor. At this rate the accelerator would be processing 195 gigabits of data per second from these large arrays, further showing dire need for a high performance tailored memory management system.

Contents

List of Figures

List of Tables

Abbreviations

ALM	Adaptive Logic Module
CLR	Compisite Likelihood Ratio
DFG	Data Flow Graph
DNA	Deoxyriboneoclic Acid
DNB	Dynamic Binary Instrumentation
DSP	Digital Signal Processing
FIFO	First In First Out
FPGA	Field Programmable Gate Array
GPU	Graphics Processing Unit
HDL	Hardware Description language
HLS	High Level Synthesis
IP	Intellectual Property
LCD	Loop Carried Dependency
LD	Linkage Disequillibrium
PVT	Process, Voltage, Temperature
RAiSD	Raised Accuracy in Sweep Detection
REPL	Read Evaluate Print Loop
RNA	Ribonucleic Acid
RSQ	Rate Smoothing Queue
RTL	Register-Transfer Level
SFR	SNP Frequency Range
SFS	Site Frequency Spectrum
SNP	Single Nucleotide Polymorphism
SweeD	Sweep Detector
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit

1 Introduction

1.1 General introduction and Problem statement

With the technology of whole genome sequencing becoming more affordable and widespread over time, processing this type of data (whole genome sequences) becomes more and more important. One of the ways that we can use this technology is to detect and localize recent mutations [4] [1]. As of writing, the world is at the mercy of the global SARS-CoV-2 pandemic. Detecting mutations in the RNA of SARS-CoV-2 can help to monitor the evolution of the virus [5] and keep track of different strains [6]. When a beneficial mutation occurs, it will become more frequent in the population due to natural selection. This increase in frequency will locally reduce genetic variation among the population due to the hitchhiking effect [7]. The hitchhiking effect is the hitchhiking, or tagging along, of genetic information that surround a beneficial mutation as the beneficial mutation spreads among the population due to natural selection. The local reduction in genetic variation that can be observed as consequence of a beneficial mutation spreading among the population is called a "selective sweep".

Detecting selective sweeps can be used to answer important questions regarding human evolution over a longer period of time [8], but can also give insights in human evolution present in current day and age [9]. The detection of selective sweeps on whole genome data is a relatively new field and few tools have been developed that can perform this task. Nielsen et al. [4] developed a method of detecting these selective sweeps in whole genome data and implemented it in a tool called "SweepFinder" which processes whole genome data and calculates the location of the most probable selective sweep among the genome. SweepFinder proved to be a useful tool, but contains numerical instabilities that severely limited the maximum samples a data set could have to be processed by SweepFinder [1]. Besides the fact that SweepFinder contained instabilities it is also relatively slow [1].

SweepFinder was the first tool that could be used to process whole genome data, but since its release multiple new tools have been created that are capable of selective sweep detection in large data sets. A new effort to create an improved implementation of SweepFinder was done in 2013 and resulted in Sweep Detector (SweeD) [1]. This new effort fixed the numerical instabilities and showed improved performances compared to SweepFinder, speedups up to 22x were achieved, but the performance improvement significantly decreased as the length of the genomes grew (down to 2.3x faster than SweepFinder).

Where SweepFinder and SweeD both rely on changes in the Site Frequency Spectrum (SFS), other methods of detection selective sweeps already exist. Tools such as OmegaPlus [10] rely on the correlation between different mutations to detect selective sweeps. A recurring theme in the researches that present tools for selective sweep detection is the fact that due to the large data set sizes and relatively complex calculations, running these tools can take a long time. Pavlidis et al. [11] compared a multitude of tools and found that for large scale analysis, high performance computing solutions are required to reduce computation times. One of the most recent tools that was released for detection positive selections is RAiSD [12], which makes use of a new method for sweep detection that requires significantly less computational power. The authors of RAiSD later presented RAiSD-X [13], which is a hardware accelerated version of RAiSD that reported execution times over a

thousand times lower than SweeD. This system was designend with Xilinx Vivado HLS. While SweeD is still faster than SweepFinder, processing big data sets of large genomes can take an incredible long time with runtimes up to 8.25 hours measured [1]. In this report we will explore a different method of creating flexible hardware accelerators by accelerating SweeD using the functional HDL Clash [14] (previously known as Cλash). We chose to accelerate SweeD because of the fact that has been widely adopted and the fact that its low false positive rate makes it a valuable addition to the bio-informatics toolbox. Furthermore acceleration of SFS based sweep detection methods have not been attempted before.

1.2 Proposed solution

The aim of this project is to create a flexible design for accelerating selective sweep detection. For this research we will focus on FPGA implementations. Given the fact that there are many different FPGAs with different kinds and amounts of resources, the flexibility should allow the user to quickly tailor their design to the available hardware. As a case study we will accelerate SweeD because SFS based selective sweep detection requires a lot of computational power and has not been accelerated before. Furthermore SweeD contains a lot of repetition which is a great indicator that dedicated hardware could speed up the process significantly. We accelerate SweeD by off-loading repetitive calculations to a platform that is capable of mass parallel computation such as GPU or FPGA.

Given the mathematical nature of the problem at hand, we used the functional programming language Clash. Using Clash has multiple advantages such as the fact that the functional nature of Clash is closer to mathematics than conventional HDLs or other programming languages such as C that are often used for high level synthesis. Also Clash offers high level functionality such as polymorphic typing, pattern matching and higher order functions which makes it easier to create flexible designs. Usually testing hardware designs is a very time consuming process, but Clash offers an interactive environment that can be used to very quickly prototype and test functionality.

1.3 Research questions

This section describes the research questions that provide guidance to the project and are aimed at the most important aspects of the results. The first and foremost goal is to design a hardware accelerator for selective sweep detection that can be used for any FPGA. It is not possible to consider every FPGA, so we try to stay independent of specific technology solutions to prevent e.g. vendor lock-in. This hardware accelerator should deliver a significant speedup compared to the already existing C implementation.To achieve this we would need to maximize the throughput of the design. Increasing the throughput of FPGA designs is mostly done through introducing parallelism in the design. The amount of parallelism that can be introduced depends mostly on the available resources. If we can reduce the amount of resources required, we can introduce more parallelism. The aspects that we just discussed are captured in the research questions presented below:

1. How can we design a technology-independent hardware accelerator that significantly speeds up SweeD using Clash?

    a Which part of the algorithm should be accelerated?

   b How can we reduce the area of the implementation without significantly impacting the precision?

2. How can we create flexibility in our design such that it is easily adjustable for different target devices?

   a Where can we introduce extra parallelism in the design?

   b Which parameters can we offer to easily tune the design?

1.4 Outline

Chapter two discusses the background information regarding genetics, sweep detection with SweeD, Clash and method of pipelining nested loops with LCDs that is necessary to understand the rest of the work. Chapter three presents the literature study regarding the related work with other methods of sweep detection and hardware acceleration for sweep detection. Chapter four contains the design space exploration where we discuss how we determined what to accelerate, how the memory requirements of the likelihood calculations interfere with the accelerator, what the memory access patterns look like and how we determined that we use single precision floating point rather than fixed point instead of double precision floating point. Also discusses options regarding how we introduce flexibility in our design. Chapter five discusses the implementation, how we use Clash for packaging data and ensuring the related variables are synchronised. Discusses how we implemented floating point operators. It also discusses the proposed memory access method. Chapter six discusses how we tested the functional correctness of the accelerator and how we adjusted the design for a target FPGA, maximizing the parallelism that can be fitted on the device. It discusses the maximum theoretical speedup that can potentially be gained with this device under the assumption of ideal memory. Chapter seven discusses the testing methods and results. Chapter eight presents the conclusion by answering the research questions and chapter nine recommends additions and changes to the method.

2 Background

This section discusses the background information necessary to interpret this work. Section 2.1 discusses all relevant information about genetics and section 2.2 describes how SweeD processes whole genome data to perform selective sweep detection. Section 2.3 introduces Clash and presents some simple examples of hardware design in Clash. Lastly Section 2.4 presents a method of pipelining nested loops, which is partially adopted in this work.

2.1 Genetic Background

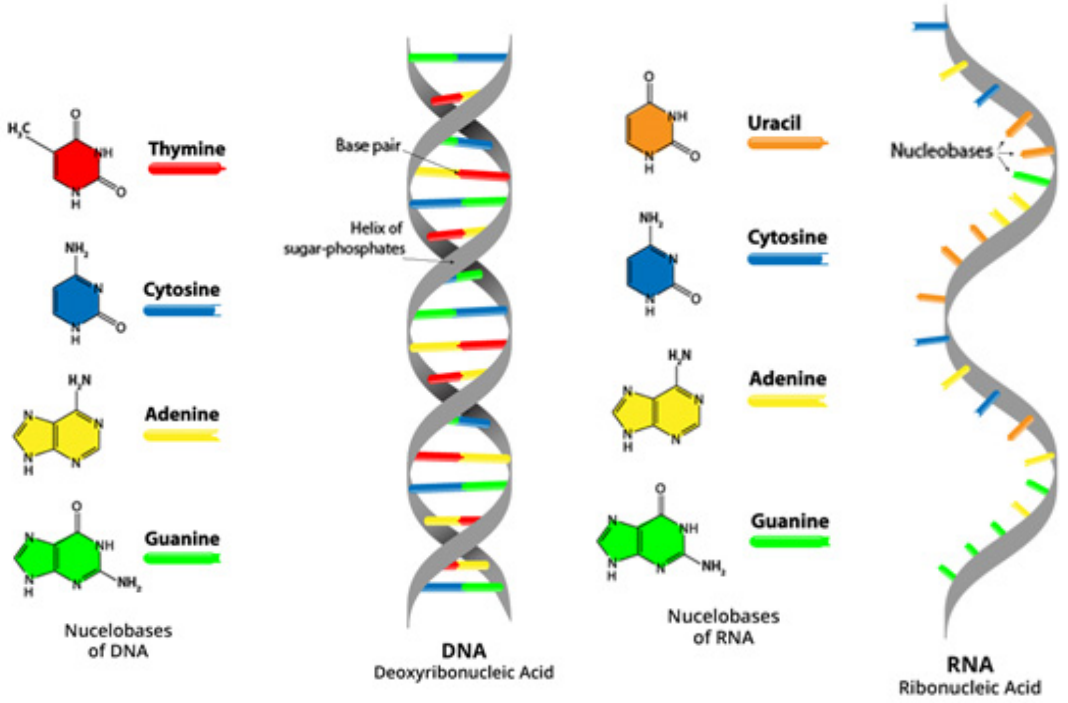
2.1.1 DNA, RNA and genomes

All genetic information that describes what an organism looks like is called the organism's genome and is stored in the form of DNA (deoxyribonucleic acid). Together with RNA (ribonucleic acid) and proteins, DNA is one of the three major macro molecules that are essential for life. DNA consists of 2 strains of genetic bases. Each strain is a sequence of different bases. The bases that occur in DNA are: Cytosine, Guanine, Adenine and Thymine, typically referred to by C, G, A and T, respectively. RNA Consists of a single strain made up of the bases Cytosine, Guanine, Adenine and Uracil where Uracil is typically referred to with U. The strains in DNA form a double helix structure where the bases of the two strains form pairs. Two different pairs are possible, A with T and G with C. Figure 1a shows the DNA bases and figure 1b shows visually the helical DNA structure where the individual bases are visible. Figure 1c shows the RNA bases and figure 1d shows visually the single strain of RNA with visible bases.

2.1.2 Mutations, SNP Data and selective sweep

A mutation is a change that occurs in a DNA or RNA sequence and can have a variety of causes. These changes introduce genetic variation and can have different effects. Mutations can have a negative effect, a positive effect or neutral effect based on where they happen in the DNA / RNA sequences. A positive mutation is a mutation that enables the species to better survive in its environment. A negative mutation would have the opposite effect where the individual with this mutation has a lower chance of survival in its environment. A large portion of our DNA is non-functional and does not encode for anything. When a mutation occurs in this non-functional DNA it has no effect and we call it a neutral mutation. Currently one of the biggest driving forces in human natural selection is the malaria disease [15]. Hedrick et al. [15] gave an insight in how positive mutations in humans lead to higher malaria resistance, increasing the chances of survival.

When such a positive mutation occurs and it increases chances of survival, a selective sweep can occur. Figure 2 shows neutral mutations in blue and a beneficial mutation in red. It is clearly visible how the beneficial mutation spreads among the population. As can be observed from the figure, the neighbouring genetic information is also spreading among the population together with the mutation. This effect is known as hitchhiking or genetic draft [7]. The consequence of hitchhiking is a local reduction in genetic variation among the population. A selective sweep is the region in the DNA that is affected by hitchhiking as a consequence of a beneficial mutation spreading through a population.



(a) DNA Bases (b) Strain of DNA (c) RNA Bases (d) Strain of RNA

Figure 1: Visual representation of DNA, RNA and their bases, adapted from R. Mackenzie [3]

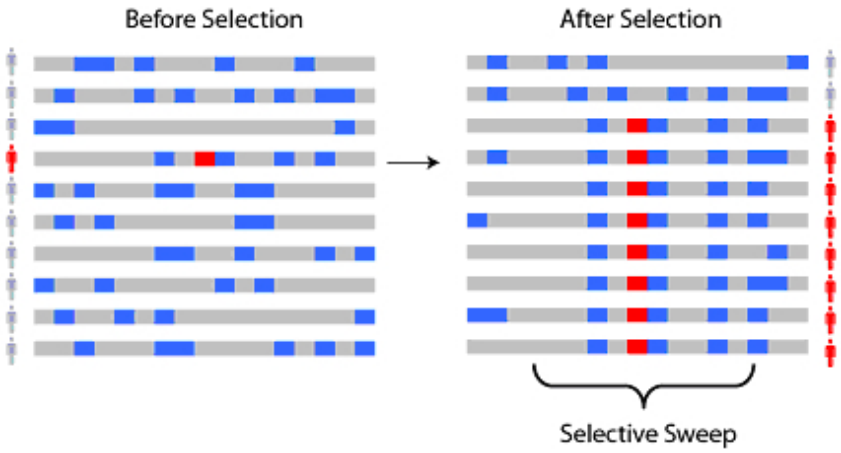


Figure 2: Visual representation of how hitchhiking occurs when a beneficial mutation (red) spreads among a population.

When we compare genetic sequences from different people to each other, we'll notice that our DNA is similar, but has subtle differences. Most of us have two arms, two legs, a human-like face, but what these exactly look like is vastly different. Genetic variation among individual humans today is about 0.1% on average [16]. The positions in our genetic sequences where we differ as a population are called Single Nucleotide Polymorphisms, commonly referred to as SNPs (Pronounced "snips"). When we compare a current population to a genetic ancestor, we can encode the

differences in binary. In this binary representation a '0' is the same as the ancestral reference and '1' means that the individual differs on this position from the ancestral reference. Table 1 shows a small data set with an ancestral reference and 3 individuals. Table 2 shows the same data set, but now sites of the derived individuals are encoded in binary. We can determine the SNP frequencies and the amount of samples included per site. We keep track of the amount of samples per site in case of missing or misaligned data. Data sets greatly differ in size, varying from thousands, to tens of thousands to millions of sites per genome.

Sites ->	1	2	3	4	5	6	7	8	9	10	11	12
Reference	AT	GC	GC	AT	GC	AT	AT	CG	AT	CG	CG	CG
Individual 1	AT	AT	AT	CG	GC	AT	AT	CG	AT	AT	AT	CG
Individual 2	AT	GC	GC	AT	GC	AT	CG	CG	AT	AT	AT	CG
Individual 3	AT	AT	AT	CG	GC	CG	AT	AT	AT	CG	CG	AT

Table 1: Example of a small genetic data set consisting of 3 individuals and a reference with 12 SNPs

Sites ->	1	2	3	4	5	6	7	8	9	10	11	12
Reference	AT	GC	GC	AT	GC	AT	AT	CG	AT	CG	CG	CG
Individual 1	0	1	1	1	0	0	0	0	0	1	1	0
Individual 2	0	0	0	0	0	0	1	0	0	1	1	0
Individual 3	0	1	1	1	0	1	0	1	0	0	0	1
Frequencies	0	2	2	2	0	1	1	1	0	2	2	1
Samples	3	3	3	3	3	3	3	3	3	3	3	3

Table 2: Binary representation of small data set capturing genetic differences

2.2 Sweep Detection with Sweep Detector

Researchers are interested in finding selective sweeps among certain populations. Multiple algorithms have been developed to detect these selective sweeps, but only few of these algorithms are designed for analyzing whole genome data. One that is designed for analyzing whole genome data is Sweep Detector (SweeD) [1]. SweeD is an open-source tool for detection of selective sweeps in whole genomes. This tool is based on the widely used SweepFinder program [4]. SweepFinder is limited to analyzing up to 1,027 sequences. SweeD is a scalable implementation of SweepFinder that allows for analyzing thousands of genomes. However, the speedup of SweeD (with respect to SweepFinder) with datasets of genomes with 1,000,000 SNPs is significantly lower (2,9x instead of 22x) compared to data sets of genomes with 10,000 SNPs. Currently, running the SweeD algorithm can take a long time depending on the amount of samples and the size of these samples.

SweeD uses the method presented by Nielsen et al. [4] to detect selective sweeps. This method is based on considering the way the spatial distribution of frequency spectra along the chromosome is affected by a selective sweep. Before a selective sweep, the site frequency spectrum of the population will be $\mathbf{p} = (p_1, p_2, ...p_{n-1})$. As a first approximation, a selective sweep can be modeled by assuming that each

ancestral lineage has an independent and identically distributed probability of escaping a selective sweep (probability P_e). Based on this, Nielsen et al. [4] concluded that when a beneficial mutation occurs on a chromosome carrying a particular copy of a neutral allele, 1 - P_e is the expected frequency of descendants from that neutral copy at the end of selective sweep. Nielsen et al. [4] shows that the probability P_e has the following functional form:

$$P_e = 1 - e^{-\alpha d}, \tag{1}$$

where α depends on the rate of recombination, effective population size and the selection coefficient for the selected mutation and d is the distance from the location of the sweep to the sampled SNP location. By multiplying these we obtain the alphadistance αd. The probability that k lineages escape the selective sweep is given by the binomial distribution:

$$P_e(k) = \binom{n}{k} P_e^k (1 - P_e)^{n-k}. \tag{2}$$

In the case that k lineages escape the sweep, the ancestral sample right before the sweep contains $H = min\{n, k+1\}$ lineages. Consider that the site frequency spectrum before a selective sweeps is $\mathbf{p} = (p_1, p_2, ..., p_{n-1})$ then the probability of observing j mutant lineages in an ancestral sample of size H is

$$p_{j,H} = \sum_{i=j}^{n-1} p_i \frac{\binom{i}{j}\binom{n-i}{H-j}}{\binom{n}{H}}. \tag{3}$$

If in a sample size of k+1 lineages, there are j mutant lineages with the derived mutation, then the probability that the most recent common ancestor of the lineages that did not escape the selective sweep is of the mutant type, is $j/(k+1)$. This implies that observing a mutant site of frequency B out of n in the sample after a selective sweep is

$$p_B = P_e(n) p_B + \sum_{k=0}^{n-1} P_e(k) \left(p_{B+1-n+k,k+1} \frac{B+1-n+k}{k+1} + p_{B,k+1} \frac{k+1-B}{k+1} \right). \tag{4}$$

For a more elaborate explanation regarding the formulas we refer to Nielsen et al. [4]. The expression above enables us to process whole genome data and calculate the composite likelihood ratio (CLR) for a location on the genome by assuming a value of α. SweeD scans a location for a selective sweep by determining an initial lower bound (minAlpha) and upper bound (maxAlpha) for α and calculating the CLR for a certain amount of α values in between minAlpha and the maxAlpha. It keeps track of the α value for which the returned CLR is the highest and readjusts minAlpha and maxAlpha to encapsulate all possible α values that could lead to the highest CLR. This method is applied iteratively until the boundaries converge to a single value for α for which the CLR is the highest.

Figure 4 shows an example of 3 iterations of finding the highest CLR within alpha boundaries, every tick represents the calculation of a CLR for a certain alpha, but only the highest result for the iteration is shown. Calculating the CLR for a position on the genome with a given α is done by calculating the probability p_b of observing an SNP with frequency b at alphadistance αd from the position for which we calculate the CLR. For each calculated probability, SweeD takes $log(p_b)$, corrects for a base likelihood and accumulates the results. The amount of SNPs for which p_b is calculated depends on the alphadistance αd from the SNP to the location for which the likelihood is being calculated ($\alpha d <= 12.0$). Figure 3 shows an example of the SNPs for which p_b is calculated with $\alpha = 3.14$. Since p_b is calculated for the same combination of n and B many times and αd is within a limited range, the results of p_b are precalculated and stored in a lookup table. Since αd is a real number, it is not suitable for indexing in a lookup table. Instead, the lookup table stores 300 values for different discrete αd values and linear interpolation is used to obtain the result for a certain αd.

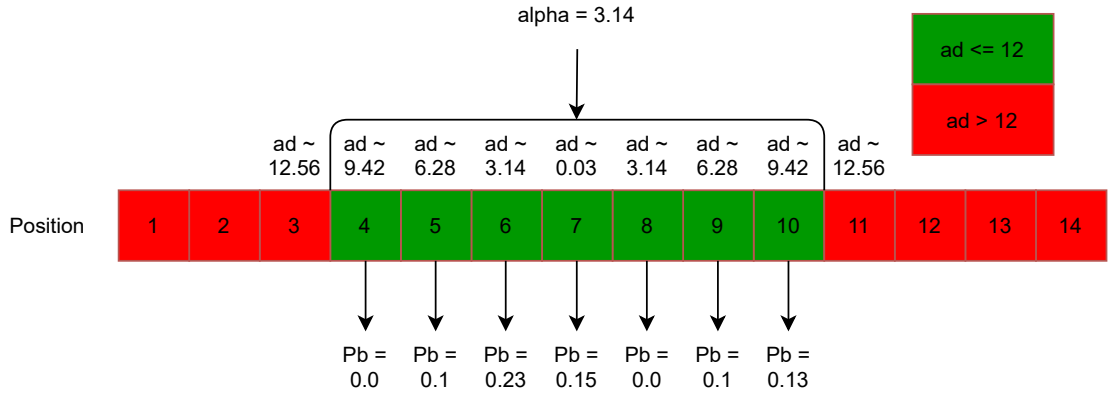


Figure 3: How the window of SNPs included in the CLR calculation is determined

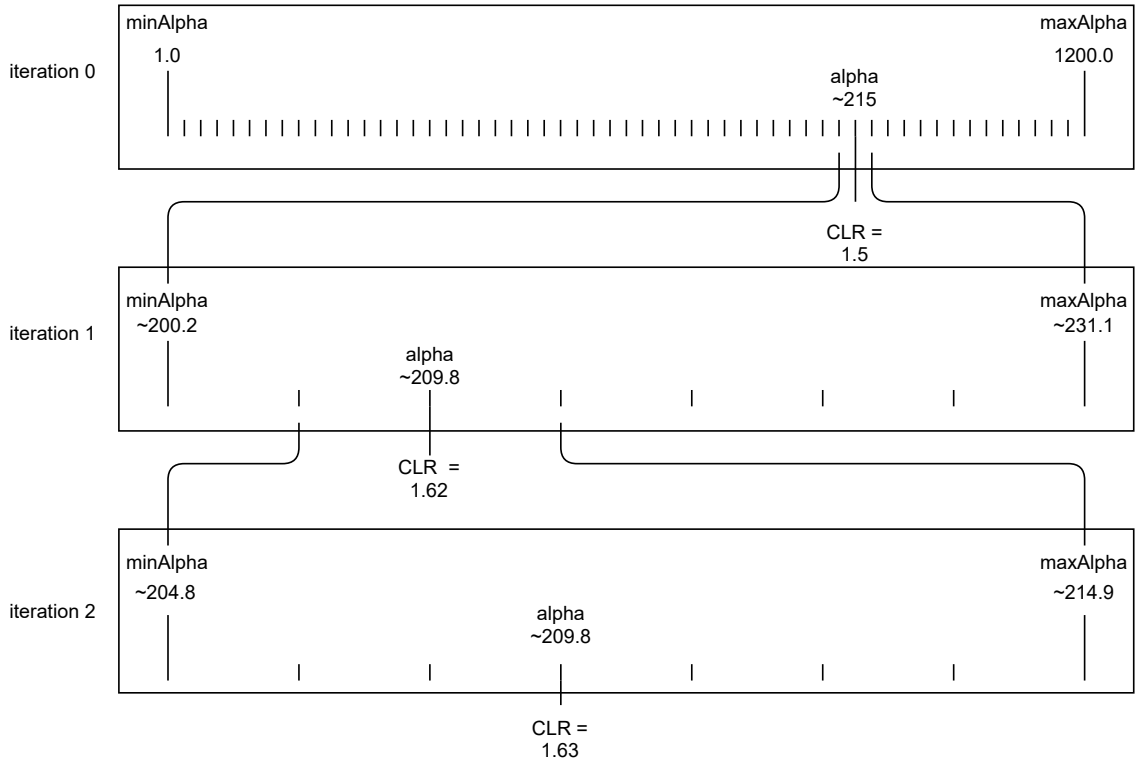


Figure 4: How SweeD iteratively calculates CLRs for α values between a minAlpha and maxAlpha and adjusts minAlpha and maxAlpha based on the highest CLR.

2.3 Clash

Developing designs for FPGAs is normally done using Hardware Description Languages (HDLs) such as VHDL and Verilog. These conventional HDLs can be very cumbersome to work with when implementing complex or abstract functionality, but in the last decade a lot of research has been done regarding the use of high level languages for design of hardware circuits. High level languages offer a higher level of abstraction that eases the implementation of complex problems. Tools like Vitis HLS [17] and Catapult [18] are capable of processing C and C++ code to create RTL descriptions which can speed up the development of hardware designs. This chapter aims to provide a basic understanding of what Clash is, together with some examples how one could approach system design in Clash.

The Clash language is a HDL of which the syntax is a subset of the functional programming language Haskell. Where imperative programming languages are based on sequential execution of instructions that alter the state of the program, functional programming languages are based on lambda calculus where we mainly write code using pure functions. Clash allows us to do structural hardware design in a functional way, the code written in Clash is mostly implementation independent which allows the code to target a wide range of different devices. The subset of Haskell functionality implemented in Clash includes polymorphic typing, function application, user-defined higher order functions and pattern matching allowing for very concise code for complex functionality.

The Clash compiler translates the Clash language into synthesizable VHDL or Verilog code, allowing it to be processed by synthesis tools such as Intel's Quartus or Xilinx' Vivado.

2.3.1 Designing in Clash

When we describe hardware in Clash, we mostly design functions on a register to register basis. For example, imagine we need to implement hardware that solves polynomials such as $y = 3x^2 - 2x + 5$ and $y = 0.5x^4 - 6x^2 + x$. We could write the polynomials that we need to execute as

$$y = f_4 * x^4 + f_3 * x^3 + f_2 * x^2 + f_1 * x + f_0 \tag{5}$$

One possible implementation in Clash is shown below:

```
1 poly x factors = sum $ multResults ++ f0
2   where
3     xs = scanl (*) x (repeat x)
4     multResults = zipWith (*) (tail factors) xs)
5     f0 = take d1 factors
```

Line 3 calculates the results of the exponents of x used in the polynomial by sequential multiplication. Line 4 multiplies xs ($xs = (x, x^2, x^3, ..., x^n)$) with the corresponding factors ($factors = (f_0, f_1, f_2, f_3, ..., f_n)$). Note that f_0 is not included in this multiplication since it is a constant in the equation. Lastly the output at line 1 shows that we return the sum of the multiplication results and f_0 to obtain the result. One of the advantages of using Clash is the fact that this function is polymorphic, the type of poly is `(KnownNat n, Num a) => a -> Vec (1 + (1 + n)) a -> a`, meaning it works on any data type in the Num class for all factor vectors of the same type with a known length greater than 2. This enables poly to be used on for example signed and unsigned integers of various lengths, but also signed and unsigned fixed point values. Figure 5 shows a visual representation of the structure of poly for a factor vector with a length of 4 (e.g. for $y = 3x^3 - 2x^2 + x - 1$).

While we can make use of polymorphic functions for the design of the hardware, the Clash compiler requires our final hardware description to be monomorphic. This means that the types of all input arguments, intermediate functions, intermediate variables and the result must be either explicitly defined or can be derived such that the system can only have a single form. The current design is purely combinational, but we can not simply create an entire design based on combinational logic because this would severely limit the operating frequency, we need to add some registers. Clash has an applicative type named `Signal dom a` that enables us to create synchronous systems. Here `dom` is a domain that contains a clock, reset and enable signal. Usually when we design hardware for synchronous systems, we do not alter the clock, reset and enable signals. For that reason, these signals are usually applied implicitly such that we do not need to apply them to every function explicitly over and over again. However, the clock, reset and enable signals can be applied explicitly by exposing them, this technique will be utilized later in our design. The advantage of exposing these signals is that we can control them, for example for clock gating or stalling logic by controlling the enable signal.

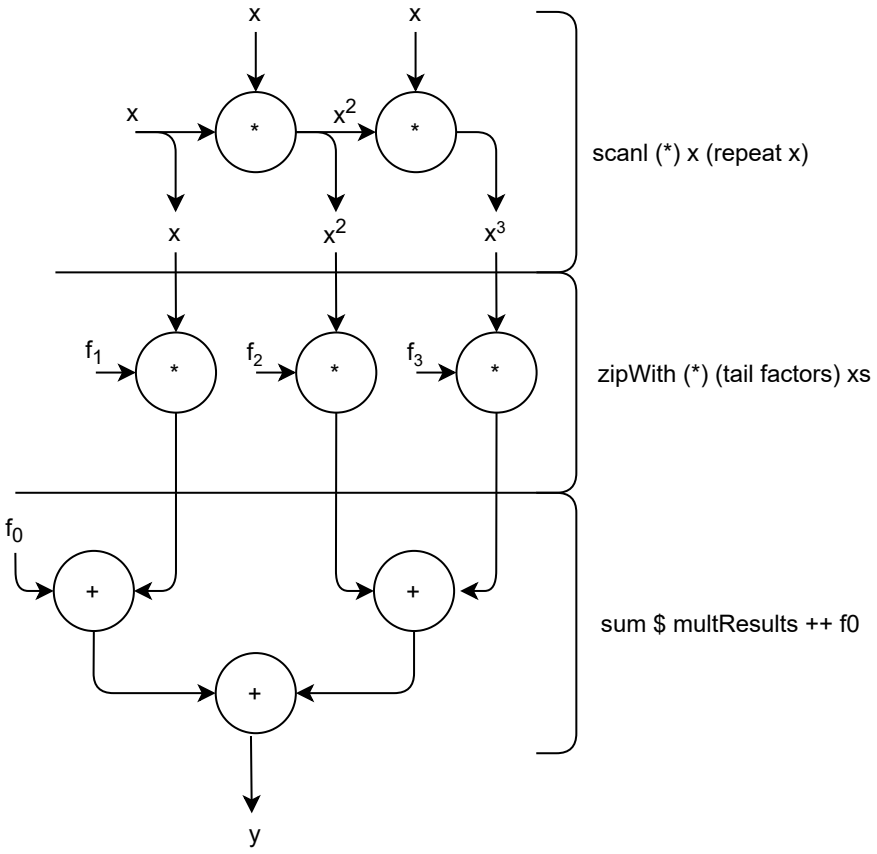


Figure 5: Visualisation of the structural description for poly with a factor vector of length 4.

Clash offers multiple functions that return or can be applied to `Signal dom A`. A couple of examples are:

- `register :: a -> Signal dom a -> Signal dom a` – Receives an initial value and an input signal and returns an output signal. This function is simply a register that stores the input signal and returns it the next cycle.
- `mealy :: (s -> i -> (s, o)) -> s -> Signal dom i -> Signal dom o` – This function can be used to create mealy machines, systems with an input that alters a state and an output that depends on both the state and the input. The first argument is a function that takes a state and an input and returns a tuple with a state and an output `(s -> i -> (s, o))`. The second argument is the initial state of the system and the third argument is the input signal. The mealy function turns the input function into a mealy machine, connecting the output state to the input state with a register in between such that the output state becomes the input state in the next clock cycle.

To make the types more compact we are going to be using the following standard type synonyms.

```
type Clk = Clock System
type Rst = Reset System
type En = Enable System
type Sig = Signal System
```

A minimal example of a synchronous system would be adding a register to the input of our poly function. Remember that the type of poly is `(KnownNat n, Num a) => a -> Vec (1 + (1 + n)) a -> a`, but the register function returns data of type `Signal dom a`. To apply the poly function to the output of the register we have to map it over the `Signal dom a` argument returned by register. To do this, we use the following functions

- `<$> :: Functor f => (a -> b) -> f a -> f b`
  This function takes a function `( a -> b)` as first argument, the second argument can be any a in context f (in our case this f is often `Signal dom a`) and returns f b (the result of `(a -> b)` applied to f a).

- `<*> ::Applicative f => f (b -> c) -> f b -> f c`
  Some functions take multiple arguments. However, since Clash borrows its syntax from Haskell, functions are curried. Meaning that a function which takes two arguments, first takes the first argument and returns a function which takes the second argument. Thus when we use `<$>` to map a function of type `(a -> b -> c)` over `f a`, it returns `f (b -> c)`. This function enables us to apply this returned function to the second input argument `f b` to obtain `f c`.

Mapping the data over an input with register looks as shown in line 7 below:

```
1  poly x factors = sum $ multResults ++ f0
2    where
3      xs = scanl (*) x (repeat x)
4      multResults = zipWith (*) (tail factors) xs)
5      f0 = take d1 factors
6
7  poly' x factors= poly <$> (register 0 x) <*> factors
```

This code enables us to utilize poly' to calculate the result of a different polynomial every clock cycle. Now consider we want to calculate the result of the constant polynomial $y = 3x^2 - 2x + 5$. We can do so using a lambda function as follows:

```
1  poly x factors = sum $ multResults ++ f0
2    where
3      xs = scanl (*) x (repeat x)
4      multResults = zipWith (*) (tail factors) xs)
5      f0 = take d1 factors
6
7  polyConstant :: HiddenClockResetEnable System
8                  => Sig (SFixed 16 16)
9                  -> Sig (SFixed 16 16)
10 polyConstant x = (\ x -> poly x (5 :> (-2) :> 3 :> Nil)) <$> register 0 x
```

Simulating hardware in Clash is relatively easy and straightforward. There are two main functions that we can utilize:

- `simulate :: (KnownDomain dom, NFDataX a, NFDataX b) => (HiddenClockResetEnable dom => Signal dom a -> Signal dom b) -> [a] -> [b]`
  This function can be used to simulate the hardware for a list of inputs [a] and returns a list of the generated outputs [b].

- `sampleN :: (KnownDomain dom, NFDataX a) => Int -> (HiddenClockResetEnable dom => Signal dom a) -> [a]`
  This function can be used to simulate hardware for a certain amount of clock cycles for a constant input.

Using the simulate function we can simulate our design for a list of inputs as follows:

```
*PolyDesign> simulate @System polyConstant [-2,-1,-0,1,2]
[5,21,10,5,6,13]
```

The first result is the initial value of the circuit. The result for the last input does not reach the output during the simulation due to the latency of the circuit.

2.4 Pipelining nested loops

One of the obstacles encountered in this thesis is the fact that SweeD contains many nested loops with different properties. The first important property is data dependent repetition, meaning we can not determine the amount of iterations before the loop starts. The second important property is containing loop-carried dependencies (LCDs), meaning that the functionality inside loops require results from the previous iteration. The main benefit of FPGA architectures is the fact that computations can be performed in parallel. However, these loop iterations have to be executed sequentially due to their data dependencies. Consider a 100 stage pipeline that is part of a loop. The computations inside the loop that is executed have a latency of around 100 clock cycles. If we have to wait ∼100 cycles before starting the next iteration it would severely affect performance. Styles et al. [2] present a method of efficiently pipelining loops with LCDs. This paper introduces 5 mechanisms that can be used to make optimal use of all stages in the pipeline. Consider the following algorithm consisting of two loops:

Algorithm 1: somethingUsefull(b)

```
1: initialize result[0..100] to 0.0
2: for i in [0..100] do
3:     a = array[i]
4:     for j in [0..b] do
5:         result[i] = result[i] + j * a^2
6:     end for
7: end for
8: return sum of result
```

The iterations of the outer loop contain no LCDs since every iteration with iterator i uses data from only array[i]. The iterations of the inner loop do contain LCDs since every iteration both reads from- and writes to result[i]. We can not perform the iteration of j = 1 before the iteration of j=0 is completed, because iteration j=0 writes to array[i], which is required in j=1. If we consider the operations in line 5 we can construct the hardware description shown in Figure 6. The latencies of the

operators are presented as delays in clock cycles. Assume the different inputs are properly delayed such that they are synchronised at the inputs of the operations. The example has an accumulated latency of 17 cycles. If we were to wait for the result of the previous cycle to start the next, the performance would be reduced significantly.

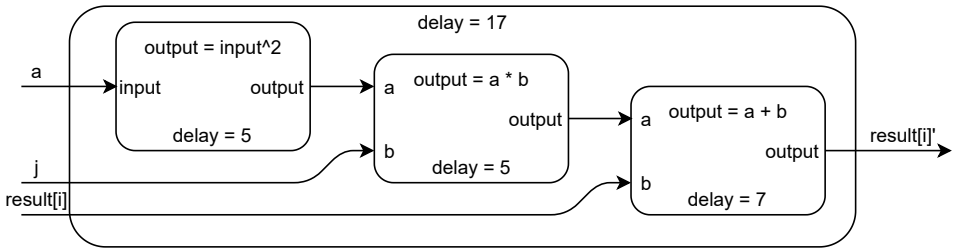


Figure 6: Example of LCD operation pipeline

The mechanisms that are introduced in the paper enable us to create a hardware description that fully utilizes the pipeline by performing multiple iterations of the outer loop at the same time. These mechanisms are described below:

- **Tagged token** - The tagged token identifies the outer loop iteration and is used to later reorder the inputs. It is assumed that this can be accomplished by using a large enough memory bank that can store the output for every concurrent call. After completion of all iterations, the data can be obtained from this memory bank in order. It is also assumed that the circuit is placed in between memory banks such that the input also comes from a memory bank.

- **Merge circuit** - The merge circuit has two data inputs and a single data output. If one of the inputs contains valid data it is routed to the output. If both inputs contain valid data the inputs are blocked in an alternating round-robin fashion.

- **Blocking** - The blocking of a data stream is achieved by a ready signal that allows the circuitry of a valid data stream to be stalled while it is not selected by the merge circuit. By stalling the circuitry, the data is not while not selected by the merge circuit.

- **Branch circuit** - The branch circuit has a single data input and two data outputs. The circuit makes the valid data at the input available at one of the outputs depending on a secondary control input.

- **Rate smoothing queues** - The rate smoothing queues are simple FIFO Buffers that are used to smooth out variations (bursts) of data at the input of the merge circuit. The FIFOs also use a ready signal to indicate when they are full.

When we implement these mechanisms in our example algorithm we can create the structural description shown in figure 7. Here RSQ refers to Rate Smoothing Queue, M refers to the merge circuit, j<b refers to the break condition of the loop and B refers to the Branch circuit. The merge circuit emits a low ready signal to the RSQ whenever the input is not selected. The RSQ emits a low ready signal whenever the buffer is full to indicate that it cannot accept more inputs. Note how due to the fact that the amount iterations depends on b, the input to output delay is variable. With this design, the hardware can process 17 iterations of the outer loop concurrently.

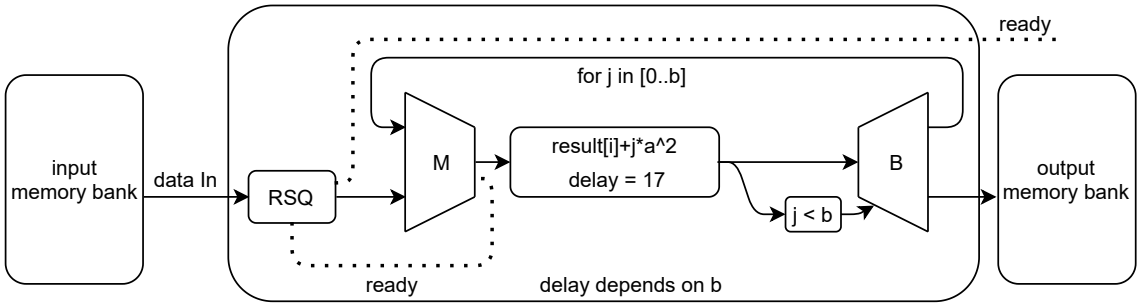


Figure 7: Example of implementing efficient pipelining method.

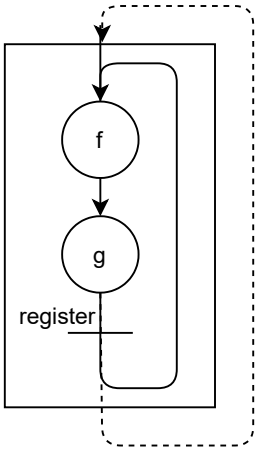
Petkov et al. [19] presents a method of pipelining nested loops by rearranging functions in such a way that increases parallelism and reduces clock cycles. This method is aimed to increase the throughput of signal processing systems that are being designed with high level programming languages in combination with high level synthesis to map abstract designs into silicon. Consider the code in altgorithm 2 with the corresponding DFG in figure 8a. By applying the unroll and squash method, multiple outer loop iterations can be processed concurrently as shown in algorithm 3. The corresponding DFG is shown in figure 8b.

Algorithm 2: Normal scheduling

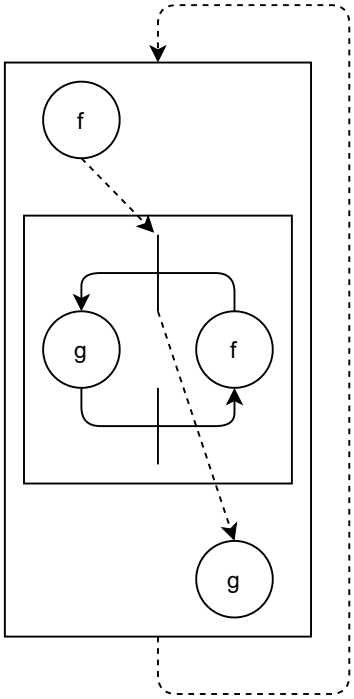
```
1: for i in [0..M] do
2:     a = data_in[i]
3:     for j in [0..N] do
4:         b = f (a);
5:         a = g (b);
6:     end for
7:     data_out[i] = a;
8: end for
```

Algorithm 3: Unroll-and-squash factor 2 scheduling

```
1: for i in [0..M], increments of 2. do
2:     a_1 = data_in[i] ; a_2 = data_in[i+1]
3:     for j in [0.. (2N-1)] do
4:         b_2 = f (a_2) ; a_1 = g (b_1);
5:         a_2 = a_1 ; b_1 = b_2
6:     end for
7:     a_1 = g (b_1);
8:     data_out[i] = a_2 ; data_out[i+1] = a_1;
9: end for
```



(a) DFG for Normal scheduling



(b) DFG for Unroll-and-squash factor 2 scheduling

Figure 8: DFGs for normale scheduling and Unroll-and-squash

3 Related work

3.1 Different methods of Sweep Detection

In this section we will discuss the various methods that are available for selective sweep detection. The method deployed by SweeD is based on the method used by SweepFinder which has already been discussed in section 2.2. Years after the release of the original SweepFinder, DeGiorgio et al. [20] released SweepFinder 2, which is an extension of SweepFinder with increased sensitivity and robustness. One of the major additions here is that SweepFinder 2 accounts of the effects of negative selection on diversity when searching for adaptive alleles [20]. SweepFinder, SweepFinder 2 and SweeD base their method on how a selective sweep affects the Site Frequency Spectrum, but this is not the only indicator of a selective sweep. There are three distinct signatures that are the consequence of a selective sweep. The first is a local reduction of polymorphism levels due to hitchhiking [21], the second is a shift in the site frequency spectrum due to hitchhiking [22] and the third is a localized pattern of Linkage Disequilibrium (LD) values [23]. The Linkage Disequilibrium is the nonrandom association of alleles at different positions on the genome [24].

One of the first tools to deploy the ω-statistic from [23] based on LD values is OmegaPlus [10]. The authors of OmegaPlus compared it to SweepFinder and managed to process a data set of 500 sequences and 70 000 SNPs in just 71.8s, where it took SweepFinder over 783.2s. OmegaPlus was the first tool to be able to use the ω statistic to detect selective sweeps so quickly. OmegaPlus was also the first tool that makes use of parallel threads to speed up the ω-statistic calculations. Pavlidis et al. [11] and Crisci et al. [25] both compared SweepFinder, SweepFinder 2, SweeD and OmegaPlus to eachother. Crisci et al. [25] mainly focuses on the type-I and type-II error rates of methods and found that OmegaPlus is best able to reject the neutral model under assumptions regarding the equilibrium of the population and suggests that the LD-based method may be more fruitful for detecting selective sweeps. Pavlidis et al. [11] also found that OmegaPlus remained the best tool to detect selective sweeps due to its comparable false positive rate, but higher true positive rate. However, when the assumption of a population at equillibrium is violated and the data sets are derived from bottlenecked populations (populations which size is reduced by 99%), the false positive rate of OmegaPlus significantly increases. For this situation, Crisci et al. [25] reported a false positive rate between 0.05 and 0.91 and Pavlidis et al. [11] reported a false positive rate between 0.07 and 0.69 for OmegaPlus. The SFS-based methods fared significantly better in this situations with false positive rates between 0 and 0.08 and 0 and 0.13 for SweepFinder and SweeD respectively according to Crisci et al. [25]. Pavlidis et al. [11] showed false positive rates between 0 and 0.18 and 0 and 0.12 for SweepFinder and SweeD respectively. Besides the ability of the tools to detect selective sweeps, Pavlidis et al. also considered the execution times of SweepFinder, SweeD and OmegaPlus on a single core processor and found that SweeD is significantly faster than SweepFinder while requiring the same computations. The execution time of OmegaPlus was orders of magnitude lower for larger data sets which was mainly attributed to the fact that it requires limited floating point operations and the majority of operations are performed on integers.

In 2018, Alachiotis and Pavlidis [12] introduced RAiSD (Raised Accuracy in Sweep Detection), which offered a new method of sweep detection. The authors introduced the μ-statistic which is a composite evaluation test that uses changes in the SFS, changes in LD levels and amount of genetic diversity to detect selective sweeps. RAiSD utilizes a sliding window approach that reuses calculated data between overlapping windows. Their new method of calculating selective sweeps is less compute intensive than the methods deployed by other tools despite relying on three different characteristics. The authors compared RAiSD to SweepFinder2, SweeD and OmegaPlus and found that in all cases RAiSD had a higher true positive rate than SweepFinder and SweeD. In most cases RAiSD also outperforms OmegaPlus in this aspect, but the authors found that in the case with the highest recombination intensity which was considered, OmegaPlus showed a slightly higher success rate than RAiSD (59.9% vs 59.1%). In the presence of background selection, SweeD and SweepFinder2 showed the lowest false positive rate of 0.3% where OmegaPlus and RAiSD showed a significantly higher false positive rate of 8.4% and 37.1% respectively. However, in this case RAiSD showed a significantly higher true positive rate of 97.5% compared to SweepFinder2, SweeD and OmegaPlus (59.9%, 56.5% and 21.1% respectively). The authors also compared the execution times of the tools and found that in their runs, RAiSD has 1 000x, 628x and 32x lower executions times compared to SweepFinder2, SweeD and OmegaPlus, respectively, utilizing a single thread. Note that unlike RAiSD, SweeD and OmegaPlus are capable of multithreading and that SweepFinder2, SweeD and OmegaPlus can process a data set arbitrarily faster or slower depending on the grid size parameter. The data sets used by the authors to obtain these results are relatively small. They contain 1 000 sets with an average of 2 215 SNPs and only 20 sequences per set.

3.2 Hardware acceleration in Selective sweep detection

Algorithms for sweep detection contain relatively complex computations, but the first efforts to use hardware acceleration for selective sweep detection have recently been made. Bozikas et al. [26] presented an FPGA-based architecture to accelerate the calculations of LD values. Bozikas et al. [26] acknowledges that calculating these values is intrinsically a memory bound operation and maximizing the amount of operations per memory access is critical for the performance of the accelerator. Two memory layout transformations are utilized to increase the information density of individual memory accesses. Initially each word contained the state of a single base while the minimum amount of bits required to represent the state is 4 bits. The first transformation packs 4 bases corresponding to a single SNP in a single memory address. The second transformation enables the utilization of multiple memory controllers that utilize multiple memory ports to increase parallel data fetching. With the accelerator mapped to 4 FPGAs Bozikas et al. [26] managed to support the processing of large data sets and measured speed ups of LD calculations from 12.7x (4 FPGAs vs 12 cores) up to 134.9x (4 FPGAs vs 1 core)[1].

[1]Virtex 6 LX980 FPGAs versus Intel Xeon E5-2630 6-core @2.6 GHz with 32 GB RAM

Alachiotis et al. [27] managed to create a system level hardware solution for positive selection inference in whole genome data that utilizes an out of core algorithm that mainly handles fetching data in parametrized chunks combined with a Decoupled Access/Execute Reconfigurable (DAER) [28] architecture. The accelerator is based on the μ-statistic [12] and only needs to analyze the data set once, enabling the authors to utilize a memory efficient sliding window approach. Furthermore the hardware description for this implementation is generated with Xilinx Vivado HLS. The authors compared their implementation to SweeD and OmegaPlus and measured a speedup of 36.8x and 29.7x, respectively, for the sequential implementations of SweeD and OmegaPlus and a speedup of 5.3x and 4.9x, respectively, for parallel implementations[2]. Later, Alachiotis et al. [13] presented RAiSD-X, an optimized FPGA-based accelerator system for fast and accurate detection of positive selection. This work utilizes the design methodology presented in their previous work [27] to implement a hardware accelerated version of RAiSD [12] with improved scalability and accelerated performance. The authors compared the results of RAiSD-X to SweeD and measured that the runtime of RAiSD-X was 1 051x lower than SweeD for their data sets and parameters[3]. Besides a reduced runtime on their accelerated platform, RAiSD-X also measured a higher detection accuracy of up to 75.9% for data sets where SweeD reached a detection accuracy of 46.1%.

We present the first hardware accelerated sweep detector that makes use of SFS-based CLR computations in order to detect selective sweeps, compared to LD-based method SFS-based methods show a lower false positive rate which makes them a suitable extension of the bio-informatics toolbox. In our method we explicitly focus on the flexibility of the system to apply the design to different types of FPGAs, which should allow for easier adaptations. Because of the use of Clash as hardware design language it is also easy to alter the implementation in case new studies offer valuable additions to SweeD.

[2]SweeD was executed on a Dell PowerEdge R530 rack server with two 10-core Intel Xeon E5-2630v4 CPUs running 20 threads per CPU at 2.2GHz. Accelerator was executed on the following hardware platform: Intel Core i7-5930K 6-core @3.5GHz host processor with 32GB DDR4 Memory, an AC-510 SuperProcessor Module with 4GB Hybric Memory Cube and a Xilinx Kintex UltraScale XCVU060 FPGA.

[3]See footnote 2.

4 Design space exploration

4.1 Determining what to speed up

SweeD contains two very computationally expensive parts: Data preparation and likelihood calculation. The run-time of data preparation mainly depends on the amount of samples in the data set and the run-time of likelihood calculations mainly depends on the length of the samples in the data set. The data preparation phase pre-calculates probabilities for all possible combinations of SNP frequency, samples and 300 alpha distance values. These probabilities are pre-calculated because their result is generally accessed significantly more often than the amount of possible combinations. The second computationally heavy part, the likelihood calculations, aims to process the whole genome. The genome is processed on a per grid point basis. Reducing the grid size reduces run-time at the cost of accuracy. To make a hardware accelerator it is important to know which part of the program should be accelerated to create the highest speed up. The highest speed up can be achieved by accelerating the part that requires most computational power. We will determine which part to accelerate with 2 approaches, the first approach is profiling to measure which part requires most processing power. We can use this to isolate computationally expensive functionality and gain insights how the runtimes change for different data sets. The second approach is on a lower level, manually analysing the C implementation of SweeD and reasoning about the code to determine which functionality is fit to be accelerated using dedicated hardware. Here we specifically pay attention to expensive functions, different kinds of loops and data dependencies. The data dependencies will be discussed seperately in section 4.3.2.

4.2 Profiling SweeD

Profiling of SweeD is done using Valgrind [29]. Valgrind is a dynamic binary instrumentation framework (DBI), which makes it easy to make checkers and profilers. The profiler that we deployed is Callgrind [30]. Callgrind records function calls, caller callee relations, and can estimate the amount of clock cycles spend on each function. Callgrind can perform branch prediction and simulate cache to determine cache hits and misses for better cycle estimation. We use Callgrind to determine which parts of the program requires most computational power. A couple of parameters influence the execution time of SweeD. The parameters that mostly affect runtimes are:

- Sequence length
- Amount of sequences
- Amount of gaps in sequences
- Grid size

The sequence length, amount of sequences and amount of gaps are characteristics of the data set at hand. The grid size can be set by the user, where a larger grid size will lead to a more accurate result at the cost of computation time. The user defined grid size is always a discrete amount of positions. The grid size is smaller than the sequence length and these points are evenly spaced along the entire length of the genome. In profiling we determine the grid size based on a percentage of the sequence length (e.g a sequence length of 1 000 with a grid percentage of 50% would

result in a grid size of 500). This ensures that the interval between grid points is the same for every profiling run. Using Hudson's ms [31], we can generate different data sets with different combinations of sequence length and amount of sequences to be used as input for SweeD. It is already known that the sequence length and amount sequences are most significant in determining the run-time of SweeD. SweeD has been be profiled with every combination of the following parameters.

- **Sequence length** - 100, 1 000, 10 000.
- **Amount of sequences** - 10, 100, 1 000.
- **Grid percentage** - 20%, 50%, 80%.

The output of callgrind will be processed with kCachegrind to estimate the amount of cycles spent on each function. Figure 9 shows the distributions of clock cycles spend by SweeD for runs with a grid of 80%. In this figure we show that most clock cycles are spent on the 3 functions createProbs, createSFS and getAlpha. The createProbs and createSFS functions are part of the data preparation phase and the getAlpha function contains the likelihood calculations. The createSFS function creates the Site Frequency Spectrum and createProbs creates the three dimensional probability grid that contains probabilities for every available combination of SNP frequency and amount of samples for 300 alphaDistance values. Appendix B shows the complete profiling results.

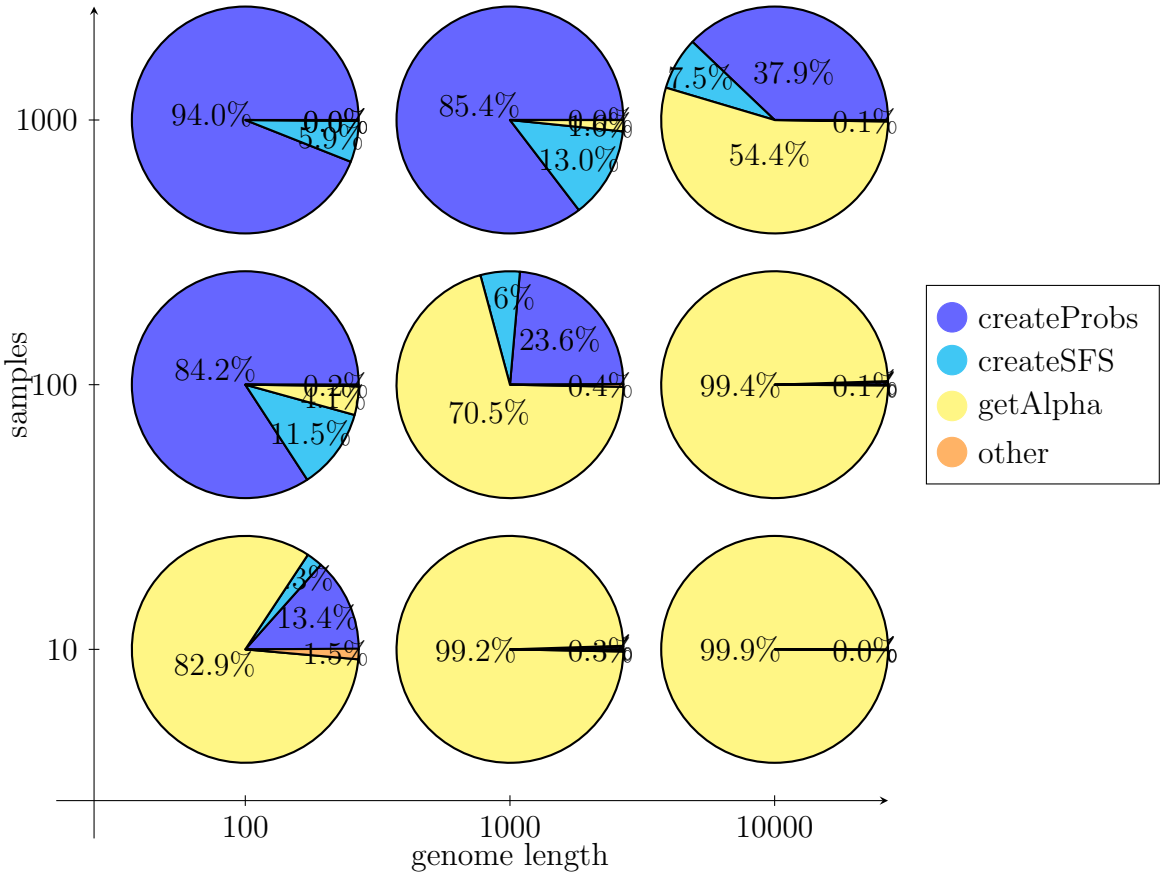


Figure 9: Clock cycle distributions with grid 80%

From figure 9 it can be deducted that if we increase the amount of samples in the data set, the proportion of runtime dedicated to the data preparation functions (createProbs and createSFS) is increased. Data sets with longer sequences result in a higher runtime for the likelihood calculations (getAlpha). Ideally both cases would be accelerated. Considering the fact that the speedup achieved by SweeD compared to SweepFinder significantly decreased for data sets with longer sequences, the choice has been made to first focus our efforts on accelerating the getAlpha function.

4.3 Manual analysis of likelihood calculations

In section 4.2 we determined that our efforts will be focused on the likelihood calculations and that accelerating the getAlpha function would significantly speed up SweeD for data sets with longer genomes. During the manual analysis we analysed the loops, data dependencies and computationally expensive functions. We separate our findings in two parts, first we present the loops and expensive functionality, secondly we present the data dependencies in context of the loops. For the each loop we identify two different characteristics, whether the loop has loop carried dependencies and whether the amount of iterations is data dependent. Loop carried dependencies are data dependencies defined out of the scope of the loop that contain write after read behaviour, in essence, an iteration of the loop alters the variable that will be used in the next iteration (Example shown in section 2.4).

4.3.1 Loops and expensive functionality

Table 3 shows a summary of the different loops that are relevant for accelerating the getAlpha function. For each loop, we keep track of the nesting depth which indicates how deeply nested the loop is in higher level loops, we also keep track of the type of the loop. While loops are generally used for data dependent repetition and for loops are generally used for loops with data independent repetition. For loops can contain conditional break statements, which makes their repetition data dependent. The first loop can be found in main and simply iterates over all points in the grid, for which it calls the getAlpha function. The rest of the loops will be explained in their respective function in the rest of this section.

Function	Nesting depth	Loop type	Contains LCDs	Data dependent repetition
main	0	for	No	No
getMinMaxAlpha	1	for	No	Yes
getMinMaxAlpha	1	for	No	No
getClosestSNPIndex	1	while	Yes	Yes
getAlpha	1	while	Yes	Yes
getAlpha	2	for	Yes	No
getLikelihood	3	for	Yes	Yes
getLikelihood	3	for	Yes	Yes

Table 3: Summary of loop characteristics

getAlpha

A pseudo code variant of the getAlpha function can be found in algorithm 4. The highlighted parts represent functions that contain loops. The getAlpha function finds the alpha parameter for which the calculated likelihood of this position is the highest and returns this likelihood with the corresponding alpha. The first 2 loops encountered are in separate functions: getMinMaxAlpha and getClosestSNPIndex, as shown in algorithm 5 and 6 respectively. The first loop of getMinMaxAlpha (line 2 to 7) does not contain LCDs, but the amount of iterations depends on the content of the positionsInd array. The second loop in getMinMaxAlpha (lines 8 - 17) iterates over every value in the positionsInd array, this array can be very large as data sets can contain millions of segsites (SNPs). The loop in getClosestSNPIndex (lines 2-9) contains the LCDs rightInd and leftInd and the amount of iterations is data dependent. The main loop of the getAlpha function (lines 4-16) requires both minAlpha and maxAlpha, which are recalculated at the end of every iteration. The variables minAlpha and maxAlpha are LCDs as well as part of the break condition, which implies that the amount of iterations can not be determined beforehand. The secondary loop of the getAlpha function (lines 6-13) contains multiple LCDs for tracking the highest alpha and likelihood combination and accesses to the sweepWidth array. The amount of iterations executed by the for loop is known beforehand since it depends on localGridSize, which is not altered in the loop.

Algorithm 4: Pseudo code variant of getAlpha

```
1:  localGridSize = 100
2:  Calculate minAlpha and maxAlpha (call getMinMaxAlpha)
3:  Calculate startPosition (call getClosestSNPIndex)
4:  while minAlpha and maxAlpha have not converged do
5:    Calculate interval.
6:    for i in [0..localGridSize] do
7:      Calculate alpha based on minAlpha, i and interval.
8:      if i == 0 or sweepWidth[i-1] > 0 then
9:        Calculate likelihood for this position, startposition and alpha (call
          getLikelihood).
10:     end if
11:     Update sweepWidth[i].
12:     Keep track of highest likelihood and alpha for this position.
13:   end for
14:   update minAlpha and maxAlpha based on highest likelihood.
15:   localGridSize = 5.
16: end while
17: return the highest likelihood and corresponding alpha.
```

Algorithm 5: Pseudo code variant of getMinMaxAlpha

```
1: maxAlpha = 1300.0, counter = 0, totDist = 0
2: for i in [0..segsites] do
3:    if positionsInd[i] != -1 then
4:       minDist = abs ( positionsInd[i] - sweepPosition)
5:       Break for-loop
6:    end if
7: end for
8: for i in [0..segsites] do
9:    if positionsInd[i] != -1 then
10:      counter++
11:      distDif = abs ( positionsInd[i] - sweepPosition)
12:      totDist += distDif
13:      if distDif < minDist then
14:         minDist = distDif
15:      end if
16:   end if
17: end for
18: if minDist !=0 then
19:    maxAlpha = 13 / minDist
20: end if
21: minAlpha = 12 * (counter / totDist)
22: return minAlpha and maxAlpha
```

Algorithm 6: Pseudo code variant of getClosestSNPIndex

```
1: leftInd = 0, rightInd = Sequence length
2: while rightInd - leftInd > 1 do
3:    index = (rightInd + leftInd) / 2
4:    if positionIndex[index] > sweepPosition then
5:       rightInd = index
6:    else
7:       leftInd = index
8:    end if
9: end while
10: return leftInd
```

The getLikelihood Function The purpose of the getLikelihood function is to calculate a composite likelihood ratio (CLR) for a position on the genome with a given α. Algorithm 7 shows pseudo code of the getLikelihood function.

Algorithm 7: Pseudo code variant of getLikelihood

```
1:  for i in [startPosition..0] do
2:    Calculate alphaDistance
3:    if alphaDistance < 12.0 then
4:      Interpolate probability (call splint ())
5:      Process folded spectrum
6:      update likelihood
7:    else
8:      break for-loop
9:    end if
10: end for
11: for i in [startPosition+1..SNPs] do
12:   Calculate alphaDistance
13:   if alphaDistance < 12.0 then
14:     Interpolate probability (call splint ())
15:     Process folded spectrum
16:     update likelihood
17:   else
18:     break for-loop
19:   end if
20: end for
21: return likelihood
```

The amount of iterations for both sequential for loops (lines 1-10 and 11-20) is data dependent since they both contain a data dependent break condition (line 8 and 18). These loops both contain an LCD since they require the likelihood calculated in the previous iteration to calculate the new likelihood. The inner iterations of these loops are computationally expensive and repeated excessively. These are the calculations that should be accelerated. For this project we only consider data sets that are not folded, so the processing of the folded spectrum will for now not be included.

4.3.2 Data dependencies of the getAlpha function

Unfortunately ideal memory that can provide all data instantly does not exist. SweeD uses multiple arrays of which the size is data dependent and too large for storage on an FPGA. Because we have to make sure that the data will be available at the right place at the right time, we need to analyze the data dependencies. In the previous section we determined that a high speedup could be gained by accelerating the getAlpha function. In this section we give an overview of the data dependencies found in getAlpha, focussing on accesses to arrays that are defined out of getAlpha's scope. A visual overview of these data dependencies is shown in figure 10. A complete overview of all data dependencies can be found in appendix C. Table 4 shows all arrays involved in the getAlpha function and their corresponding datatypes, entries and contents. The sizes of these arrays depend on characteristics of the data set, these characteristics are as follows:

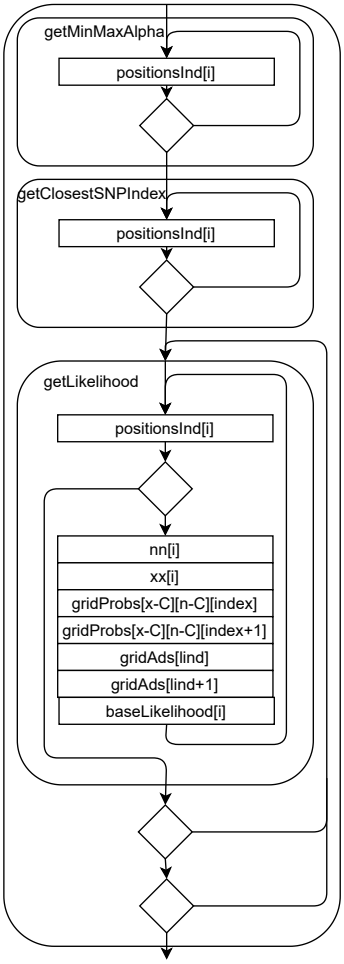


Figure 10: Overview of array accesses in different loops.

- **SNPs /segregating sites / segsites** - Amount of polymorphic sites of the genome. Realistically this characteristic could range from for example 30 thousand for a SARS-CoV-2 data set to millions for a human genome.
- **SNP Frequency Range (SFR)** - Range of the SNP frequencies of the data set, maximum range is 0 to the amount of samples. Realistically this value is close to the amount of samples in the data set.
- **Gaps** - For some data sets, not every sample contains data for every position which introduces a gap. This characteristic is the maximum amount of gaps for a single position in the data set.

Hardware design is structural, so reading data in various locations of the design occurs both over area and time. This requires us to make the data available at the right place at the right time. Considering the size of these arrays and the memory available on consumer FPGAs, not all of these arrays can be stored on chip and some external memory unit is required. When we look at the arrays in table 4, we notice that the gridAds array is small and of constant size, making it easy to store this on chip. The other arrays are all of variable size, it depends on the data set whether or not they can be stored on chip. Using off-chip memory has the advantage of large capacity, but the disadvantage of increased latency.

Array name	Array size	Data type	Access type	Contents
positionsInd	segsites	Int	Read only	Position on genome of this index
nn	segsites	Int	Read only	Amount of samples for this index
xx	segsites	Int	Read only	SNP Frequency of this index
gridProbs	SFR * (Gaps+1) * 300	Float	Read only	probability for this SNP Frequency, amount of samples and alphaDistance
gridAds	300	Float	Read only	alphaDistances
baseLikelihood	segsites	Float	Read only	base likelihoods

Table 4: Arrays accessed by getAlpha function

We could simply use only external memory to fetch the data when needed, but the increased latency of every access would significantly impact performance. To minimize stalling of our circuit caused by fetching data, we can use multiple techniques. The techniques considered are:

- Caching
- Recalculation
- Prefetching

Caching is widely used in computing, this simple principle relies on the assumption that a large percentage of consecutive data accesses have a high locality in the address space. For computers that execute programs containing loops, this is often the case. In our case, many of the access patterns are data dependent and consecutive accesses do not necessarily have high locality. The simplest caching mechanism stores the result of a previous access, such that when in the future it is requested again, it is still in cache and does not need to be fetched from external memory.

Recalculation can be used for all arrays containing data that can be calculated on the FPGA. It becomes interesting for larger arrays where the amount of resources required for recalculation is smaller than the amount of resources required for storage of the array. Especially when access patterns are unpredictable this option is worth considering. Whether or not we can recalculate values depends on the complexity of the calculation. The arrays positionsInd, nn and xx can not be recalculated since they are part of our input data.

Prefetching attempts to predict future data accesses based on known patterns. To be able to do this we must find predictability in often unpredictable access patterns. When applied correctly, prefetching can prevent stalling caused by waiting for memory. This method can be applied to all kinds of data, as long as the access patterns can be predicted.

Since it is not feasible to design and implement an external memory access system, the first iteration of the design will rely on the assumption of ideal memory for data

that can not be stored on chip. Section 4.3.3 describes a conceptual solution for prefetching the required data.

4.3.3 Memory access patterns

The patterns of the memory accesses performed by functions seen in figure 10 are explained below:

- **getMinMaxAlpha** – Every call to this function will access all elements of positionsInd in order. For datasets with many SNPs, this will result in many memory accesses.
- **getClosestSNPIndex** – When this function is called, index starts at $segsites/2$ and each iteration positionsInd[index] is accessed. The next access index is always either in the middle of leftInd and index or in the middle of index and rightInd.
- **getLikelihood** – The indexes of the accesses to positionsInd, xx, nn and baseLikelihood are directly related to the iterator of the loops in the getLikelihood function (Algorithm 7). The iterator of the loop starts at startPos and every next iteration decrements the iterator by 1 (startPos, startPos-1, startPos-2 etc.) untill the break condition has been met. After the break condition for the first loop has been met, the iterator is set to startPos+1. The direction is changed such that every next iteration increments the iterator by 1. This pattern makes the accesses to positionsInd, xx, nn and baseLikelihood highly predictable. The gridProbs array is 3 dimensional and contains 300 values for every possible combination of SNP frequency(first dimension) and amount of gaps(second dimension). The index of the third dimension depends on the calculated alpha distance.

4.4 A suitable datatype

The current C implementation of SweeD makes use of the double datatype which is a 64-bit floating point number capable of representing values from approximately $1.795 * 10^{308}$ all the way down to $4.95 * 10^{-324}$ [32]. However, if we want to offload functionality to an FPGA this might limit performance. An alternative could be using fixed point numbers instead of floating point numbers, fixed point implementations will consume fewer resources and less power[4]. Up to 50% power and area savings are not uncommon when a design is migrated to fixed point [33]. We approach this problem by first checking whether we could use single precision float instead of double precision float. Changing all doubles to floats in SweeD is error prone, so we will be focussing on the getLikelihood function. The data types of the inputs and outputs of the getLikelihood function will remain the same, but under the hood all values will processed as single precision floats. The original version of SweeD with double precision floating points will be referred to as SweeD_double, the altered version of SweeD with the float implementation will be referred to as SweeD_single. Implementation details of SweeD_single can be found in appendix A.

The likelihoods and alphas calculated by SweeD are stored in an output file, by running SweeD_double and SweeD_single on the same input data and comparing the output files we can determine whether floating point values are sufficient.

[4]If both implementations have the same amount of bits.

This test will be executed on multiple data sets. Processing the SweeD output files is done by a python script. This script reads the output files and determines the following values:

- **Likelihood deviation** - The deviation between highest likelihoods found by SweeD_double and SweeD_single.
- **Position deviation** - The deviation between the positions of the highest likelihoods on the genome found by SweeD_double and SweeD_single.
- **Alpha deviation** - The deviation between the alpha parameters for which the highest likelihoods where found by SweeD_double and SweeD_single.
- **Maximum likelihood deviation** - Maximum of the deviations in likelihoods calculated by SweeD_double and SweeD_single for all positions.

The results from comparisons between SweeD_single and SweeD_double are shown in table 5.

Dataset length	10 000	10 000	1 0000	10 000	50 000
Dataset samples	200	500	1 000	1 200	2 500
Likelihood deviation	9.999e-07	1.000e-06	1.000e-06	0.0	1.000e-06
Position deviation	0.0	0.0	0.0	0.0	0.0
Alpha deviation	2.000e-05	2.000e-06	4.999e-04	3.000e-05	5.000e-06
Maximum likelihood deviation	1.097e-04	2.999e-06	3.470e-06	1.100e-05	1.000e-05

Table 5: Comparison between SweeD_single and SweeD_double

From these results it seems that the highest calculated likelihood and corresponding alpha parameter are not significantly affected by changing the getLikelihood function from using doubles to using floats. Besides the suitability of single precision floating point values, we are mostly interested in the suitability of fixed point values. The suitability of fixed point is tested in the same way as the suitability of floating point. C offers no native support for fixed point data types. However, some libraries are available that offer this functionality. These libraries offer 32 bit fixed point values or 64-bit fixed point values. The maximum integer value that must be representable is 1200.0 since this is the maximum possible value of alpha, Thus we require at least 12 bits to represent the whole number leaving 20 bits and 52 bits for the fractional part in the 32 bit and 64-bit implementation respectively.

The fixed point tests have been executed by using fixedptc library which supports both 32 bit and 64-bit fixed points [34]. Variations included in this report are:

- **Fixed32.32** - Range = (-2147483648, 2147483647), resolution = 2.3283064e-10.
- **Fixed24.40** - Range = (-8388608, 8388607), resolution = 9.094947e-13.
- **Fixed12.52** - Range = (-2048, 2047), resolution = 2.220446e-16.

Dataset length	10 000	10 000	10 000	10 000
Dataset samples	200	500	1 000	1 200
Likelihood deviation	2.2855e04	3.7165e04	3.9153e04	3.9286e04
Position deviation	4.0824e03	1.4711e03	6.8548e03	4.2534e03
Alpha deviation	NaN	NaN	NaN	NaN
Maximum likelihood deviation	2.2855e04	3.7165e04	3.9153e04	3.9291e04

Table 6: Comparison between F32.32 and double.

Dataset length	10 000	10 000	10 000	10 000
Dataset samples	200	500	1 000	1 200
Likelihood deviation	1.7204e04	2.9276e04	3.1557e04	2.85724e04
Position deviation	4.0824e03	1.4711e04	6.854e03	4.2534e03
Alpha deviation	NaN	NaN	NaN	NaN
Maximum likelihood deviation	2.2855e04	3.7165e04	3.9153e04	3.9291e04

Table 7: Comparison between fixed24.40 and double.

Dataset length	10 000	10 000	10 000	10 000
Dataset samples	200	500	1 000	1 200
Likelihood deviation	2.0438e03	2.0439e03	2.0436e03	2.0430e03
Position deviation	3.5621e03	1.4011e02	5.7039e02	1.3811e03
Alpha deviation	2.0791e01	1.4501e00	2.7635e02	1.4650e01
Maximum likelihood deviation	2.0479e03	3.7165e04	2.0479e03	2.0479e03

Table 8: Comparison between fixed12.52 and double.

When we take a look at these deviations it can be observed that the likelihoods that are being calculated heavily deviate from the likelihoods calculated from the reference implementation. For example, table 6 shows in the last colom a deviation of ∼39 286, which is a result of the fact that with F32.32, SweeD reported a highest likelihood of 39 291, while SweeD_double reported a highest likelihood of 4.9. This leads to nonsensical results for all likelihoods that are selected by the algorithm as highest likelihood for that position. This could be attributed to the resolution of the fractional part of the fixed point number, for that reason we tested up to 52 bits resolution. Also this high resolution lead to nonsensical results where the highest likelihood was always near the maximum value that could be represented. From these results we conclude that using fixed point values up to 64-bits with the fixedptc library did not produce meaningful results and for this project, is not the way to go. Single precision floating point will be used in the rest of this project. In later stages, the precision of the floating point values could be adjusted where needed.

4.5 Relevant flexibility parameters

In this section we discuss which parameters should be available to tune the hardware design. In the original implementation, there are some command line options that can be used to alter the functioning of SweeD. Most of these parameters are not relevant to the part of SweeD that is being accelerated. The ones that are relevant are:

- **-folded** – Incorporates additional computation for SNPs where the ancestor-derivative relation is not know. During this thesis we decided to not include this part due to time constraints.
- **-threads** – In the parallel version of SweeD, this flag determines that amount of POSIX threads that are used.

While the calculations for folded data sets will not be included in our design, this could later be incorporated as a flag where pattern matching can be used to select which implementation to use. By changing this flag, one of either implementations would be generated by the clash-compiler. Besides these parameters the user has not much control regarding the functionality of SweeD in the C implementation. However, when people design hardware implementations, they often have to make choices for which there is not an obvious best solution. A lot of choices relate to limited time-area trade offs and parallelism. In this section we discuss which kinds of parameters we can introduce to enable a user to quickly adjust the design to their available resources, enabling them to squeeze as much performance out of their setup as possible.

4.5.1 Additional parallelism

One of the main advantages of using FPGA is that the fabric allows for a lot of parallelism. To make use of this advantage we need to find points in the algorithm where we can introduce parallelism. Extra parallelism can only be introduced when we have multiple data streams that we can process. In our case, we can obtain these multiple data streams from the loops that are described in section 4.3.1. In section 4.3.1 we also described how most loops that are relevant for this accelerator contain loop carried dependencies and data dependant repetition. The consequence of loop carried dependencies is that iterations have to be executed in a sequential fashion, not allowing for parallel execution of the iterations of the loop. The consequence of data dependant repetition is that we can not know beforehand how many iterations the loop will execute, making it hard to map it to parallel hardware. Normally with a for-loop of e.g. 100 iterations without loop carried dependencies, we can instantiate 100 parallel instances that execute the entire loop in parallel, or choose to process the loop in smaller chunks of e.g. 10 iterations to save area. With data dependent repetition it is not possible to do either of those without knowingly sacrificing the maximum utilization due to the fact that we might be calculating results that will not be used. Consider a for-loop that iterates over values from 0 to 99, but contains a data dependent break condition. We could instantiate 100 instances of the calculations, but if the code breaks after 10 iterations, we calculated 90 results that will not be used. If we process the loop in chunks of e.g. 10 iterations, we would be lucky when the loop executes multiples of 10 iterations, but if it has to execute 11 iterations, the utilization would drop to 55% (11 useful results, for 20 calculations). Conventional methods of introducing parallelism are not ideal. By

implementing the pipelining method presented in section 2.4, we create an architecture that can process the iterations of the outermost loop in our system. While this architecture processes the iterations of the lower loop, the inputs are stalled creating a bottleneck. We can instantiate multiple instances of the hardware that bottlenecks the system to process more of these outer loop iterations concurrently.

4.5.2 Controlling operation resources

The Clash compiler allocates resources for operators based on their type, calculating the result of Signed 16 addition requires less resources than calculating the result of Signed 32 addition. By creating custom data types based on what the data is supposed to represent, the user can control the boundaries of the data sets that can be processed by the accelerator. The advantage of this is that you can tune the data types to the minimum required for the application, reducing the required area and potentially allowing for more parallelism. Table 9 shows the different types of data that can be represented in the getAlpha function and what the range relies on.

Represented data	Data type	Data dependent range?	Range
SNP Frequency	Unsigned integer	Yes	0 - amount of samples
Samples	Unsigned integer	Yes	0 - amount of samples
Genome index	Signed integer	Yes	0 - segsites
Alpha	Floating point	No	0 - 1200.0
alpha Distance	Floating point	Yes	0 - alpha times distance
Local grid index	Unsigned integer	No	0 - 99
probability	Floating point	Yes	unbound to unbound
likelihood	Floating point	Yes	unbound to unbound

Table 9: Table of represented data with corresponding types and ranges.

We can offer the user to set a maximum for segsites and amount of samples to determine the size of the corresponding data types. However, we must consider the operations that are applied to the variables to determine whether or not the results or intermediate results of the operation require a larger range (e.g. the result of multiplying two integers will in most cases be larger than the individual inputs).

5 Design and implementation

In the previous section we described how profiling was used to find the most computationally expensive functionality of SweeD. We also discussed how we analysed SweeD based on repetition, data dependencies and generally expensive functionality to determine which parts are most suitable for acceleration. The previous section also described how we determined to use single precision floating point values in the design instead of the double precision floating point values in the original implementation. We described the data dependencies with the corresponding access patterns in the getAlpha function and how this can be handled in the final implementation. This section describes how we implemented floating point operators in Clash, how we manage to keep all data synchronised and describes the architecture of the proposed design. Due to time constraints, memory access could not be implemented in the design, but section 5.3 describes a possible method that could potentially provide a solution for the data dependencies. Section 5.4 discusses the introduced parameters that can be used to control the data types, the amount of parallelism present in the design and the size and presence of FIFO buffers to tailor the design to the available resources of the target device.

5.1 Data path design

When designing hardware that processes multiple variables, it is important that variables that belong together stay synchronised throughout the design and are easy to manipulate. In this section we will discuss how the data is packaged to enable us to easily access and manipulate the data. We also discuss how we make sure the data stays synchronised and the implementation of floating point operators in Clash.

5.1.1 Packaging data

Clash offers different methods of coupling data, to handle the large amount of variables used in our algorithm we need a method to encapsulates all data that belongs together and make it easily accessible. One method could be to use tuples to store the data. However, if the amount of variables grows this method quickly becomes confusing to design with. A better approach is using record syntax. An example of using the record syntax could be creating a data type for a person, where you want to store their names, age, height and phone number as can be seen below:

```
data Person = Person { firstName :: String
                     , lastName :: String
                     , age :: Int
                     , height :: Float
                     , phoneNumber :: String} deriving (Show)
```

One of the main benefits of this syntax is that you automatically get functions to extract information from this type. For example, if you want to retrieve the full name of a person, you can simply use the automatically generated ”firstName” and ”lastName” functions.

An example for a fullName function is shown below:[5]

```
author = Person {firstName = "Lucas",
                 lastName = "Bollen",
                 age = 24,
                 height = 1.92,
                 phoneNumber = "MightNotWantToDiscloseThis"}

fullName person = firstName person ++ " " ++ lastname person
```

In the example above, applying the function fullName on author would return "Lucas Bollen".

5.1.2 Data synchronisation

Clash offers a type based method to enforce synchronisation of our data in the form of the `DSignal dom n a` context. This context is the same as the `Signal dom a` context, but with the addition of having accumulated delays encoded in the type. Functions that work with `DSignal dom n a`, do not work with arguments of `Signal dom a` (and vice versa) without converting them from one to another. When a `DSignal dom n a` is converted to `Signal dom a`, the delay annotation is simply stripped, but when a `Signal dom a` is converted to `DSignal dom n a`, the delay is set to zero such that it becomes `DSignal dom 0 a`. Consider an operator that takes two arguments, it is important that both arguments arrive at the inputs of the operator in the same clock cycle. In the case that the first argument is delayed (e.g. has an accumulated delay of 3 cycles) and the second argument is not delayed (e.g. accumulated delay of 0 clock cycles), we need to explicitly delay the second argument such that its delay matches the first argument. Delaying variables can be done in multiple ways, one of which is explicitly delaying a variable for a certain amount of clock cycles. A second way to is to use a function that derives the required delay from context. For example, in the case where we need to delay the second input argument to match the first, Clash can derive the required delay from context, because it knows the required delay based on the first input argument.

5.1.3 Floating point operators

For the design of the accelerator we require floating point precision. However at time of writing, Clash can not generate any floating point operators. For every operation on floating point values we have to either design our own operators or use available IP cores. For this project we chose to use external IP cores. To use these IP cores in Clash we must make sure that the IP cores are easy to use in our design and that the simulation behaviour matches the behaviour of the IP core. To instantiate external IP cores from Clash we use black boxes. A black box describes a piece of HDL code (VHDL or Verilog) that instantiates the IP and correctly connects all the inputs as they are applied in the Clash code. In our case we will be focussing on VHDL and the inputs and outputs of the IP cores for floating point values will be of type std_logic_vector (31 downto 0). For that reason we define our own datatype called Decimal, which is simply a BitVector 32.

[5]Technically, Strings are lists of characters and the (++) function is defined for vectors in Clash, so one would have to import (++) from the Data.List module, however this example works in Haskell.

The IP cores that will be instantiated by our Clash implementation will be generated with Quartus, but any other source of IP cores can be used. Each floating point operator has their own latency. The operators for which functions are defined, together with its latency are shown in in table 10.

Operatore	Latency
Multiplication	5 cycles
Addition	7 cycles
Subtraction	7 cycles
Comparisions (>, >=, ==)	1 cycle
Integer to float conversion	6 cycles
Float to integer conversion (truncation)	6 cycles
Division	7 cycles
Natural logarithm	21 cycles
Exponential	17 cycles
Absolute value	0 cycles

Table 10: Required floating point operators and their latencies as generated by Quartus

As example of how such operators are implemented, we will discuss the implementation of the multiplication operator. The functions required for implementing multiplication for floating point are shown below:

```
1  type MUL = 5
2  mult_sim :: (HiddenClockResetEnable System) =>
3              DSignal dom n Float ->
4              DSignal dom n Float ->
5              DSignal dom (n + MUL) Float
6  mult_sim a b = dLay 0 ((*) <$> a <*> b)
7
8  mult_block
9    :: KnownDomain dom =>
10      Clock dom ->
11      Reset dom ->
12      Enable dom ->
13      DSignal dom n (BitVector 32) ->
14      DSignal dom n (BitVector 32) ->
15      DSignal dom (n+MUL) (BitVector 32)
16 mult_block clk rst en dataa datab =  pack <$> (exposeClockResetEnable result clk rst en)
17     where
18         result = (mult_sim (unpack <$> dataa) (unpack <$> datab))
19 {-# NOINLINE mult_block #-}
20
21 mult_f :: (HiddenClockResetEnable System) =>
22            DSig n (Maybe (Decimal)) ->
23            DSig n (Maybe (Decimal)) ->
24            DSign (n + MUL) (Maybe (Decimal))
25 mult_f in_a in_b = mux cond (Just <$> mult_result) (pure Nothing) where
26   mult_result = mult_block hasClock hasReset hasEnable mult_a mult_b
27   cond = dLay False ((&&) <$> (isJust <$> in_a) <*> (isJust <$> in_b))
28   mult_a = (fromMaybe 0b0) <$> in_a
29   mult_b = (fromMaybe 0b0) <$> in_b
```

The `mult_f` function shown in line 21 to 29 is the IP core interface for the programmer, here line 21 to 24 show the type of the function and line 25 to 29 show the implementation. The function is designed to work with `DSignal dom n (Maybe Decimal)` arguments because all floating point variables in the accelerator are of the Maybe Decimal type (section 5.2 discusses the relevant data structures). The `mult_f` function calls the mult_block function, lines 8 to 15 show the type of `mult_block` and lines 16 to 18 the implementation. This function represents the generated IP, its inputs and outputs must correspond to the inputs and outputs of the IP (the clock, reset and enable signals are applied explicitly). The `mult_block` calls the `mult_sim` function described in line 2 to 6 where line 2 to 5 describe the type of the function and line 6 describes the behaviour of the IP. The behaviour is simulated by executing the corresponding operation (in this case multiplication) and delaying the output based on the latency of the generated IP. The latency of the IP is set with the MUL type shown on line 1. Line 19 shows *{-# NOINLINE mult_block #-}*, which is required for the Clash compiler to process `mult_block` as a separate component. It will either look for a black box for this function or create a separate .vhd file for it. In our case we want to use a black box to instantiate the IP. The black box for the multiplier is shown below:

```
1  { "BlackBox" :
2          { "name"     : "Float.mult_block"
3          , "kind"     : "Declaration"
4          , "type"      : "mult_block :: KnownDomain dom
5          > Clock dom
6          -> Reset dom
7          -> Enable dom
8          -> Signal dom (BitVector 32)
9          -> Signal dom (BitVector 32)
10         -> Signal dom (BitVector 32)"
11         , "template" : "~GENSYM[Mult_Block][0] : block
12         signal tmp : std_logic;
13         begin
14         tmp <= '1' when ~ARG[3] else '0';
15             ~GENSYM[Mult_Inst][1] : SweeD_Mult PORT MAP (
16                                 aclr         => ~ARG[2],
17                                 clk_en         => tmp,
18                                 clock         => ~ARG[1],
19                                 dataa         => ~ARG[4],
20                                 datab         => ~ARG[5],
21                                 result         => ~RESULT);
22         end block ~SYM[0];"
23         }
24     }
```

The black box shown above contains 4 entries, these entries contain:

- **name** - Name of the function, in our case `mult_block` in the Float module.
- **kind** - Can be either a declaration or an expression, in our case it is a declaration.
- **type** - The type of the function.
- **template** - The code that will be inserted, the arguments such as *ARG[0]* relate to the arguments of the function. The constraints are also included as arguments, thus *ARG[0]* relates to the KnownDomain dom and *ARG[1]* is the clock signal.

5.2 Base architecture

This section describes the structural architecture that forms the basis of the accelerator. The tunable parameters and how we create parallelism is discussed in section 5.4. Based on the profiling of SweeD we know that we will accelerate the getAlpha function. The getAlpha function can be split into three big subparts, the getMinMaxAlpha function, the getClosestSNPIndex function and the nested loops that call the getLikelihood function. Based on the design space that was presented in section 4 we have decided to only accelerate the nested loops for the following reasons:

1. The access patterns to the positionsInd array performed by getMinMaxAlpha and getClosestSNPIndex are expected to significantly increase the load on the external memory interface while offering little speedup possibilities.

2. The getLikelihood function uses ∼99% of the clock cycles used by the getAlpha function in profiling runs with longer data sets.

Including getMinMaxAlpha and getClosestSNPIndex in the accelerator would increase the required area, which reduces the amount of area available for parallelising nested loops.

5.2.1 Interfacing with the accelerator

When getAlpha is called in the C code, the only argument that is passed is a position for which the likelihood of a sweep will be calculated (sweepPos). Based on sweepPos, getMinMaxAlpha is used to calculate minimum value of alpha (minAlpha) and the maximum value of alpha (maxAlpha) and getClosestSNPIndex is used to determine a starting position startPos. For the accelerator design, we assume that the accelerator is controlled by a host PC which can communicate to the accelerator. Due to the fact that we split the getAlpha function, the host PC is responsible for providing inputs for the accelerator (getAlpha calls with a sweepPos, startPos, minAlpha and maxAlpha). The accelerator contains an input buffer to store this data to be able to continuously provide input to the getAlpha circuit. The size of this input buffer has yet to be determined. The accelerator also contains an output buffer where the results of the getAlpha calls will be stored, these results will contain sweepPos with corresponding likelihood and alpha. The host PC is responsible for monitoring this buffer. A visual overview of the minimal communication between the host PC and the accelerator is shown in figure 11.

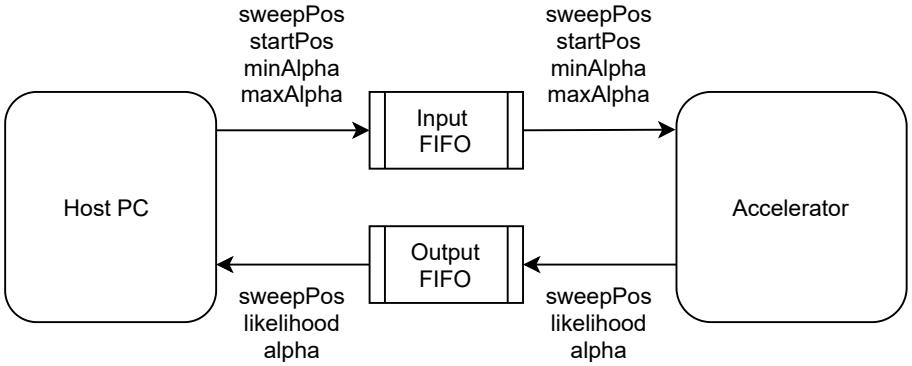


Figure 11: Interface between the host PC and accelerator

5.2.2 getAlpha structure

As described in section 5.2, the accelerator design is only implementing the nested loops of the getAlpha function. The getAlpha function contains two nested loops, we isolate the inner loop of the two loops in a seperate function such that any function contains no more than a single loop, we call this new function the getAlphaLoop function. The structural description of the getAlphaLoop function will be discussed in the next section. The C implementation of getAlpha uses arrays with 100 elements to keep track of the highest likelihoods with corresponding alpha for a single position and to keep track of the amount of probabilities calculated by the previous iteration of the inner loop. The latter will be discussed in the next section. The alpha values are used to determine the minAlpha and maxAlpha for the next while loop iteration of the getAlpha function. The C implementation of SweeD is designed to be executed on personal computers, these devices are in modern days well equipped with memory and storing this information in arrays is no problem. However for our FPGA design, we will be working on hundreds or thousands of getLikelihood calls concurrently. If we would use the same system, we would have to store these array for every call that we are concurrently processing. For that reason we introduce the following alterations:

- Instead of keeping track of 100 likelihoods and an index which indicates the highest likelihood (maxPos), we explicitly keep track of the highest likelihood.
- The C implementation requires 4 alpha values to calculate the new minAlpha and maxAlpha, the indexes of these alpha values are: maxPos - 2, maxPos - 1, maxPos and maxPos + 1. Instead of deploying large arrays and keeping track of the index maxPos, we keep track of the values that correspond to maxPos - 2, maxPos - 1, maxPos and maxPos + 1 explicitly.[6]

The accelerator has been designed such that every call to getAlpha creates large record structure with all relevant variables in Maybe context. Putting the variables in Maybe context allows them to always be Nothing in stages where the values are not required. The complete record structure is shown in appendix D. A visual representation of the structural description designed in Clash is shown in figure 12.

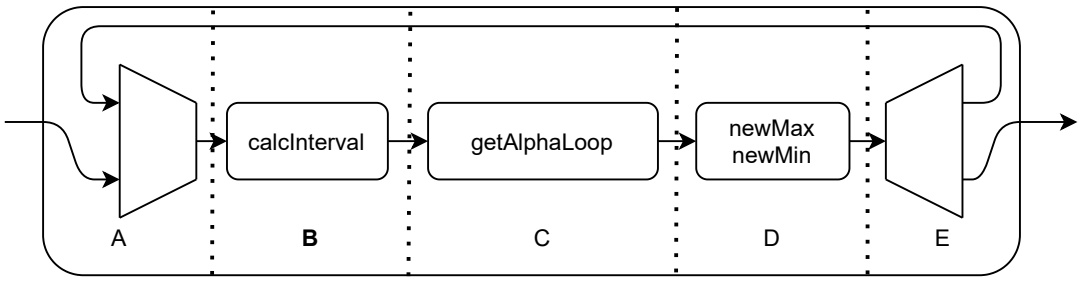


Figure 12: Structural description of getAlpha function

[6]These values represent the alpha values of the two iterations before the iteration in which the highest likelihood was found, the iteration in which the highest likelihood was found and the iteration after the highest likelihood was found.

The different functions implemented in getAlpha are explained below:

A - Merge circuit

This is the merge circuit, it contains 2 inputs for valid data and prioritizes valid data from one input over the other. We refer to the prioritized input as input 1, and the other is input as input 2. Input 1 is connected to the output of the branch circuit in section E. When valid data is present at both inputs, only one of them will be selected. This circuit returns a signal that indicates whether input 2 is selected which enables us to stall the circuit that produces this data such that the data will not be lost.

B - calcInterval

The calcInterval circuit calculates the interval between consecutive alphas for the getAlphaLoop. The equation used by calcInterval shown in equation 6.

$$newInterval = \frac{\log\left(\frac{maxAlpha}{minAlpha}\right)}{localGridSz} \tag{6}$$

C - getAlphaLoop

The getAlphaLoop structure represents the inner loop of the getAlpha function and will be discussed in section 5.2.3.

D - newMax newMin

This circuit calculates the new minAlpha and maxAlpha based on the results of the getAlphaLoop. The equations to calculate the new minAlpha and maxAlpha are shown in equation 7 and 8 respectively.

$$minAlpha = \begin{cases} maxPos == 0 & = \exp(\log(minAlpha) - (localGridSz * interval)) \\ otherwise & = alphas[maxPos - 1] \end{cases} \tag{7}$$

$$maxAlpha = \begin{cases} maxPos == localGridSz - 1 & = maxAlpha + maxAlphaDelta \\ otherwise & = alphas[maxPos + 1] \end{cases} \tag{8}$$

$$maxAlphaDelta = alphas[maxPos - 1] - alphas[maxPos - 2] \tag{9}$$

E - Branch circuit

The branch circuit routes its input to one of the outputs based on whether or not the condition for the next iteration has been met. The condition for this branch circuit is based on equation 10 is true.

$$(maxAlpha - minAlpha)/((maxAlpha + minAlpha) * 0.5) > 0.5 \tag{10}$$

If the condition has been met, valid data will be sent back to the merge circuit in section A. If not, valid data will be sent to output of getAlpha.

5.2.3 getAlphaLoop Structure

As discussed in the previous section, SweeD utilizes a large array to keep track of the amount of probabilities calculated for each iteration of the getAlpha for loop. If there were no probabilities calculated in the previous iteration, the loop will skip likelihood calculations for the remaining iterations (For the first iteration, the likelihood calculation will never be skipped). Instead of keeping track of the sweepWidth for 100 localGridSz indexes, we use the Maybe type to check if the getLikelihood function has returned a likelihood. If no likelihood is returned, it is treated as equivalent for no probabilities calculated. In our implementation we break the loop with use of the branch circuit. Figure 13 shows the structural description of the getAlphaLoop hardware.

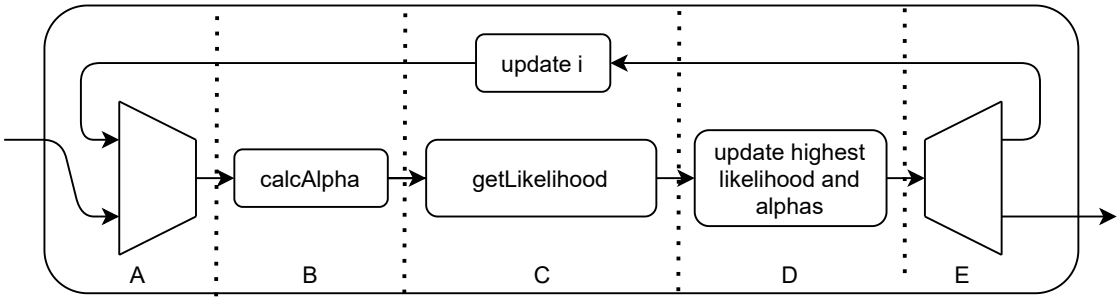


Figure 13: Structural description of getAlphaLoop

A - Merge circuit
This is the merge circuit as explained in section 5.2.2.

B - calcAlpha
Function that calculates a new alpha for this iteration between minAlpha and maxAlpha. The equation used to calculate a new alpha is shown in equation 11.

$$newAlpha = \exp(\log(minAlpha) + (i * interval)) \tag{11}$$

C - getLikelihood
The getLikelihood function which calculates a likelihood for a position based on an alpha value. This circuit will be described in the next section.

C - update i
Updates the iterator for the next iteration.

D - Update highest likelihood and alphas Based on the result of the getLikelihood function, multiple variables are updated. These variables are:

- The highest likelihood with corresponding alpha
- The alpha trackers: maxPos -2, maxPos -1, maxPos and maxPos +1 which will be required to update minAlpha and maxAlpha in the getAlpha function[7].

[7]These values were introduced in section 5.2.2

E - Branch circuit

The branch circuit as described in section 5.2.2. In this case the condition is that either the calculated CLR is Nothing, or the iterator has reached its limit. If the condition has been met, valid data will be sent back to the merge circuit in section A. If the condition has not been met valid data will be sent to output.

5.2.4 getLikelihood structure

We learned from profiling SweeD that the getlikelihood function uses about 99% of the clock cycles spent by getAlpha for data sets with long genomes. The getLikelihood function does most of the computations of the likelihood calculations and the highest speedup can be gained by accelerating this. From manual analysis we know that one of the reasons for this is the fact that getLikelihood contains two sequential loops that perform computationally expensive operations. Since the calculation performed by both loops in the getLikelihood function is the same, but only the iterator changes in a different way, we combine these loops into a single structure. The latency of this structure is 100 clock cycles (based on the latency of the current floating point operators). We can apply the efficient pipelining method described in section 2.4 to increase the performance of the nested loops. Recall that the efficient pipelining method uses the merge and branch circuits described in section 5.2.2 to concurrently process multiple iterations of the outer loop. We can apply this method because the highest likelihood has to be calculated for all grid points and these calculations contain no loop carried dependencies for different grid points. The application of this method has the added benefit that the data dependent repetition of the loops does not degrade performance. The structural overview of how the getLikelihood function is implemented in Clash is shown in figure 14.

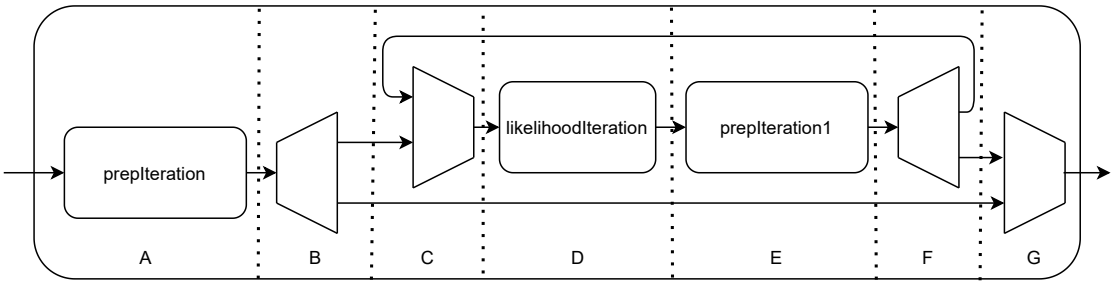


Figure 14: Structural overview of getLikelihood function without parallelism

All structural parts of architecture shown in figure 14 have been labelled and will be discussed below:

A - prepIteration
This function calculates the alpha distance for the first iteration of the first loop and checks if the conditions are met. If these conditions are not met, it calculates the alpha distance for the first iteration of the second loop and checks if the conditions are met. This function returns the data in one of 3 states:

- Data has met conditions for the first iteration of the loop.
- Data has met conditions for the first iteration of the second loop.
- Data has not met conditions for the first iteration of the first or the second loop.

The conditions that the data must meet are the following: The alpha distance must be below 12.0 and the iterator must be greater or equal to 0 and smaller than the amount of SNPs. In case that the condition is not met the first time, the input of the circuit must be stalled to calculate the alpha distance for the second loop.

B - Multiplexer
Based on the state of the data at the input, this hardware routes the data to the merge circuit in section C, or it bypasses the likelihood calculations in C, D, E and F and sends the data to the demultiplexer in G. The likelihood calculations are only bypassed if the conditions for both loops were not met.

C - Merge circuit
This is the merge circuit as described in section 5.2.2.

D - likelihoodIteration
This hardware performs a single iteration of the loops in the getLikelihood function, consisting of probability interpolation and accumulation and preperation for the next iteration. It can be considered as a $\sim$100 stage pipeline with a latency of $\sim$100 cycles.

E - prepIteration1
This hardware is similar to A - prepIteration in the sense that it performs the same task, however, this version is designed such that it calculates the alpha distance for the next iteration of the current loop and the first iteration of the second loop concurrently. This enables us to select the correct output rather than stalling likelihoodIteration to improve throughput.

F - Branch circuit
The branch circuit as described in section 5.2.2.

G - Priority demultiplexer
Functionally this hardware does the same as the merge circuit in section C, here it prioritizes the output of the branch circuit in section F.

5.3 Proposed memory access method

From profiling SweeD we know that the amount of iterations executed by the loops in getLikelihood is significantly higher than there are positions in our data set (based on calls to splint divided by length of data set, results in 11 thousand calls per position on average in our profiling runs). This means that on average every value in the positionsInd, nn, xx and baseLikelihood array is accessed ∼11 thousand times, suggesting that a lot of values are reused. Based on the access patterns described in section 4.3.3, we propose a way of prefetching the data of the arrays. Predicting indexes for consecutive accesses to positionsInd, xx, nn and baseLikelihood is rather straight forward, the proposed method for prefetching these values is as follows:

- We use blockRams as temporary storage for the contents of the positionsInd, xx, nn and baseLikelihood arrays. These blockRams are grouped into memory blocks and we keep track of an address book containing entries that specify which data is held in which memory block (e.g memory block 0 holds data for $0 <= i <= 10$).
- We monitor the output of the merge circuit. From this output we take the iterator and the direction of the sweep, if the current iterator or the next iterator is not in one of the memory blocks (e.g. next iterator is 11), the oldest address book entry will be cleared and filled with a certain window of values (e.g $11 <= i <= 21$).

The gridProbs array contains 3 dimensions, the accesses to this array rely on three things: the first index relies on the frequency of this SNP, the second index relies on the amount of samples for this SNP and the third index relies on the alpha Distance, which results in an index from 0 to 299. For a decrementing sweep, we know that in the current iteration xx[i] will be accessed. It is likely that in the next iteration xx[i-1] will be accessed. By analysing multiple datasets that were used for profiling SweeD, we found that the data in xx (the SNP frequencies) seem to have some degree of spatial locality. Figures 15 and 16 show graphs of the frequencies of the deviations between neighbouring SNPs. Mainly observe the frequency peak near a deviation of 0. This result was achieved with a python script that executes the following steps:

1. For each SNP, calculate the difference between itself and both neighbouring SNPs and store both differences in a list.
2. Determine the frequency of each difference in the list (e.g a difference of 0 occured 218 times in the data set that belongs to figure 15).

This means that there's a significant chance that the deviation of xx[i-1] and xx[i] is close to zero, thus there is spatial locality, meaning that prefetching data in a certain range based on xx[i] could prove to be useful.

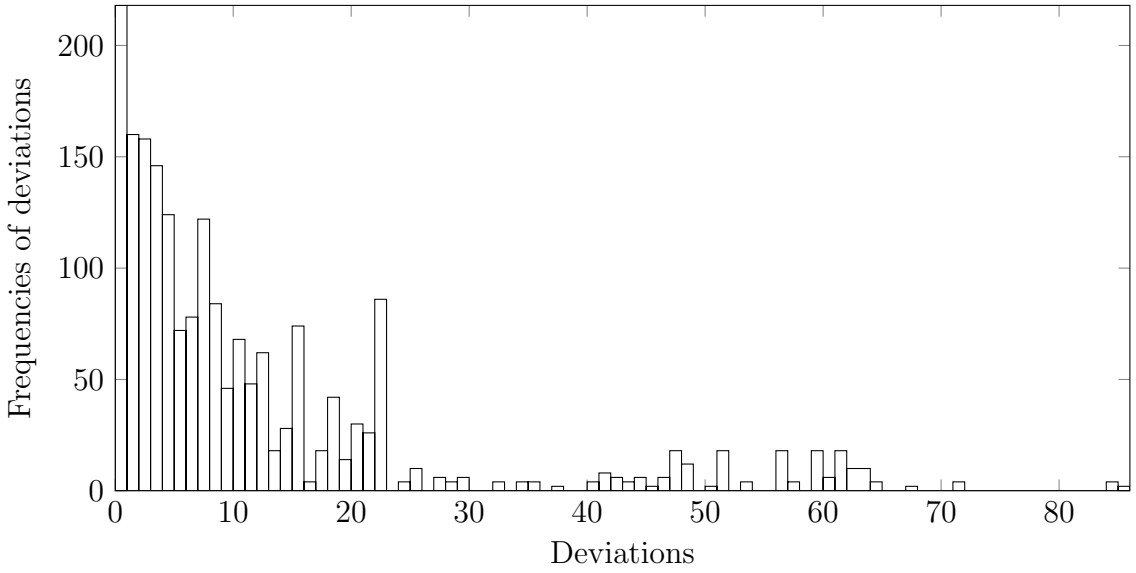


Figure 15: Frequencies of deviations between direct neighbouring SNPs, data set length 1000, samples 100

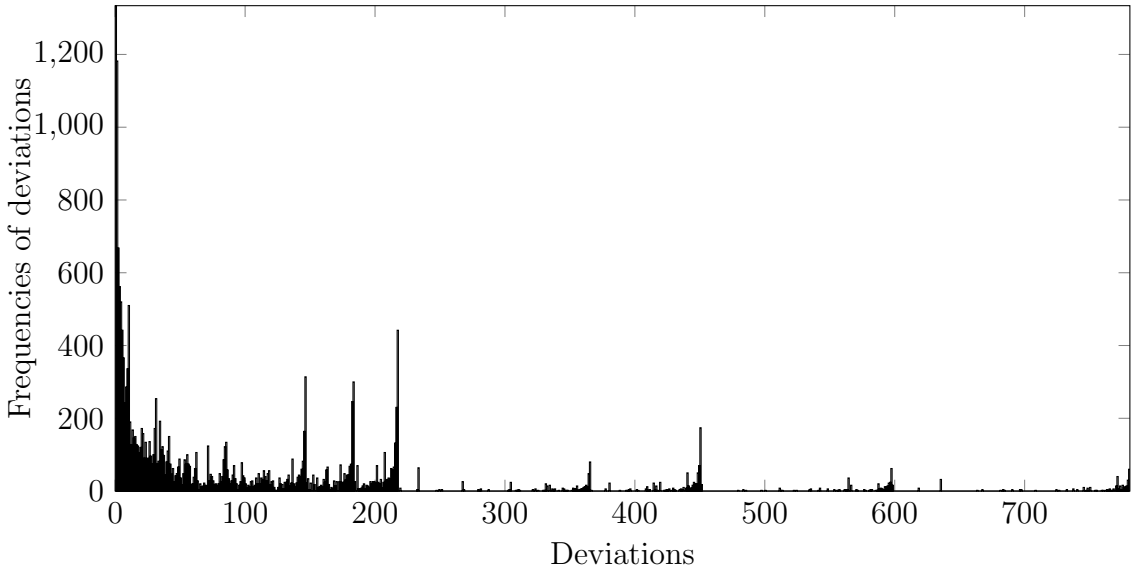


Figure 16: Frequencies of deviations between direct neighbouring SNPs, data set length 10000, samples 1000

A method of prefetching gridProbs values could be as follows:

- We use blockRams as temporary storage for the contents of the gridProbs arrays. These blockRams are grouped into memory blocks and we keep track of an address book containing entries that specify which data is held in which memory block (e.g memory block 0 holds data for $37 <= i1 <= 48$, $i2 = 100$, $0 <= i3 <= 299$ where i1 corresponds to the first index of gridProbs, i2 to the second index and i3 to the third index.).
- We monitor the values that are being prefetched from the xx[] and nn[] arrays, for each pair of values we check if it is stored in our address book. If it is not stored in our address book, we clear the oldest address book entry and prefetch a window of values around the missing entry (e.g. xx[i] = 63, nn[i] = 100 is not in our address book, fetch $57 <= i1 <= 68$, $i2 = 100$, $0 <= i3 <= 299$).

5.4 Tunable parameters

In section 4.5 we discussed methods of introducing flexibility in our design which we categorized in two categories: parameters controlling the data types that are used and parameters that control the amount of parallelism. In this section we discuss how we implemented both types of parameters.

5.4.1 Controlling data types

The variables that are used in the accelerator are either unsigned integer values or floating point values. Determining the required amount of bits for the mantissa and exponent of floating point numbers is no trivial task and changing this would require all floating point operators to be regenerated. For now the floating point variable will remain single precision 32 bit floating point. Based on table 9 we create four data types, these data types are shown in table 11. The size of these data types can be controlled using the parameters SampleBits and SitesBits which represent the minimum amount of bits required to represent the amount of sample and the minimum amount of bits required to represent the amount of SNPs respectively. We also defined the function `bitsRequired` to easily enable the user to determine how many bits are required to represent certain values. The bitsRequired function uses equation 12 to determine the amount of bits required.

Type name	Type	Description
SNPFreq	Unsigned SampleBits	Represents the SNP Frequency of an SNP.
Samples	Unsigned SampleBits	Represents the amount of samples of an SNP.
GenomeIndex	Signed (SitesBits+1)	Represents the index that is used to iterate over the SNPs.
LocalGridIndex	Unsigned 7	Represents the iterator in getAlphaLoop
Decimal	BitVector 32	Can be used for all floating point variables

Table 11: Custom data types for SweeD

$$bitsRequired = \left\lceil \frac{\log(x-1)}{\log(2)} \right\rceil \tag{12}$$

While applying this method will enable us to reduce the size of some operators. The saved area is expected to be insignificant due to the fact that most operators are floating point operators. Due to the limited time, we stick to the Signed 32 type for all signed bit variables in this thesis.

5.4.2 Controlling parallelism

To make further use of the resources offered by FPGAs we can introduce more parallelism into our design. In section 4.5.1 we discussed the problems that arise with introducing parallelism for designs with data dependent repetition and loop carried dependencies. However design presented section 5.2 presents new opportunities to introduce parallelism. The opportunity offered by this implementation is that we can concurrently process iterations of the outer loop which contains no data dependent repetition or loop carried dependencies. The biggest bottleneck in the design without parallelism arises at the merge circuit of the getLikelihood function (Figure 17 shows an abstraction of the architecture of the getLikelihood function). In our profiling runs there was an average of 358 likelihood iterations per getLikelihood call, meaning that on average every piece of valid data propagates around 358 times through the merge circuit, stalling the circuitry at its low priority input.

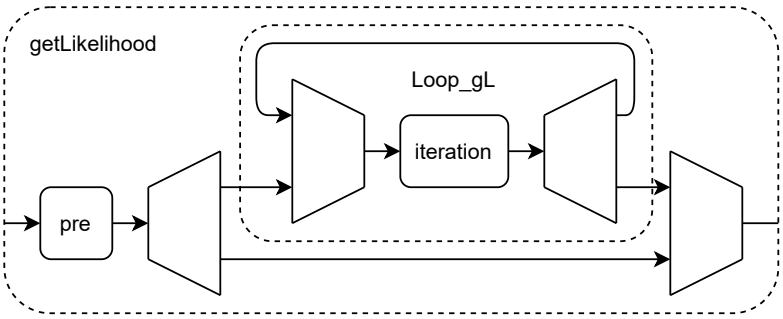


Figure 17: getLikelihood 1 instance

We introduce parallelism by creating multiple instances of the loop such that if the input of the first instance is stalled, valid data can be sent to the second instance. An example with 4 instances is shown in figure 18, here we use a demultiplexer to send data to one of the multiple instances and a multiplexer to select one of the outputs of the instances based on which instance holds valid data.

Observe how the output of the demultiplexer can only produce a single valid data input for the instances per clock cycle. In the event that multiple Loop_gL instances are ready to accept input data, only one of these instances will receive the input. This significantly impacts the performance of the accelerator since the utilization of the other Loop_gL instances drops. To prevent this drop in utilization we use FIFO buffers.

The FIFO buffers at the input of the instances prevent data starvation by accumulating sets of valid input data. These FIFO buffers have a large enough capacity to smooth out large variations in data consumption which should maximize the utilization of each loop_gL instance. Observe in figure 18 that the outputs of all instances are connected to the same multiplexer which can only process one set of valid data per clock cycle. The FIFO buffers at the output are used to prevent stalling in the case that multiple instances produce valid data concurrently. These FIFO buffers

have a large enough capacity to smooth out bursts of data production from the Loop_gL instances, this should prevent stalling of the loop_gL instances.

The amount of loop instances we can create depends on the resources available on the target device. Without parallelism, the bottleneck of this design is the amount of results Loop_gL can deliver, but as we increase parallelism the bottleneck will shift to the amount of results we can concurrently return from the getLikelihood circuit (1 result per cycle).

This concept of parallelism can also be applied at a higher level to the loops in the getAlpha function and getAlpaLoop function. Doing so enables us to increase the amount of parallelism further when the output of getLikelihood is nearing the maximum capacity (can not handle the outputs produced by the Loop_gL instances). An example of applying this method to getAlphaLoop is shown in figure 19. We identify four levels of tuning parameters, the first level controls the amount of instances of getAlpha and the second level controls the amount of instances of getAlphaLoop in each getAlpha. The third level controls the amount of instances of getLikelihood in each getAlphaLoop. The fourth level controls the amount of instances of loop_gL in getlikelihood. These parameters can be controlled by changing the parameters presented in listing 1. Currently the FIFOs are implemented with a constant size of 100 elements, however we propose to later enable the user to control the size of the FIFOs per level. We can also control the presence of the FIFOs, e.g. FIFOs are not required at levels with no extra parallelism. This can be a consideration when choosing one combination of parameters over the other.

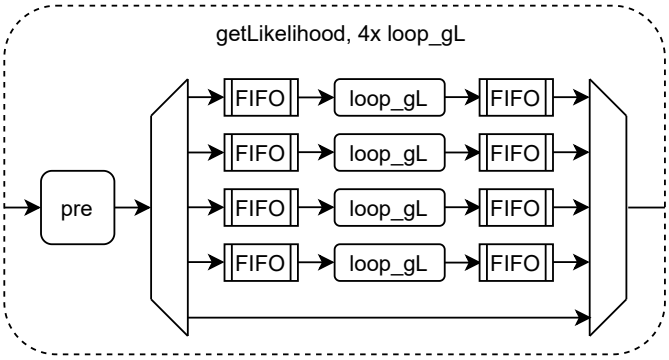


Figure 18: getLikelihood 4 instances

```
type L1Parallelism = 1
type L2Parallelism = 2
type L3Parallelism = 2
type L4Parallelism = 3
```

Listing 1: Parameters for controlling the amount of parallelism

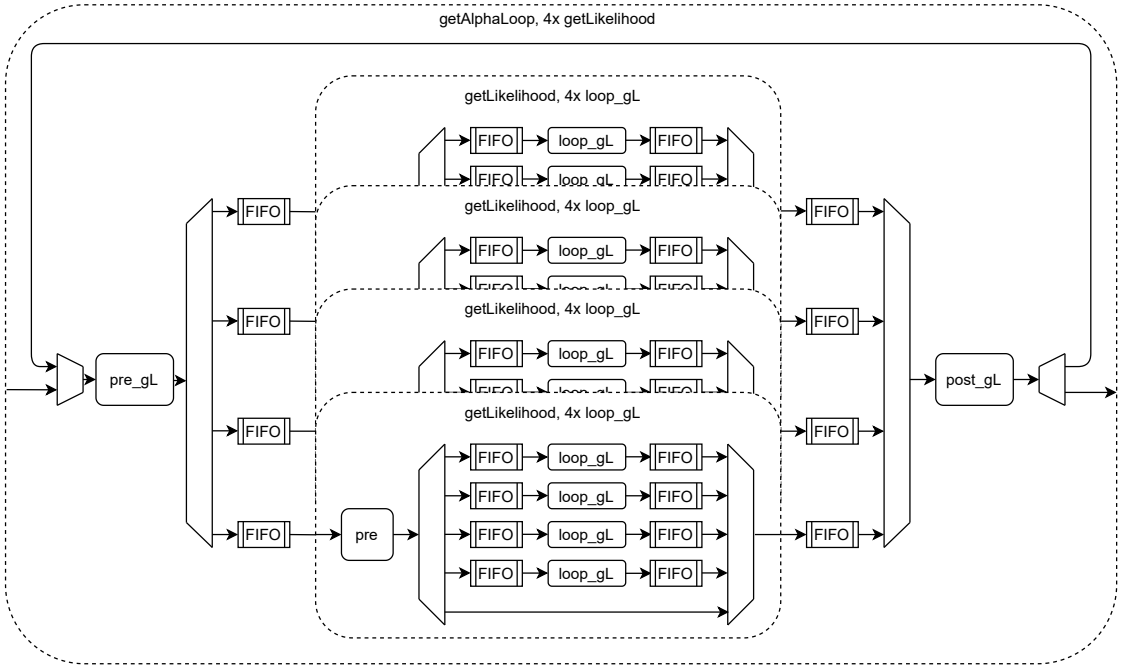


Figure 19: getAlphaLoop 4 instances of getLikelihood with 4 instances.

Concluding design remarks
Our implementation of the merge circuit can be described as a priority multiplexer that prioritizes one input over the other. Combined with stalling the circuitry that holds valid data when the input does not accept any data, this results in a data path that automatically controls the flow of data. This automatic decentralised control methodology has both advantages and disadvantages. The first great advantage of this approach is that we do not require a complex control mechanism that controls individual circuits. The second great advantage of this approach is that with the FIFOs at the input and output of each parallelised circuit, the throughput of the hardware accelerator can be maximized. The disadvantage of of this decentralised approach is that finding bottlenecks may become cumbersome. We propose to explicitly monitor the enable signals that are produced by the priority multiplexers. This could be done by counting the amount of clock cycles that each enable signal is low.

5.5 Decentralised automatic flow control

The implementation of the architecture requires a lot of stalling due to the use of the merge circuit which is in our case implemented as priority multiplexer. The priority multiplexer prioritizes its feedback input (the input it receives from the branch circuit) over the other input whenever it holds valid data. However, when both inputs contain valid data, the circuitry that produces data for the non prioritized input must be stalled in order to prevent data loss. Figure 20 shows a linear pipeline with function A, B and C, none of which contain feedback or conditional stalling. All enable inputs are controlled by the ready signal from the succeeding hardware. This ready signal is high when the succeeding hardware of function C can consume its output or when the output of function C does not hold valid data. However our architecture contains a circuit that consumes an enable signal and produces its own ready signal. Figure 21 shows an abstraction of this hardware as function D which clearly shows how function D controls the enable signal of the preceding

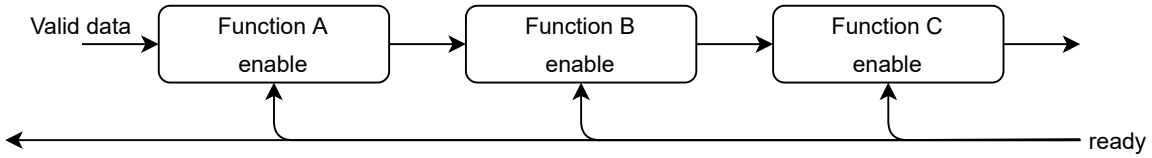


Figure 20: Linear pipeline enable control

hardware. It is unknown when function D will produce a valid output and unknown when function D will consume a valid input. Figure 22 shows an abstraction of the

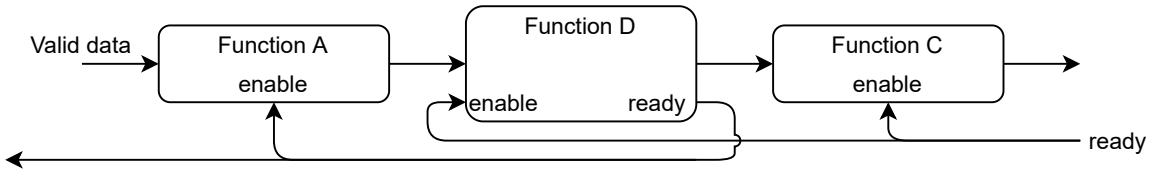


Figure 21: Nonlinear pipeline enable control

internals of function D. Function A and C are the same as in figure 20 and figure 21. The input dataIn can hold valid data or no data, the output dataOut can hold valid data or no data. Input enable indicates whether succeeding circuitry accepts data from dataOut in the next cycle. The output ready indicates whether function D reads data from dataIn in the next cycle. Assume that function E always consumes an input whenever it produces valid data. The enable of function C is high in the following cases: enable is high, output of function C does not contain valid data, function E produces valid data and dataOut does not produce valid input data. We have to be very careful to not create a circular combinational dependency where the enable of function E depends on its own ready signal while its ready signal depends on its enable signal (The latter is a given since there are cases where it can only consume data whenever it produces data).

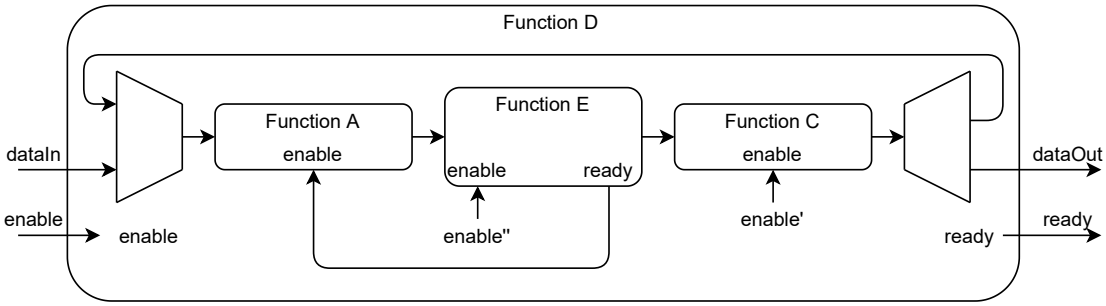


Figure 22: Visualisation of the internals of function D.

6 Evaluation

6.1 Functional correctness

To be able to use the hardware accelerator it is important that it works as expected. We test the hardware implementation by comparing it to the C implementation. The entire accelerator consists of the getAlpha circuit with input and output FIFO buffers. In the design we identify four separate circuits, the splint circuit, the getLikelihood circuit, the getAlphaLoop circuit and the getAlpha circuit. We test the correctness of these circuits by running the C implementation of SweeD for a small data set, collecting inputs for these circuits and collecting the results that correspond to these inputs. The accelerator requires access to certain lookup tables, to make these in Clash we write the contents of these lookup tables to a text file and load them into Clash. We simulate each circuit in the interactive environment with the inputs that were recorded from the C implementation. For each output we receive, we compare it to the output that was originally produced by the C implementation and calculate the deviations. We also count the amount of outputs to make sure that every input results in a single output. From all the deviations we determine the average and maximum deviation for each function. The data sets that were used for verification are generated with Hudson's ms [31]. All tests are performed on the version without parallelism to be able to verify the correctness of the calculations and whether or not data is lost. The simulations of the Clash design require a lot of memory and are not feasible on personal computers. The simulations were all executed on a server with two Xeon Gold 6126 processors and 251 gigabytes of RAM. The highest RAM usage measured exceeded 200 gigabytes, this is expected to be either a result of the feedback loops of the design or the lazy evaluation nature of Clash.

Functional correctness of likelihood interpolations

The likelihood interpolation circuit (splint) takes 3 input values and returns 1 output value. The inputs are a SNP frequency (x), amount of samples (n) and an alphaDistance (ad). The output is a probability. Given the fact that the splint circuit contains no loops, we know when to expect the result (in the test case, the result is present at the output of splint after 65 clock cycles). The splint circuit has been tested for a data set with 25 sequences of length 25. With 1525 calls to splint, an average deviation of 2.487e-8 was measured with a maximum deviation of 3.79e-5. These deviations are to be expected given that the values from the lookup tables are copied with limited precision from the C implementation to Clash and given that floating point arithmetic is non associative.

Functional correctness of getLikelihood

The getLikelihood circuit contains a loop with data dependent repetition, so the order in which the results are present at the output may be different from the order that data was sent to the getLikelihood circuit. This circuit will occasionally return negative likelihoods, these are the result of a logarithmic function. As the input of the logarithmic function approaches 0, the output approaches -infinity. Small deviations in the calculations that result from the non associative property of floating point arithmetic can lead to large deviations as result of this logarithmic function. The SweeD implementation ignores negative likelihoods and for this comparison we will change negative likelihoods to 0.0. For 13 215 inputs, we received 13 215 outputs, indicating that no data was lost. For these inputs the accelerator implementation

produced 5 543 likelihoods larger than 0.0. The average deviation that was calculated is 1.6455122e-7 and a maximum deviation is 3.4794211e-6. This indicates that the likelihoods that SweeD is trying to find are being found and the implementation will provide sufficiently similar results to the C implementation. The differences are mainly attributed to how data from the arrays in the C implementation are copied to Clash and the non associative property of floating point arithmetic.

Functional correctness of getAlphaLoop

The getAlphaLoop circuit contains multiple nested loops. The order in which the results are received from this circuit may be different from the order in which the inputs are supplied to the circuit. For each input we return the highest likelihood and compare it to the highest likelihood in the C implementation for that getAlphaLoop cycle. In a simulation of 1 000 000 clock cycles, getAlphaLoop was provided 1480 inputs and returned 1009 results with an average deviation of 1.9127324e-2 and a maximum deviation of 0.27449661. Some data is lost after the priority multiplexer when the new alpha must be calculated. This seems to be caused by incorrect application of the enable signals. A maximum deviation of around 0.27 seems too high to attribute to the non associative property of floating point arithmetic, unfortunately there was no time to investigate the root cause of this deviation. The current implementation of the getAlphaLoop circuit is currently not functionally correct.

Functional correctness of getAlpha

The functional correctness of the getAlpha function is hard to test due to the long simulation durations and the high memory requirements of the simulations. The main reason for the long simulation durations is the fact that the accelerator contains three nested loops of which the iterations are executed sequentially. The accelerator can process many data points concurrently, but it takes a long time before the results appear at the output of the accelerator. Simulating the accelerator for 500 000 cycles with 80 inputs resulted in 30 results with an average deviation of 9.640486e-2 and a maximum deviation of 1.074974. We already know that the getAlphaLoop circuit contains bugs that results in data loss, so it is no surprise that not every input results in an output. Some maxAlpha values present in the output became NaN. It is suspected that the tracking system that is applied for the alpha values instead of the tracking arrays contains bugs and leads to false results. Unfortunately there was not enough time to correct the implementation.

A summary of the evaluation is shown in table 12.

Table 12: Functional correctness results for all circuits regarding presence of data loss and deviations of calculated likelihoods.

Circuit	Supplied inputs	Received outputs	Data loss	average deviation	maximum deviation
splint	1 525	1 525	0	2.487e-8	3.79e-5
getLikelihood	13 215	13 215	0	1.645e-7	3.479e-6
getAlphaLoop	1480	1009	471	1.913e-2	2.745e-1
getAlpha	80	30	50	9.640e-2	1.075

6.1.1 Synthesizing the design for different parameters

In this section we will show the results of synthesis with different parameters for a target device. Due to the limited amount of time we only perform synthesis runs with different amounts of parallelism. As target device we chose a Cyclone V 5CGXFC9E7F35C, which offers 113,560 adaptive logic modules (ALMs), 1220 RAM blocks providing 12,49Mb of memory and 342 DSP blocks. Table 13 shows an overview of synthesis runs with varying configurations for the amount of parallelism present in the design. The amount of parallelism is indicated by the different levels (L1, L2, L3 and L4), L4 being the most fine grained level of parallelism. For each run we recorded the resources used as percentage of the amount available and whether the Fitter succeeded in fitting the design on the FPGA. The task of the Fitter is to map the compiled design to the hardware components on the chip. For all these runs we only included the FIFO buffers at level L4. The other levels contain no FIFO buffers.

L1	L2	L3	L4	ALMs %	Memory bits %	DSPs	Fits?
1	1	1	1	12,51	2,42	66 (19,30%)	Yes
1	1	1	2	17,85	4,08	122 (35,67%)	Yes
1	1	1	4	28,47	7,41	108 (31,58%)	Yes
1	1	1	8	49,96	14,05	164 (47,95%)	Yes
1	1	1	16	92,77	27,32	276 (80,70%)	No
1	1	1	15	87,08	25,69	262 (76,61%)	No
1	1	1	12	71,14	20,71	220 (64,33%)	Yes
1	1	2	8	94,73	27,36	227 (66,37%)	No
1	2	2	4	97,39	27,62	294 (85,96%)	No
2	2	2	2	112,64	28,52	342 (100,00%)	No
1	1	1	13	76,58	22,37	234 (68,42%)	Yes
1	1	1	14	81,75	24,03	248 (72,51%)	Yes
1	1	2	7	82,37	24,08	249 (72,81%)	Yes
1	2	2	3	74,78	21,05	238 (69,59%)	Yes

Table 13: Synthesis results for different configurations for the amount of parallelism in the design with FIFOs only at the inputs and outputs of L4. Resource usage is a percent age of the total resources available on the device.

From this table it can be observed that the highest level of parallelism can be achieved by setting the levels to L1 = 1, L2 = 1, L3 = 1, L4 = 14 or L1 = 1, L2 = 1, L3 = 2, L4 = 7. Quartus also reports information regarding the highest frequency (FMax) that can be used for the design in varying circumstances. Quartus reports the FMax for 4 PVT (Process, Voltage, Temperature) models, these models take in consideration the speed grade of the chip (Process), the voltage the chip runs at and the temperature of the chip. All models considered assume a voltage of 1100mV. From table 14 it can be observed that the design which utilizes multiple levels of parallelism offers a higher maximum frequency that enables us to reach higher performance. Furthermore, the ability to introduce parallelism at a higher level reduces the risk of data congestion where multiple lower level instances produce valid data at the same time that have to be processed individually, this could result in stalling.

Parallelism				Maximum clock frequencies			
L1	L2	L3	L4	Slow 85C	Slow 0C	Fast 85C	Fast 0C
1	1	1	14	20.6 MHz	20.5 MHz	46.9 MHz	51.83 MHz
1	1	2	7	29.7 MHz	29.6 Mhz	63.47 MHz	72.8 MHz

Table 14: Maximum frequencies for different PVT models with different two different parallelism configurations

A direct consequence of adding parallelism and FIFOs to our design is that the accelerator will process more grid points concurrently. We estimated that the design with the highest frequency processes around 3100 grid points concurrently[8]. A direct consequence of this concurrency is that the accelerator can only reach full utilization when there are enough grid points to be processed.

6.2 Estimating speedup upper limit

In this section we estimate the upper limit of potential speedup of the design by estimating the throughput of our accelerator for a set of parameters and comparing it to the throughput of the C implementation. The throughput of the hardware accelerator is based on the maximum amount of parallelism that we can achieve. To estimate the upper limit for the throughput on this accelerator, we make the following assumptions:

- **Maximum utilization of fine grained loops** - Due to the processing bottleneck imposed by the fine grained loops, valid data can accumulate at the inputs in the FIFO buffers. We assume that the direct consequence of this is the fact that there is always valid data available at the inputs, preventing data starvation. We also assume that the FIFO buffers at the outputs of the fine grained loops have enough capacity to prevent the necessity of stalling. Under these assumptions we can obtain the maximum throughput where each instance of a level 4 loop can produce one valid iteration per clock cycle due to its pipeline design.

- **Ideal memory** - To estimate the maximum theoretical speedup we will assume ideal memory such that there is no performance restriction imposed by a memory subsystem. Each valid iteration mentioned above requires six variables that need to be fetched from memory. With 14 times parallelism this would result in 84 accesses per iteration. At a maximum frequency of 72.8 MHz this ideal memory would be supplying 195 Gigabits of data per second.

The throughput of the hardware accelerator will be compared to the throughput delivered by the C implementation. We will estimate the throughput of the C implementation by using the profiling results to determine amount of clock cycles spent on the same functionality and using the total execution time of a separate run with the same parameters to determine the time spent on this functionality. Unfortunately, the computational cost of the functions in getAlpha that where excluded from the accelerator design (getMinMaxAlpha and getClosestSNPIndex) can not be

[8]Consider the design to be a pipeline with feedback loops and stalling capacity. L4 contains 112 stages, L3 contains (L4Parallelism * L4) + 12 stages, L2 contains (L3Parallelism * L3) + 46 stages, L1 contains (L2Parallelism * L2) + 80 stages, total contains (L1Parallelism * L1) stages, it is unlikely that all FIFOs will be filled at one point, so our estimate is aimed at full input buffers and empty output buffers as a heuristic.

explicitly retrieved. Instead we will consider the computational cost used by the whole getAlpha function. Given the fact that in the profiling runs, the getLikelihood function consumed >99% of the clock cycles spent by getAlpha this is not expected to have a significant impact. Table 15 shows the estimated time spent on the likelihood calculations by the processor. The sequential version of SweeD was used which ran on a system that contains two Xeon Gold 6126 CPUs running at 2.6Ghz with 256 GiB RAM.

Sequences	Samples	Exec. Time(s)	Total cycles	getAlpha %	getAlpha time (s)
10000	10	41,38	2,86E+11	99,9	41,36
10000	100	43,16	2,80E+11	99,4	42,91
10000	1000	69,78	4,99E+11	54,4	37,97

Table 15: Execution time estimation of the getAlpha function based on execution time and profiling results.

The accelerator has been designed such that under the assumptions presented above we can perform one iteration of the level 4 loop per clock cycle per level 4 instance. Given the fact that we can instantiate 14 instances on a Cyclone V 5CGXFC9E7F35C at 72.8 MHz in the best case, we can process up to $1.02 * 10^9$ iterations per second. Table 16 shows the time estimated for the accelerator to process these iterations. We know that each iteration results in one call to the splint function and from profiling we know how many calls to the splint function were made each run. Thus based on the amount of calls to splint we know the amount of iterations to process and based on the amount of iterations we can process per clock cycle and the maximum clock frequency of 72.8 MHz we can estimate an upper bound for the runtime of the accelerator on this chip. This estimate is based on the assumption that we can maintain maximum utilization for the whole duration of the likelihood calculations. However, we know that this will never be the case since at the beginning of the likelihood calculations since the FIFO buffers and pipeline stages are all empty. How fast the utilization will rise to the maximum will depend on the speed at which the accelerator can be supplied data to process. The utilization is also expected to drop at the end of the likelihood calculations from the moment that the accelerator stops receiving new input data. The speed and intensity at which the utilization will drop is currently unknown. The upper limit of the speedup is calculated by dividing the estimated time of the processor by the estimated time of the accelerator (e.g. $\frac{41,36}{0,46} \approx 89$) .

Sequences	Samples	grid points	splint calls	est. time (s)	speedup upper limit (x)
10 000	10	8 000	472 045 078	0,46	89,30
10 000	100	8 000	468 975 078	0,46	93,25
10 000	1 000	8 000	469 663 644	0,46	82,40

Table 16: Estimating the execution time of the accelerator to determine the speedup with a Cyclone V FPGA.

The Cyclone V FPGA for which the design was synthesised was released back in 2011 (∼10 years before time of writing) and integrated circuit design has come a long way since then. The reason the design was synthesized for this chip is a limitation

of the available Quartus license. One of the most recently released FPGAs from Intel is the Stratix 10 GX 10M FPGA (released in 2019), which offers $\sim$ 30 times as many ALMs, $\sim$24 times as much memory and $\sim$10 times as many DSPs, we can not blatantly expect to be able to introduce 10 times as much parallelism to this chip, this would need to be researched. However it is reasonable to expect a significant improvement on newer chips. Furthermore the accelerator design has not yet been analysed for potential frequency bottlenecks, potentially a higher FMax could be reached with some modifications. At the beginning of the section we assumed ideal memory. The feasibility of this accelerator design heavily relies on the memory management system which has not been implemented.

7 Discussion

The likelihoods that are being calculated by the getLikelihood circuit are expected to deviate due to the non associative nature of floating point arithmetic [35]. The getLikelihood circuitry shows no data loss, indicating that the method of stalling and using loops works. However, when simulating the getAlphaLoop circuit the first indications of data loss appear that could not be fixed. Completely simulating the hardware accelerator for larger data sets still proves to be rather difficult due to the high memory consumption and long runtimes of the simulations, the reason for this is still unknown. The interactive Read Evaluate Print Loop (REPL) offered by Clash is otherwise still very capable at providing a way to quickly test new functionality separately. Also the method of stalling circuitry to preserve valid data has been shown to work through testing on the getLikelihood function (since we received an output for each input that was given). However, correctly controlling all enable signals for circuits with nested loops still proved to be rather difficult.

Adjusting the amount of parallelism in the system is nearly effortless. However, the user has to synthesize the design multiple times to check whether the configuration fits on the target device. Unfortunately even after extensive testing in the C implementation, no suitable fixed point representation was found. For this project we chose to use floating point operators which are implemented through IPs. It is relatively easy, though time consuming to generate new IPs, but the method is not vendor bound. The design makes use of BlockRam, while this is a specific technology, it is expected to not pose a problem due to the fact that it is widely adopted.

We managed to determine an upperbound of the speedup that can be offered by this accelerator on a Cyclone V FPGA, the upperbound is around $\sim$86 times faster than the sequential version of SweeD running on a Xeon Gold 6126 running at 2.6 GHz. This is a significant speedup, especially considering the fact that the FPGA is $\sim$10 years old at the moment of writing. Note that this only concerns the likelihood calculations since the other functionality is not accelerated. Using other target devices is possible, but then IP cores must be generated for the target device. The actual great advantage of this design is the fact that you can freely and easily change the amount of parallelism without losing synchronisation. The estimated upper limit assumes ideal memory that does not take up space, does not have latency and has infinite capacity. Of course in the real world no such thing exists. A memory access solution has been proposed, but implementing an actual memory interface was outside the scope of this thesis. The final speedup of the accelerator will depend on how the data from the arrays is provided.

8 Conclusion

Software tools for selective sweep detection have become important in the bioinformatics toolbox to analyze genetic data. The goal of this thesis was to accelerate selective sweep detection with dedicated hardware due to the long typically long runtimes of these types of tools. The tool that was accelerated is Sweep Detector (SweeD) which makes use of SFS based CLR calculations. One of the main focuses of this thesis was to create a flexible design that can be adjusted to different FPGA setups such that it is usable with available setups. The functional HDL Clash was used because of the mathematical nature of the problem and the ease of implementing flexibility due to its high level abstractions. To guide the project, research questions have been formed. We will be answering these questions based on the findings in this thesis. For each research question we will first answer the sub-questions.
Research sub-question 1a:

1a. Which part of the algorithm should be accelerated?

It is trivial that, to gain the highest speedup while making an accelerator, we must accelerate the part that consumes most time. By profiling SweeD with Callgrind we managed to identify the two most computationally expensive parts. These parts were data preparation and likelihood calculations. Ideally both should be accelerated given the fact that they both require a lot of computation time, but for this thesis we chose to accelerate the likelihood calculations. The main reasons for this are the large amount of time consumed by the likelihood calculations as shown by the profiling results and the great amount of repetition. Within the likelihood calculations we found parts that were suitable for acceleration and parts that were not suitable for acceleration. We found that the parts that should be accelerated are calculating likelihoods for given a alpha value and finding the alpha value that results in the highest likelihood for points on the genome. The part that should not be accelerated is finding the starting point of the selective sweep and finding the minimum and maximum alpha. These functions did not significantly contribute to the runtime of SweeD and their memory access patterns would introduce a significant load on the memory interface.

Research sub-question 1b:

1b. How can we reduce the area of the implementation without significantly impacting the precision?

The data type that is used for real numbers in SweeD is the double precision floating point data type. This uses 64 bit floating point values. It is already known that floating point operations do not map well to FPGA fabric compared to fixed point operations. Furthermore, double precision floating operators in FPGAs require a significant amount of resources as consequence of this. Hardware designers often use the concurrent nature of FPGAs to create multiple instances of the same hardware to process multiple data streams in parallel. This leads to a higher resource usage based on how much resources a single instance requires. If we can reduce the resources required for a single instance, we can create more parallelism with the same amount of resources. We found that the data type of the decimal values can be reduced to at least single precision floating point without significantly affecting the precision of the results. Ideally we use fixed point values, but we determined that this was not a suitable solution because we were not able to produce meaningful

results with up to 64 bit fixed point values.

Main research question 1:

1. How can we design a technology-independent hardware accelerator that significantly speeds up SweeD using Clash

One of the main obstacles encountered while accelerating SweeD is that the likelihood calculations contain multiple nested loops with loop carried dependencies and data dependent repetition. By implementing tailored versions of the merge circuit, branch circuit en blocking concept presented by Styles et al. [2] we were able to concurrently process iterations of the outermost loop which contains neither data dependent repetition or loop carried dependencies. By stalling the circuitry at the non prioritized input whenever it holds valid data but is not selected, we ensure that data is not getting lost. We applied this method at any point in the hardware that may require stalling to create an automatic decentralized control system for the hardware. The main question remains partly unanswered because the hardware accelerator requires a memory management system that can provide the hardware with data with minimum stalling. While there is a proposed solution that utilizes prefetching to minimize stalling, no such memory system has been implemented. An estimate for the upper limit speedup was made assuming ideal memory. For a Cyclone V FPGA (2011) we were able to instantiate 14 parallel instances of the lower level loops with an maximum achievable clock frequency of 72.8 MHz. Based on this we estimated a speedup of around 89 times compared to the sequential version of SweeD running on a Xeon Gold 6125 (2017). Whether this speedup is actually achievable highly depends on the memory management system. The ideal memory that was assumed would be supplying 195 gigabits of data from the arrays, since no single data bus can provide that kind of throughput it is definitely necessary to create a temporary on chip storage solution.

Research sub-question 2a:

Where can we introduce extra parallelism in the design?

Usually parallelism can be introduced at points where the same function has to be executed for multiple data sets. In our case the only place where this occurs are the loops. The data dependent repetition and loop carried dependencies prevent efficiently processing multiple iterations of the same loop at the same time. However, we can concurrently process outer loop iterations. We know that the inner most loop stalls the non prioritized input of the multiplexer most of the time, this creates a bottleneck. We can create parallelism at these bottlenecks where we can process even more outer loop iterations in an efficient manner.

Research sub-question 2b:

Which parameters can we offer to easily tune the design?

Determining which parameters to offer is important because it defines how the user can adjust the design to its preference. We found that initially there is not much for the user to control regarding the functionality of the algorithm. The most important parameters that we introduced are for varying the amounts of parallelism at different granularities and controlling the presence and sizes of the FIFO buffers to achieve the highest speedup.

Main research question 2:

How can we create flexibility in our design such that it is easily adjustable for different target devices?

We create flexibility by enabling the user to control the amount of parallelism at different granularities to obtain the highest throughput possible with the available resources. Every parallel instance contains FIFOs of which we can control the presence and the size. We managed to create a design for a Cyclone V FPGA where we could instantiate 14 parallel instances of the inner most loop, combined with the highest possible FMax and some assumptions regarding the availability of data we determined an upper limit of speedup of around 86 times for this FPGA compared to the sequential version of SweeD. The current implementation utilizes floating point operator that are generated by Quartus for a certain chip FPGA family. However, any other set of pipelined FPGA operators can be used to adjust the design for different FPGAs. The latency of the operators can be set per operator in Clash and Clash will automatically ensure that all data is synchronised when the latency is changed.

9 Recommendations

9.1 Memory management system

As discussed in the conclusion, the final performance of the accelerator will heavily rely on the memory management system. It is recommended to develop a system that uses a smart tailored method using for example caching or prefetching to temporarily store data on chip. It is recommended to consider the prefetching concept described in section 5.3. Section C goes into depth regarding the most relevant data dependencies.

9.2 Floating point operators in Clash

At the moment of writing, floating point operators are not supported by Clash. This problem was circumvented by generating IPs with the vendor tool that is used for synthesized and making adjustments to Clash such that these can be used in the code. How these are currently implemented is shown in section 5.1.3. A more general solution could be to use FloPoCo [36]. FloPoCo is a project aimed at creating a generator for floating point operators for FPGAs. At a glance, this project seems to be highly suitable for replacing the vendor generated IP cores. However, recent Clash documentation files regarding version 1.5.0 mentions a "-fclash-float-support" flag that hints at upcoming floating point support. When it will actually be released and whether or not it will be suitable for the current implementation is unknown. When going forward with the current approach of using custom IP cores, it is proposed to use visible type application to delay one of the inputs to match the others for operators with two inputs. This will eliminate the need to explicitly delay variables in equations as Clash will handle all of the delays appropriately. An example of applying visible type applications to a delayed multiplication is shown below:

```
multiply ::
  forall m n dom a .
  (Num a, HiddenClockResetEnable dom, NFDataX a, KnownNat m, KnownNat n) =>
  DSignal dom m a ->
  DSignal dom n a ->
  DSignal dom (Max m n + 3) a

multiply a b = result
    where
      result = delayN d3 0 ((*) <$> inA <*> inB
      inA = delayI @(Max m n - m) 0 a
      inB = delayI @(Max m n - n) 0 b)
```

For example when m = 3 and n = 1, inA will be delayed `((Max 3 1) - 3)` = 0 cycles. inB will be delayed `((Max 3 1) - 1)` = 2 cycles. Using this method will make designing a lot easier because it will not be necessary anymore to explicitly delay the different inputs of functions that take multiple inputs.

9.3 Generator Accumulator loops

One of the main problematic aspects of the likelihood calculations are the loop carried dependencies. The only loop carried dependency present in the level 4 loop is the accumulation of likelihoods. The accumulator variable is not necessary for the majority of the operations and only accumulates the calculated likelihoods. The circuitry that calculates the likelihoods can be considered as a pipeline with about 112 stages. Through the pipeline we route all variables that are required in the accelerator, which can lead to high area usage due to the fact that we route a lot of variables through these stages without them being required for the computation. We propose a conceptual generator accumulator hardware loop for the level 4 loop in which the generator circuit iterates over positions of the genome similarly to the C implementation, propagating to the left from the starting position and later to the right of the starting position. For each position it generates an alphadistance value. As long as the alphadistance is below 12.0 and the iterator is within range, calls will be made to perform a likelihood calculation for this position. These calls will be accumulated in the accumulator. As long as the circuit is working on the same sweep position, the values that are not required for the calculation can be stored in a dedicated register, eliminating the need to store values for every stage. When there are no more likelihoods to be calculated (the conditions are not met), the iteration generator starts working on the next iteration. When a new sweep position arrives at the input of the accumulater, the accumulater produces an output containing the total likelihood alongside the variables that belong to the pipe. This method could severely reduce area and potentially allow for more parallelism, improving performance. A visual representation of the circuit is shown in figure 23.

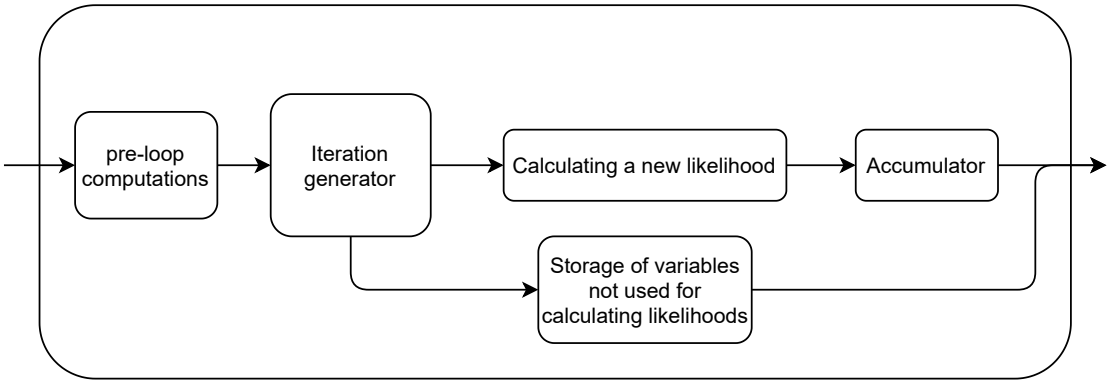


Figure 23: The generator / accumulator circuit proposed to eliminate storage of unused variables for all stages of the likelihood calculation pipeline.

9.4 Combining input / output FIFOs

Currently the loop circuits feature a FIFO buffer at their input and at their output to be able to constantly provide input data and to minimize stalling. Due to the design of the priority multiplexer and branch circuit we know that each time the loop circuit produces a new output, it also consumes an available input from the FIFO. We can reduce the amount of FIFO buffers that is required by combining the input and output buffer. The way we can do this is by writing the output to the position of the consumed input as shown in figure 24. In this FIFO we use three trackers that keep track of the first element of a set of valid inputs, valid outputs or empty addresses. The empty tracker is used to keep track of the next address where input data can be written to. The valid input tracker keeps track of the first address containing valid input data. The valid output tracker keeps track of the first address containing valid output data. By combining these FIFOs we expect to be able to make a significant reduction in required resources.

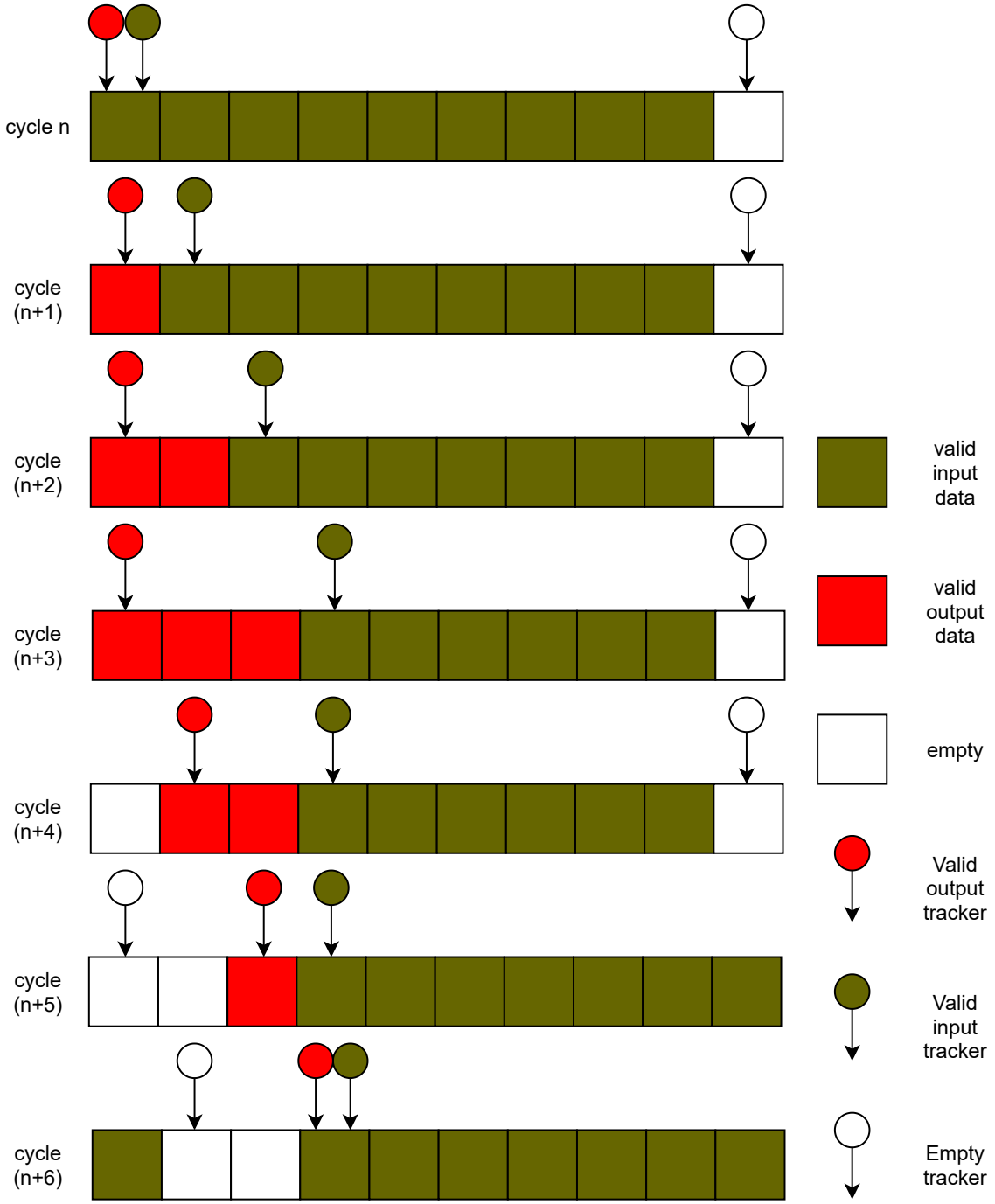


Figure 24: Method of using a multi purpose FIFO that stores the output of the loop when an input is consumed. Shown are 7 cycles wherein the first three cycles an input is consumed and replaced by the output of the circuit. In the next three cycles the outputs are sent to the output of the FIFO.

References

[1] P. Pavlidis, D. Živković, A. Stamatakis, and N. Alachiotis, "Sweed: likelihood-based detection of selective sweeps in thousands of genomes," *Molecular biology and evolution*, vol. 30, no. 9, pp. 2224–2234, 2013.

[2] H. Styles, D. B. Thomas, and W. Luk, "Pipelining designs with loop-carried dependencies," in *Proceedings. 2004 IEEE International Conference on Field-Programmable Technology (IEEE Cat. No.04EX921)*, 2004, pp. 255–262.

[3] "Dna vs. rna – 5 key differences and comparisons, author = Ruairi J Mackenzie, Technology Networks, howpublished = https://www.technologynetworks.com/genomics/lists/what-are-the-key-differences-between-dna-and-rna-296719, note = Accessed: 17-06-2021."

[4] R. Nielsen, S. Williamson, Y. Kim, M. J. Hubisz, A. G. Clark, and C. Bustamante, "Genomic scans for selective sweeps using snp data," *Genome research*, vol. 15, no. 11, pp. 1566–1575, 2005.

[5] L. Kang, G. He, A. K. Sharp, X. Wang, A. M. Brown, P. Michalak, and J. Weger-Lucarelli, "A selective sweep in the spike gene has driven sars-cov-2 human adaptation," *bioRxiv*, 2021.

[6] N. K. Biswas and P. P. Majumder, "Analysis of rna sequences of 3636 sars-cov-2 collected from 55 countries reveals selective sweep of one virus type," *The Indian journal of medical research*, vol. 151, no. 5, p. 450, 2020.

[7] N. H. Barton, "Genetic hitchhiking," *Philosophical Transactions of the Royal Society of London. Series B: Biological Sciences*, vol. 355, no. 1403, pp. 1553–1562, 2000.

[8] Å. Johansson and U. Gyllensten, "Identification of local selective sweeps in human populations since the exodus from africa," *Hereditas*, vol. 145, no. 3, pp. 126–137, 2008.

[9] M. Pybus, P. Luisi, G. M. Dall'Olio, M. Uzkudun, H. Laayouni, J. Bertranpetit, and J. Engelken, "Hierarchical boosting: a machine-learning framework to detect and classify hard selective sweeps in human populations," *Bioinformatics*, vol. 31, no. 24, pp. 3946–3952, 2015.

[10] N. Alachiotis, A. Stamatakis, and P. Pavlidis, "Omegaplus: a scalable tool for rapid detection of selective sweeps in whole-genome datasets," *Bioinformatics*, vol. 28, no. 17, pp. 2274–2275, 2012.

[11] P. Pavlidis and N. Alachiotis, "A survey of methods and tools to detect recent and strong positive selection," *Journal of Biological Research-Thessaloniki*, vol. 24, no. 1, pp. 1–17, 2017.

[12] N. Alachiotis and P. Pavlidis, "Raisd detects positive selection based on multiple signatures of a selective sweep and snp vectors," *Communications biology*, vol. 1, no. 1, pp. 1–11, 2018.

[13] N. Alachiotis, C. Vatsolakis, G. Chrysos, and D. Pnevmatikatos, "Raisd-x: A fast and accurate fpga system for the detection of positive selection in thousands of genomes," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 13, no. 1, pp. 1–30, 2019.

[14] C. Baaij, M. Kooijman, J. Kuper, A. Boeijink, and M. Gerards, "C? ash: Structural descriptions of synchronous hardware using haskell," in *2010 13th Euromicro Conference on Digital System Design: Architectures, Methods and Tools*. IEEE, 2010, pp. 714–721.

[15] P. W. Hedrick, "Population genetics of malaria resistance in humans," *Heredity*, vol. 107, no. 4, pp. 283–304, 2011.

[16] N. M. o. N. H. Smithsonian, "Genetic evidence," https://humanorigins.si.edu/evidence/genetics, accessed: 04-05-2021.

[17] Xilinx, "Vitis unified software development platform 2020.2 documentation," https://www.xilinx.com/html_docs/xilinx2020_2/vitis_doc/introductionvitishls.html, accessed: 10-06-2021.

[18] T. Bollaert, "Catapult synthesis: A practical introduction to interactive c synthesis," in *High-Level Synthesis*. Springer, 2008, pp. 29–52.

[19] D. Petkov, R. Harr, and S. Amarasinghe, "Efficient pipelining of nested loops: unroll-and-squash," in *Proceedings 16th International Parallel and Distributed Processing Symposium*, 2002, pp. 6 pp–.

[20] M. DeGiorgio, C. D. Huber, M. J. Hubisz, I. Hellmann, and R. Nielsen, "Sweepfinder2: increased sensitivity, robustness and flexibility," *Bioinformatics*, vol. 32, no. 12, pp. 1895–1897, 2016.

[21] J. M. Smith and J. Haigh, "The hitch-hiking effect of a favourable gene," *Genetics Research*, vol. 23, no. 1, pp. 23–35, 1974.

[22] J. M. Braverman, R. R. Hudson, N. L. Kaplan, C. H. Langley, and W. Stephan, "The hitchhiking effect on the site frequency spectrum of dna polymorphisms." *Genetics*, vol. 140, no. 2, pp. 783–796, 1995.

[23] Y. Kim and R. Nielsen, "Linkage disequilibrium as a signature of selective sweeps," *Genetics*, vol. 167, no. 3, pp. 1513–1524, 2004.

[24] M. Slatkin, "Linkage disequilibrium—understanding the evolutionary past and mapping the medical future," *Nature Reviews Genetics*, vol. 9, no. 6, pp. 477–485, 2008.

[25] J. L. Crisci, Y.-P. Poh, S. Mahajan, and J. D. Jensen, "The impact of equilibrium assumptions on tests of selection," *Frontiers in genetics*, vol. 4, p. 235, 2013.

[26] D. Bozikas, N. Alachiotis, P. Pavlidis, E. Sotiriades, and A. Dollas, "Deploying fpgas to future-proof genome-wide analyses based on linkage disequilibrium," in *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2017, pp. 1–8.

[27] N. Alachiotis, C. Vatsolakis, G. Chrysos, and D. Pnevmatikatos, "Accelerated inference of positive selection on whole genomes," in *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2018, pp. 202–2027.

[28] G. Charitopoulos, C. Vatsolakis, G. Chrysos, and D. N. Pnevmatikatos, "A decoupled access-execute architecture for reconfigurable accelerators," in *Proceedings of the 15th ACM International Conference on Computing Frontiers*, 2018, pp. 244–247.

[29] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," *ACM Sigplan notices*, vol. 42, no. 6, pp. 89–100, 2007.

[30] V. Developers, "Callgrind: a call-graph generating cache and branch prediction profiler," https://valgrind.org/docs/manual/cl-manual.html, accessed: 10-06-2021.

[31] R. R. Hudson, "ms a program for generating samples under neutral models," 2004.

[32] A. N. Hirani, "Fine precision," https://faculty.math.illinois.edu/~hirani/cbmg/precision.html, accessed: 01-02-2021.

[33] A. Finnerty and H. Ratigner, "Reduce power and cost by converting from floating point to fixed point," https://www.xilinx.com/support/documentation/white_papers/wp491-floating-to-fixed-point.pdf, 2017, accessed: 05-02-2021.

[34] I. Voras, "Fixed point math library for c," https://sourceforge.net/p/fixedptc/code/ci/default/tree/, accessed: 04-02-2021.

[35] D. Goldberg, "What every computer scientist should know about floating-point arithmetic," *ACM computing surveys (CSUR)*, vol. 23, no. 1, pp. 5–48, 1991.

[36] F. de Dinechin and B. Pasca, "Designing custom arithmetic data paths with FloPoCo," *IEEE Design & Test of Computers*, vol. 28, no. 4, pp. 18–27, Jul. 2011.

A Alternative implementations

The C version of SweeD That operates with floats in the getLikelihood function as made by the following changes:

- A new data structure called "alignment_f" has been made that contains the following all data from the "alignment" structure that is stored as doubles and used in the getLikelihood function. The "alignment_f" data structure however, stores this data in the float format.
- Memory allocation for the "alignment_f" data components is done for each component directly after memory allocation for its corresponding "alignment" component. (Same goes for freeing memory)
- Filling the "alignment_f" data structure is done through a function named "Fill_Alignment" which is called before the for loop where the getAlpha function is called.
- The getLikelihood function is adjusted to use the floating point values from "alignment_f" and change all functions to their floating point version (mainly changing log(x) to log_f(x)).
- The result of the getLikelihood function is cast to double for the rest of the program to process.

B Profiling results

			Main			CreateProbs			createSFS			getAlpha				getLikelihood				splint			
Length	Samples	Grid	calls	%	cycles	calls	% / Main	cycles	calls	% Cycles	cycles	calls	% / Main	cycles	cycles/call	calls	% / getAlpha	cycles	cycles/call	calls	% / getAlpha	cycles	cycles/call
100	10	20	1	100	4,15E+07	1	33,59	1,39E+07	1	5,68	2,36E+06	20	58,44	2,43E+07	1,21E+06	3,70E+03	94,61	2,29E+07	6,20E+03	4,42E+04	44,29	1,02E+07	230
100	10	50	1	100	7,34E+07	1	19,00	1,39E+07	1	3,21	2,36E+06	50	76,06	5,58E+07	1,12E+06	9,09E+03	94,25	5,26E+07	5,79E+03	1,03E+05	44,86	2,36E+07	230
100	10	80	1	100	1,04E+08	1	13,37	1,39E+07	1	2,26	2,36E+06	80	82,86	8,65E+07	1,08E+06	1,46E+04	94,05	8,13E+07	5,57E+03	1,58E+05	44,66	3,63E+07	230
100	100	20	1	100	1,37E+09	1	86,92	1,19E+09	1	11,89	1,63E+08	20	1,07	1,47E+07	7,34E+05	3,32E+03	91,29	1,34E+07	4,03E+03	2,52E+04	43,34	5,81E+06	231
100	100	50	1	100	1,39E+09	1	85,49	1,19E+09	1	11,70	1,63E+08	50	2,67	3,71E+07	7,42E+05	8,37E+03	91,36	3,39E+07	4,05E+03	6,29E+04	42,80	1,45E+07	230
100	100	80	1	100	1,41E+09	1	84,21	1,19E+09	1	11,53	1,63E+08	80	4,11	5,80E+07	7,24E+05	1,33E+04	91,13	5,28E+07	3,97E+03	9,61E+04	41,95	2,22E+07	231
100	1000	20	1	100	2,01E+11	1	94,07	1,89E+11	1	5,92	1,19E+10	20	0,01	1,50E+07	7,48E+05	3,07E+03	91,45	1,37E+07	4,46E+03	2,62E+04	44,15	6,04E+06	230
100	1000	50	1	100	2,01E+11	1	94,06	1,89E+11	1	5,92	1,19E+10	50	0,02	3,61E+07	7,23E+05	7,53E+03	91,16	3,30E+07	4,38E+03	6,23E+04	43,54	1,43E+07	230
100	1000	80	1	100	2,01E+11	1	94,05	1,89E+11	1	5,92	1,19E+10	80	0,03	5,64E+07	7,05E+05	1,21E+04	90,94	5,13E+07	4,22E+03	9,62E+04	43,24	2,22E+07	231
1000	10	20	1	100	9,20E+08	1	1,51	1,39E+07	1	0,32	2,99E+06	200	97,55	8,97E+08	4,49E+06	3,19E+04	98,20	8,81E+08	2,76E+04	1,59E+06	41,50	3,66E+08	230
1000	10	50	1	100	2,25E+09	1	0,62	1,39E+07	1	0,13	2,99E+06	500	98,85	2,23E+09	4,45E+06	8,03E+04	98,19	2,19E+09	2,72E+04	3,98E+06	41,80	9,14E+08	230
1000	10	80	1	100	3,54E+09	1	0,39	1,39E+07	1	0,08	2,99E+06	800	99,17	3,51E+09	4,39E+06	1,29E+05	98,16	3,45E+09	2,67E+04	6,21E+06	41,39	1,43E+09	230
1000	100	20	1	100	2,35E+09	1	50,60	1,19E+09	1	11,97	2,81E+08	200	36,96	8,67E+08	4,34E+06	3,24E+04	98,13	8,51E+08	2,62E+04	1,50E+06	40,73	3,47E+08	231
1000	100	50	1	100	3,69E+09	1	32,20	1,19E+09	1	7,61	2,81E+08	500	59,79	2,20E+09	4,41E+06	8,07E+04	98,16	2,16E+09	2,68E+04	3,83E+06	40,83	8,84E+08	231
1000	100	80	1	100	5,04E+09	1	23,56	1,19E+09	1	5,57	2,81E+08	800	70,52	3,55E+09	4,44E+06	1,30E+05	98,18	3,49E+09	2,69E+04	6,18E+06	40,83	1,43E+09	231
1000	1000	20	1	100	9,20E+08	1	1,51	1,39E+07	1	0,32	2,99E+06	200	97,55	8,97E+08	4,49E+06	3,14E+04	98,20	8,81E+08	2,81E+04	1,49E+06	39,27	3,46E+08	232
1000	1000	50	1	100	2,20E+11	1	85,87	1,89E+11	1	13,11	2,89E+10	500	0,99	2,19E+09	4,38E+06	7,97E+04	98,14	2,15E+09	2,69E+04	3,78E+06	40,85	8,77E+08	232
1000	1000	80	1	100	2,22E+11	1	85,35	1,89E+11	1	13,03	2,89E+10	800	1,58	3,51E+09	4,38E+06	1,26E+05	98,15	3,44E+09	2,73E+04	6,08E+06	41,07	1,41E+09	232
10000	10	20	1	100	7,09E+10	1	0,02	1,39E+07	1	0,00	3,40E+06	2000	99,90	7,08E+10	3,54E+07	3,71E+05	99,30	7,03E+10	1,89E+05	1,17E+08	38,25	2,69E+10	230
10000	10	50	1	100	1,78E+11	1	0,01	1,39E+07	1	0,00	2,99E+06	5000	99,94	1,78E+11	3,56E+07	9,27E+05	99,30	1,77E+11	1,91E+05	2,93E+08	38,18	6,75E+10	230
10000	10	80	1	100	2,86E+11	1	0,00	1,39E+07	1	0,00	3,40E+06	8000	99,95	2,86E+11	3,57E+07	1,48E+06	99,30	2,84E+11	1,92E+05	4,72E+08	38,26	1,09E+11	230
10000	100	20	1	100	7,10E+10	1	1,67	1,19E+09	1	0,40	2,82E+08	2000	97,78	6,94E+10	3,47E+07	3,32E+05	99,29	6,89E+10	2,08E+05	1,16E+08	38,90	2,68E+10	231
10000	100	50	1	100	1,76E+11	1	0,68	1,19E+09	1	0,16	2,82E+08	5000	99,08	1,74E+11	3,48E+07	8,24E+05	99,29	1,73E+11	2,10E+05	2,91E+08	38,89	6,72E+10	231
10000	100	80	1	100	2,80E+11	1	0,42	1,19E+09	1	0,10	2,82E+08	8000	99,41	2,79E+11	3,48E+07	1,32E+06	96,06	2,68E+11	2,03E+05	4,69E+08	40,42	1,08E+11	231
10000	1000	20	1	100	2,95E+11	1	64,08	1,89E+11	1	12,71	3,75E+10	2000	22,82	6,74E+10	3,37E+07	3,27E+05	100,00	6,74E+10	2,06E+05	1,17E+08	40,27	2,71E+10	232
10000	1000	50	1	100	3,97E+11	1	47,62	1,89E+11	1	9,44	3,75E+10	5000	42,77	1,70E+11	3,40E+07	8,19E+05	99,27	1,69E+11	2,06E+05	2,93E+08	40,39	6,82E+10	232
10000	1000	80	1	100	4,99E+11	1	37,92	1,89E+11	1	7,52	3,75E+10	8000	54,42	2,72E+11	3,40E+07	1,31E+06	99,27	2,70E+11	2,06E+05	4,70E+08	40,45	1,09E+11	232

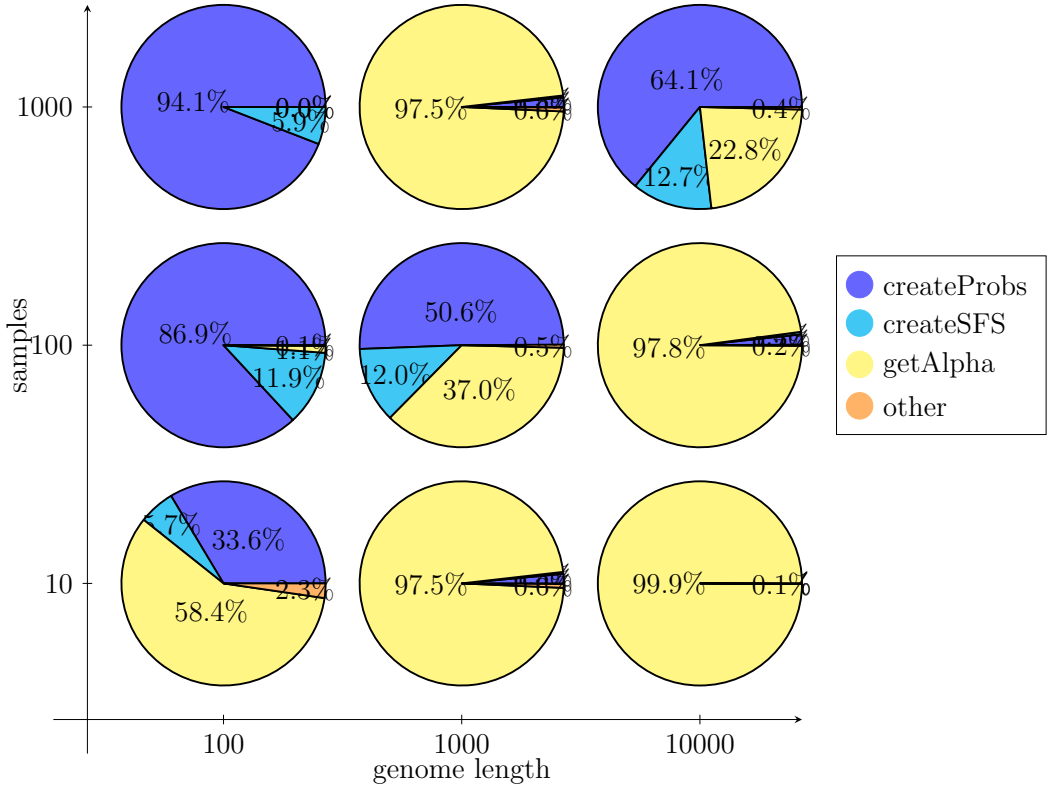


Figure 25: Clock cycle distributions with grid 20%

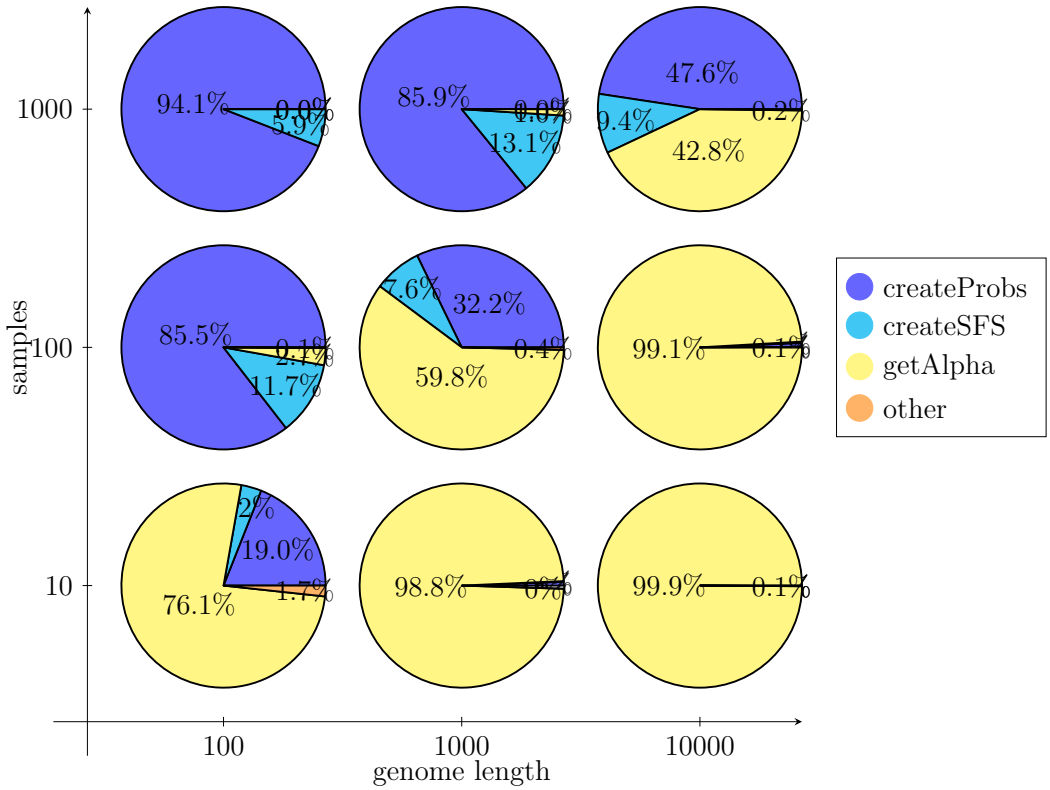


Figure 26: Clock cycle distributions with grid 50%

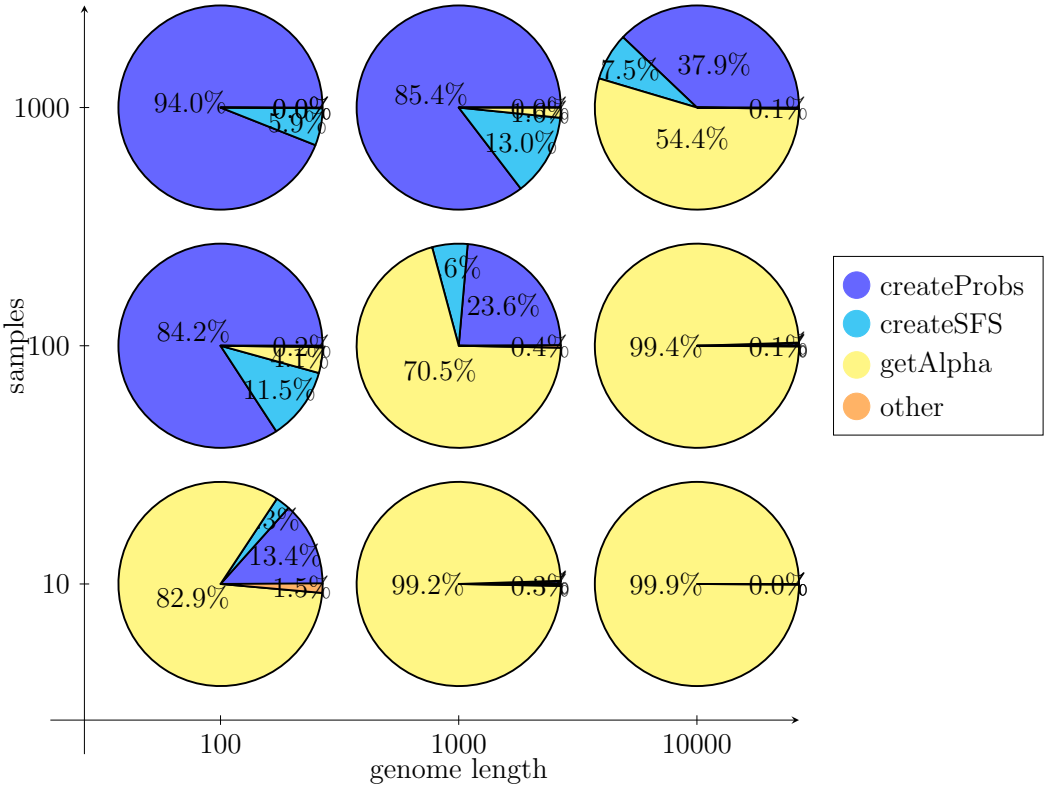


Figure 27: Clock cycle distributions with grid 80%

C Data dependencies of getAlpha

- getAlpha(sweepPosition)
  - val[] – Array with loop carried dependencies, size = 100, can be replaced by tracking some values
  - lik[] – Array with loop carried dependencies, size = 100, can be replaced by tracking some values.
  - SweepWidths – Array with loop carried dependencies, size = 100, can be replaced by tracking some values.
  - getMinMaxAlpha(sweepPosition, &minAlpha, &maxAlpha)
  - getClosestSNPIndex(sweepPosition)
  - getLikelihood(sweepPosition, alpha, &sweepWidth, startPos)
- getMinMaxAlpha(sweepPosition, *minAlpha, *maxAlpha)
  - segsites – Constant integer value
  - positionsInd[0] – First item of array positionsInd.
  - positionsInd[] – Array of size segsites consisting of constant integer values. All indexes of this array will be accessed in order.
- getClosestSNPIndex(sweepPosition)
  - positionsInd[] – Array of size segsites consisting of constant integer values. Access of this array in this function is rather predictable.
- getLikelihood(sweepPos, alpha, startPos)
  - segsites – Constant integer value
  - positionsInd[] – Array of size segsites consisting of constant integer values.[9]
  - xx –list containing frequencies x for all SNP's.[9]
  - nn – list of amount of samples for all SNP's.[9]
  - gridProbs – 3 dimensional array of size (max of xx - min of xx) * (max of nn - min of nn) * 300.
    Accesses patterns of this array are data dependent and thus hard to predict.
  - splint(x, n, ad)
  - baseLikelihood[] – Array of size segsties consisting of constant floating point values.[9]
- splint(x, n, ad)
  - gridADs[0] – first element of gridAd's array
  - gridSz – Constant integer value
  - logAD0 – Constant floating point value
  - interval – Constant floating point value
  - minn – Constant integer value

– startSFS – Constant integer value

– gridADs – Array size 300 floating point values, this could be calculated instead in 35 cycles, require 2 neighbouring values.

– gridProbs – 3 dimensional array of size (max of xx - min of xx) * (max of nn - min of nn) * 300.
  At least 2 accesses per function call, these accesses are neighbours in the third dimension.

[9]Access is based on startPos, amount of accesses is data dependent, but semi predictable due to the incrementing and decrementing indexes that are used.

D Record structure of PipeData

```
type Value = Signed 32
type Decimal = BitVector 32
type GenomeIndex = Signed 32
type SNPFreq = Signed 32
type AlphaIndex = Signed 32
data Direction = Incrementing | Decrementing deriving (Generic, NFDataX, Eq, Show, Bundle)
data LocalGridSize = Hundred | Five deriving (Generic, NFDataX, Eq, Show, Bundle)
data PipeData = Pipe {  sweepPos            :: Maybe GenomeIndex,       --global
                        minAlpha            :: Maybe Decimal,           --getAlpha
                        maxAlpha            :: Maybe Decimal,           --getAlpha
                        startPos            :: Maybe GenomeIndex,       --global
                        localGridSize       :: Maybe LocalGridSize,     --getAlpha
                        alphaIndex          :: Maybe AlphaIndex,        --getAlphaLoop
                        alphaInterval       :: Maybe Decimal,           --getAlpha
                        alpha               :: Maybe Decimal,           --getAlphaLoop
                        prevAlpha           :: Maybe Decimal,           --getAlpha
                        prevprevAlpha       :: Maybe Decimal,           --getAlpha
                        likelihoodIndex     :: Maybe GenomeIndex,       --getLikelihood
                        alphaDistance       :: Maybe Decimal,           --getLikelihood
                        direction           :: Maybe Direction,         --getLikelihood
                        totalLikelihood    :: Maybe Decimal,           --getLikelihood
                        highestLikelihood  :: Maybe Decimal,           --getAlpha
                        highestAlpha        :: Maybe Decimal,           --getAlpha
                        highestAlphaL       :: Maybe Decimal,           --getAlphaLoop
                        highestAlphaLL      :: Maybe Decimal,           --getAlphaLoop
                        highestAlphaR       :: Maybe Decimal,           --getAlphaLoop
                        captureAlphaR       :: Maybe Bool,              --getAlphaLoop
                        } deriving (Generic, NFDataX, Eq)
```