UNIVERSITY OF TWENTE.

**Faculty of Electrical Engineering,
Mathematics & Computer Science**

Accelerating selective sweep detection software with the GPU architecture

Reinout Corts

M.Sc. Thesis
20 January 2022

Supervisors:
dr. ir. M. E. T. Gerards
dr. ir. N. Alachiotis
dr. ir. A. B. J. Kokkeler

Computer Architecture for Embedded Systems
Faculty of Electrical Engineering,
Mathematics & Computer Science
University of Twente
P.O. Box 217
7500 AE Enschede
The Netherlands

Abstract

Selective sweep detection software processes sequenced genomic data to localize targets of recent and strong positive selection. These targets are found by analyzing Single-Nucleotide Polymorphisms (SNPs) in the genomic data which is stored as Multiple Sequence Alignments (MSAs). Due to advances in DNA sequencing, the amount of DNA data available is increasing rapidly. This causes Bioinformatics workloads to become more complex and especially more computational demanding. As multiple sequence alignment algorithms have been a big topic of research to cope with the surge of genomic data, Bioinformatics algorithms further up the processing pipeline, as selective sweep detection software, reveal high execution times for these large amounts of genomic data. This master thesis describes acceleration of a state-of-the-art selective sweep detection tool called OmegaPlus [1], [2], using the GPU architecture. The goal of this project is to boost performance of the selective sweep detection tool by utilizing the massively parallel architecture of the GPU and by providing a stepping stone for further research on this topic. The project focuses on implementing an optimized GPU kernel in which several GPU acceleration techniques are applied. The developed solution extends OmegaPlus with GPU-acceleration capabilities using the OpenCL General-Purpose GPU (GPGPU) framework [3]. OmegaPlus is based on Linkage Disequilibrium (LD), which is the non-random association of SNPs on different positions in the genomic data. The tool implements the ω-statistic that uses LD to accurately localize selective sweeps. Both the LD computation and the ω-statistic computation are compute intensive parts of the tool which together take up $>$95% of the total execution time. These compute intensive parts are targeted for GPU-acceleration. The LD computation in OmegaPlus is accelerated using an adaptation of an existing, highly optimized tool that utilizes Dense Linear Algebra (DLA) operations mapped on the GPU architecture. The ω-statistic computation is accelerated using a novel dynamic approach for different workloads. Two kernels have been developed for either a high or low ω-statistic workload. A performance evaluation using simulated datasets showed that the GPU-accelerated ω-statistic computation is up to 3.37x faster than the corresponding sequential implementation and the GPU-accelerated LD computation is up to 33.75x faster than the corresponding sequential implementation using the

same system for both versions. The complete GPU-accelerated OmegaPlus version, including both LD and ω-statistic computation, showed speedups up to 12.02x over the sequential OmegaPlus version using the same system for both versions. Speedups indicate a boost in performance but can be improved by applying additional acceleration techniques.

Contents

List of acronyms

CPU	Central Processing Unit
GPU	Graphics Processing Unit
FPGA	Field Programmable Gate Array
SNP	Single-Nucleotide Polymorphism
DNA	deoxyribonucleic acid
RNA	ribonucleic acid
LD	Linkage Disequilibrium
DLA	Dense Linear Algebra
CU	Compute Unit
SM	Streaming Multiprocessor
SP	Stream Processor
CUDA	Compute Unified Device Architecture
SC	SIMD Core
PB	Processing Block
TMRCA	time to the most recent common ancestor
RBD	Receptor-Binding Domain
HPC	High Performance Computing
PSA	Pairwise Sequence Alignment
MSA	Multiple Sequence Alignment
GATK	Genome Analysis Tool Kit

SFS Site Frequency Spectrum

ALU Arithmetic and Logic Unit

SIMT Single Instruction Multiple Threads

SIMD Single Instruction Multiple Data

GCN Graphics Core Next

RDNA Radeon DNA

GPGPU General-Purpose GPU

HLC High-Level Computing

PPF Phylogenetic Parsimony Function

PLF Phylogenetic Likelihood Function

PN Progressive Neighborhood

PLL Phylogenetic Likelihood Library

BLAS Basic Linear Algebra Subprograms

GEMM General Matrix Multiplication

BLIS BLAS-Like Instantiation Software

ISM Infinite Sites Model

FSM Finite Sites Model

GWAS Genome-Wide Association Studies

PE Processing Element

SW software

VCF Variant Calling Format

DP Dynamic Programming

Chapter 1

Introduction

Within health and life sciences processing of molecular data, e.g. deoxyribonucleic acid (DNA), to answer complex biological questions is an important topic. These questions can range from untangling the evolutionary history of organisms [4] to identifying clinically meaningful genes for cancer risk assessment, diagnosis, prognosis, and treatment [5]. The field of Bioinformatics plays an important role by providing researchers in this field with the necessary toolset in order to be able to answer these complex questions. Bioinformatics is a subdiscipline of biology and computer science where computer programs are used for the acquisition, storage and analysis of molecular data.

1.1 Motivation

The variation of molecular data, usually referring to genes, DNA, RNA and proteins, is acquired using direct and nowadays also indirect DNA sequencing technologies. Advances in DNA sequencing technologies in the past years, however, increased sequencing throughput while reducing sequencing costs, leading to an ever-increasing accumulation of whole genomes in private and public databases such as GenBank [6], GISAID [7], ENCODE [8], and The Cancer Genome Atlas [9]. As a result, studies today include a constantly growing number of organisms (sample size) in an effort to improve statistical power and thus accuracy and reliability of the outcome [10]. This has, inevitably, transformed Bioinformatics to a computational discipline that increasingly requires scalable algorithms and high-performance processing systems.

The COVID-19 pandemic brought Bioinformatics into the spotlight, exposing weaknesses in existing methods and tools with respect to handling large datasets efficiently. For instance, in April 2020, the authors of the widely used software MAFFT [11] for aligning multiple protein/DNA sequences released an experimental implementa-

tion only suitable for SARS-CoV-2. This implementation exploits certain characteristics of the viral genome (e.g., the high similarity between sequences due to the rapid spread of the virus) to approximate the alignment process in an effort to prevent prohibitively long processing times as the number of available SARS-CoV-2 genomes continue to increase rapidly. A phylogenetics study by Morel et al. [12] concluded that one of the two reasons why constructing the phylogenetic tree of 75,000 SARS-CoV-2 genomes is ”difficult” is the excessive computational requirements due to the large number of sequences (the other reason is the small number of mutations due to the rapid spread of the virus). The phylogenetic analysis was restricted to just 5% of the parameter space, possibly due to the heightened urgency to reach conclusions because of the pandemic.

Tools and algorithms that directly process the raw molecular data, e.g. the field of sequence alignment, have been an important topic of research to meet these higher computational demands. Performance advancements in these fields have shifted the computational load to algorithms further up the processing pipeline, thus requiring similar optimisations or acceleration to cope with the surge of data. A field further up the processing pipeline of Bioinformatics as a whole, is population genetics. Population genetics studies the genetic composition of populations as well as changes in this genetic composition. Algorithms in this field generally process aligned sequences to find these changes in the genes of a population. Selective sweep is a process within population genetics where a recent beneficial mutation has driven out all other variations of a gene in a population. This process is fueled by positive selection and subsequently leads to reduction in genetic variation near this mutation. Genetic variation in a DNA sequence is typically observed in the form of Single-Nucleotide Polymorphisms (SNPs), i.e., smallest variation of a gene on this sequence. So because of the sweep-induced reduction of genetic variations, a region on the chromosome will appear that exhibits a reduced number of SNPs when compared to the rest of the chromosome.

Detection of selective sweeps can be of importance in fighting diseases through understanding recent genetic changes in pathogenic bacteria and viruses. As previously stated, selective sweeps allow for rapid genetic changes in a population, Sá et al. [13] found that this process can play a key factor in the ability of pathogens to attack their hosts and survive medicines humans use to treat them. This can be seen as a competition between host and parasite about which organism can adapt its method of attack or defence, thus its genetics, the fastest. An individual being more resistant to a pathogen or itself being a more effective pathogen will have an advantage over its congeners, thus providing the fuel for a selective sweep. The human influenza virus, which has been involved in a genetic competition with humans for hundreds of years is an example for this phenomenon. Rambout et al. [14] found

that besides antigenic drift, which is considered the traditional model for changes in the viral genotype, selective sweeps have an effect as well. The time to the most recent common ancestor (TMRCA) of ”sister” strains in several flu populations showed, that within a few years they all evolved from a common ancestor indicating a selective sweep has acted on the population. Kang et al. [15] found signatures of a selective sweep in the SARS-CoV-2 virus, which acted in the spike protein Receptor-Binding Domain (RBD). Kang et al. suggest that this mutation likely contributed to SARS-CoV-2 emergence from animal reservoirs or enabled sustained human-to-human transmission. Due to advancements in DNA sequencing technologies and subsequently the computational advancements in the sequence alignment field, the need for more optimised selective sweep detection software is of great importance to keep execution times feasible.

Over the years, the computing landscape has become highly heterogeneous, with modern platforms currently integrating multi-core CPUs with GPUs and/or Field Programmable Gate Arrays (FPGAs). Modern GPUs exhibit massively parallel computer architectures with thousands of processing cores. FPGAs, on the other hand, do not have a fixed processing architecture but can implement any specialized computer architecture by configuring and interconnecting millions of fine-grained logic elements. The exploitation of hardware acceleration and high-performance computing solutions is an important factor in coping with the ever-increasing accumulation of whole genomes. Binder et al. [16] observe that GPUs hold great performance for SNP comparison computations, which is generally a computationally intensive part of selective sweep detection software. In a work by Alachiotis et al. [17], the observation is made that other parts of the detection software can be cast in terms of a series of Dense Linear Algebra (DLA) operations, which are well-studied by the High Performance Computing (HPC) community. It can therefore be concluded that GPUs hold great potential in accelerating selective sweep detection to cope with the surge of molecular data. Advantages of using a GPU as accelerator are, relatively low implementation times and high portability over multiple architectures when using a framework. Moreover, due to the big commercial market GPU performances have grown significantly when compared with CPUs and FPGAs. Since many personal computers and laptops already feature a reasonable powerful GPU nowadays, researchers can easily make use of this computing power.

1.2 Research questions

As previously stated, the use of a GPU to accelerate selective sweep detection software has some great advantages. The general goal of this research will be the acceleration of a state-of-the-art selective sweep detection tool using the GPU

architecture. The main research question to answer for this project is:

- How much performance gain can be achieved when accelerating a state-of-the-art selective sweep detection tool using GPU architectures?

In order to achieve the best results, with respect to the main research question, the following sub-question is set up to target the most time consuming parts:

- Which parts of the state-of-the-art tool are the most computationally intensive to target for acceleration?

After pinpointing the most intensive parts of the tool the following sub-question is set up to research the best design choices in order to achieve the main goal:

- How can the computationally intensive parts be mapped optimally on the computational units and the different types of memory of the GPU?

The last sub-question is set up for design validation and performance evaluation under different input parameters:

- How do performance values scale for datasets with different number of genomes and datasets with different number of SNPs per genome?

These sub-questions are formulated in order to be able to answer the main research question of this project to the best extend. Through analysis of the tool its algorithm the most computationally intensive parts can be found. Accelerating these parts will result in the highest performance gain as they are the most time consuming. In order to maximise the performance gain an optimal mapping of the intensive parts to the computational units and memory of the GPU needs to be found. To specifically answer the measurable part of the main research question, the last sub-question is formulated. By performing a thorough performance evaluation with a variety of datasets the resulting overall performance gain can be specified.

1.3 Scientific contribution

As mentioned earlier in this chapter, the advances in DNA sequencing technologies have transformed Bioinformatics to a computational discipline that increasingly requires scalable algorithms and high-performance processing systems. Furthermore, the COVID-19 pandemic brought Bioinformatics into the spotlight, exposing weaknesses in existing methods and tools with respect to handling large datasets efficiently. Selective sweep detection specifically, is proven to play a role in pathogens evolution and thus can be of importance in fighting these pathogens. The time span

for developing medicines or vaccines to counter diseases greatly depends on the time needed for analysing the genetic data and subsequently processing this data. Researchers thus benefit from high-performance tools and algorithms in order to reduce development time and have a bigger impact on fighting diseases.

Within this research the computational possibilities of applying GPU computing power in selective sweep detection software are explored, with the aim of reducing execution times of a state-of-the-art tool. Lower execution times can contribute in making analysis work by biologists more convenient and subsequently increase its impact in fighting diseases for example. Furthermore this research can act as a building block for future research to continue on, or as a building block for applying GPU computing power in other related tools and algorithms that heavily rely on SNP comparisons.

1.4 Outline

The outline of this work is as follows.

Chapter 2 gives background information about population genetics, selective sweeps and corresponding methods and tools for selective sweep detection in section 2.1 and information about the GPU architecture, programming model and acceleration techniques in section 2.2.

Chapter 3 gives an insight into related work regarding other hardware acceleration efforts within Bioinformatics and particularly in fields related to selective sweep detection. This literature research was conducted to gather initial knowledge and global insight in the topic of optimisation and acceleration of Bioinformatics tools/algorithms. Tools and algorithms closely related to population genetics, and in particular positive selection, are topic of this research. Attention will be paid to the initial problem of the tool (computationally/memory), the platform used for optimisation and/or acceleration (CPU, GPU, FPGA), the methods and techniques used to tackle this initial problem and a performance evaluation. Section 3.1 and section 3.2 focus on phylogenetics and population genetics tools and algorithms respectively. Section 3.3 will elaborate on selective sweep detection software, where multiple tools with their respective methods are discussed. Section 3.4 concludes the chapter with an performance overview of different platforms and a definitive selective sweep detection tool to accelerate.

Chapter 4 will elaborate on the chosen selective sweep detection tool and on an already existing GPU-accelerated tool for computing LD. Section 4.1 and section 4.2 will thoroughly explain these respective implementations. The chapter is concluded with section 4.3, in which the computationally intensive parts of the chosen tool are found through profiling.

Chapter 5 will talk about the adaptation of the existing GPU-accelerated LD tool and the developed design for GPU-acceleration of the chosen selective sweep detection tool. An introduction will be presented in section 5.1, the adaptation of the GPU-accelerated LD tool will be explained in section 5.2, section 5.3 will elaborate on the developed GPU kernels and the host processor code will be explained in section 5.4. The chapter is concluded with an overview of the complete design in section 5.5.

At the end of this work a performance evaluation is presented in chapter 6, with section 6.1 explaining the experimental setup to get an insight in the scaling of the solution's performance and section 6.2 will present a kernel verification with respect to correctness and resulting errors. The chapter is concluded with a performance comparison between the original tool and the GPU-accelerated version in section 6.3, which includes discussion of these results.

This work as a whole is concluded in chapter 7 with a conclusion and future work recommendation.

Chapter 2

Background

This chapter will elaborate on important background information for this project. An introduction to population genetics will be given in order to explain the foundation of selective sweep theory. Signatures of selective sweeps will be discussed in this chapter, and respective selective sweep detection tools that exploit these signatures through various methods. Furthermore some background information will be given about the GPU architecture, where programming frameworks and acceleration techniques for this specific architecture will be discussed.

2.1 Population genetics

Population genetics studies the genetic composition of populations as well as changes in this genetic composition. The DNA determines the genetic composition and is build up of genes, where these genes are a region on the DNA that have its own function. A set of genes, or a single gene make up a genotype which determines, together with environmental and development factors, an organisms' phenotype. A phenotype is a certain observable trait an organism has, e.g., eye colour but also a persons' blood group. Genes can have different variants, hence the multiple phenotypes, these variants are determined through alleles. An allele is a variant of the same gene at the same position on the chromosome in a population, which can be one, two or more variants. The genetic composition can also be described as the frequency of alleles in the DNA of populations. The position of a gene or allele on the chromosome is called a locus, or loci in plural. Alleles can differ in size, from thousands of base-pairs to only a single base-pair which is called a single-nucleotide polymorphism (SNP). A base-pair consists of two nucleotide bases and within DNA four different nucleotide bases exist, i.e., guanine (G), adenine (A), cytosine (C) and thymine (T). The nucleotide bases form pairs in the DNA, where Adenine always pairs with thymine (A-T) and cytosine always pairs with guanine (C-G), these pairs

can be stacked upon each other to form a long-chain helical DNA structure. A graphical representation of the DNA is shown in fig. 2.1.

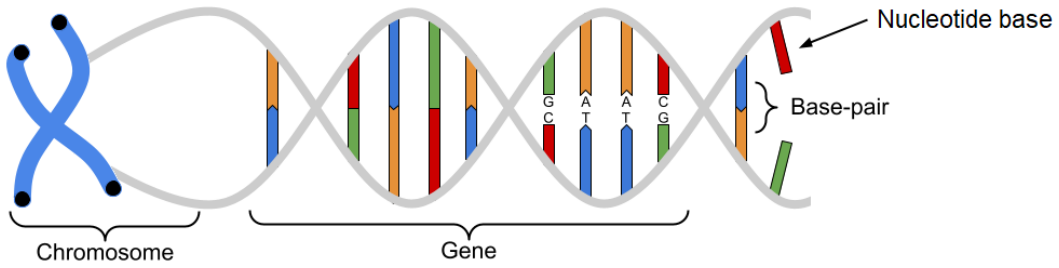


Figure 2.1: Graphical representation of the DNA, zoomed in on a chromosome. An example of a gene formed of multiple base-pairs is showed, with nucleotide bases G, A, C and T. The pairs together form the long-chain helical DNA structure found in chromosomes and cells.

Typically genetic variation is observed in the form of SNPs and this variation is driven through four main evolutionary processes, i.e., natural selection, genetic drift, gene flow and repeating mutation.

Natural selection occurs when a phenotype is preferred for survival and/or reproduction over an other phenotype [18]. Over time this specific phenotype will occur more in the population due to its advantages. Positive selection is a form of natural selection where an advantageous allele increases in frequency and eventually becomes fixed reducing genetic variation. This phenomenon was first discovered by Darwin and Wallace [19]. Figure 2.2 shows an illustration of natural selection where a light coloured butterfly is advantageous over a darker coloured butterfly, which results in it occurring more over time.

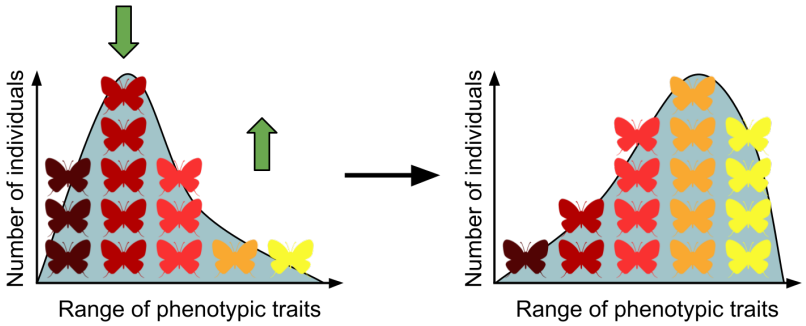


Figure 2.2: Illustration of natural selection over time. At a certain point in time the phenotype of more light coloured wings becomes the preferred one for survival/reproduction (left). Over time this will result in this phenotype to occur more in the population, a directional shift in allele frequency (right) (adapted from [20]).

A mutation is an alternation in the DNA or RNA sequence which can be caused by errors during replication, exposure to ionizing radiation, exposure to chemicals, or infection by viruses [18]. When mutations occur in the eggs or sperm these can

be passed on to offspring and are called germ line mutations, otherwise they are called somatic mutations. Other evolutionary processes as natural selection heavily depend on mutations, as it is the foundation of genetic variation. Figure 2.3 shows a single base-pair mutation after DNA replication.

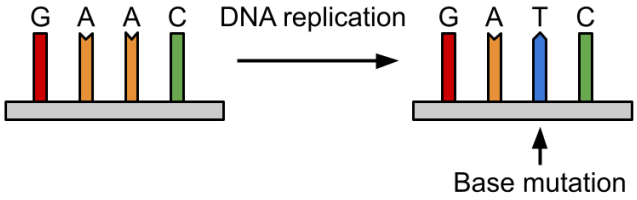


Figure 2.3: Illustration of a base mutation in the DNA (change of one base-pair to another), where the mutation is permanent after replication of this part of the DNA (adapted from [21]).

Genetic drift is the change in the frequency of an existing allele from generation to generation due to chance events, i.e., random fluctuations [18], [22]. Genetic drift can initiate a reduction in genetic variation by causing gene variants to disappear from a population or make certain traits more dominant and appear more frequent. Effects of genetic drift are more notable in smaller populations due to the fact that a variation of a gene faces a greater chance of being lost when it occurs infrequently [18]. Figure 2.4 shows an example of a chance event reducing genetic variation, i.e., genetic drift.

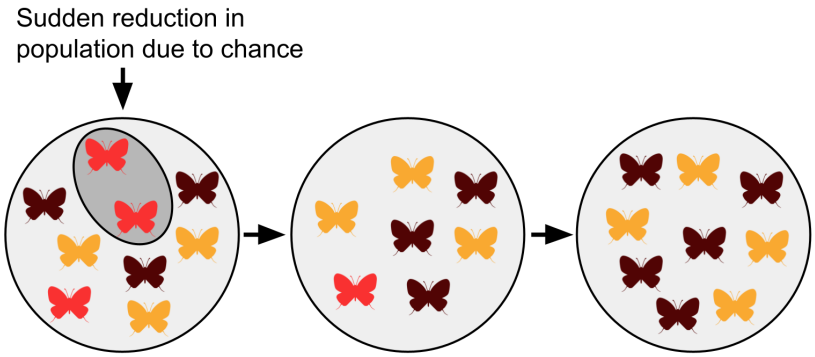


Figure 2.4: Illustration of genetic drift over multiple generations. At a certain point in time the population experienced a sudden reduction due to a chance event (left). Due to the small size of the population, over time this chance event caused a reduction in genetic variation (red wing phenotype disappeared) (adapted from [23])

Gene flow is a collective term that explains every way of gene transfer from one population to another [24]. Gene flow rates are high within a single effective population, when these rates decrease under a certain level, a population can split into two. Gene flow largely determines the independence of genetic changes in local populations within a single species. Figure 2.5 shows an illustration of gene flow through a

geographical barrier resulting in an increase in genetic variation in population A.

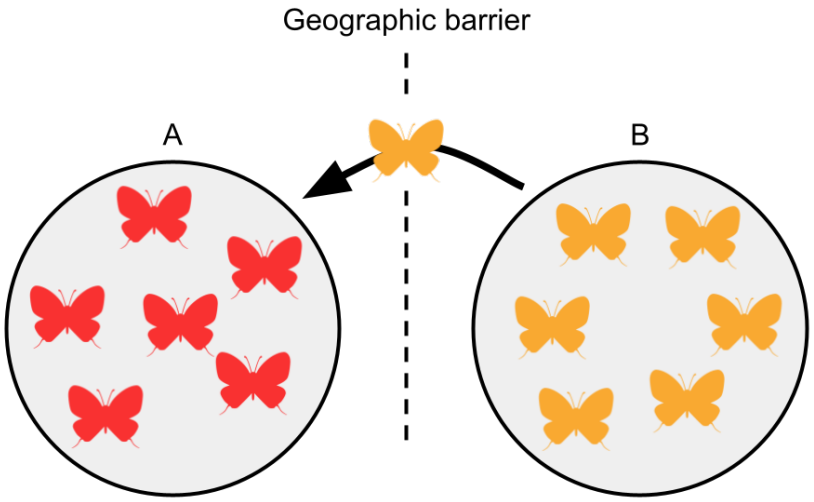


Figure 2.5: Illustration of gene flow from population B to A through a geographic barrier (mountain/river etc.), where population A only consists of individuals with the red coloured wing phenotype and B only with the yellow coloured wing phenotype. Due to the movement of a individual from B to A the genetic variation in population A increases (adapted from [25]).

An excellent book that further elaborates on the evolutionary processes and population genetics as a whole is ”Essentials of Genetics” by Klug et al. [18].

2.1.1 Data collection and preparation

A big part of Bioinformatics and population genetics as a whole is focused on the analysis of molecular data, which generally consists of DNA, RNA or proteins. This section will elaborate on the collection of molecular data through DNA sequencing and the preparation of the data through sequence alignment and pinpointing SNPs. The resulting aligned sequences with the acquired SNPs are the input data for selective sweep detection software.

DNA sequencing is the process of determining the order of four nucleotide bases that make up the DNA molecule [26]. The ability of these nucleotide bases to form pairs is the basis for DNA replication, needed for cell division, and the pairing underlies the basic principle DNA sequencing methods are built on. DNA sequencing is also used to indirectly sequence other types of molecular data, e.g., RNA or proteins. In RNA thymine is replaced with the nucleotide base uracil (U), which also pairs with adenine. Protein sequences consist of a larger number of characters as these are made up of amino acids instead of the 5 nucleotide bases.

In order to find genetic variation within a population, DNA sequences are compared to each other to find existing alleles and subsequently SNPs. These comparisons are done by aligning sequences based on the same loci. Sequence alignment

is performed to identify regions of similarity that may be a result of functional, structural, or evolutionary relationships between sequences. There are three types of sequence alignment, i.e., one-to-one, one-to-many or many-to-many comparisons, all with the goal to find the optimal alignment distance, i.e., the highest similarity between sequences. One-to-one is also called Pairwise Sequence Alignment (PSA), one-to-many and many-to-many comparisons are also called Multiple Sequence Alignment (MSA), in which multiple sequences are analysed and sub-groups of these sequences are aligned. The Smith-Waterman (S-W) [27] algorithm is a frequently used PSA algorithm. NCBI BLAST [28] and ClustalW [29] are tools for performing MSA. Typically the sequences are represented as rows in a 2D matrix where the nucleotide base characters are the entries in this matrix. Sequence alignment can be performed with gaps between the nucleotide bases or gapless, where a gap is an insertion of a dummy character in order to have a higher similarity between characters in successive columns. Gap insertion however yields a gap penalty which reduces the alignment distance score. Figure 2.6 shows the result of a multiple sequence alignment where sequences are grouped based on their distance score.

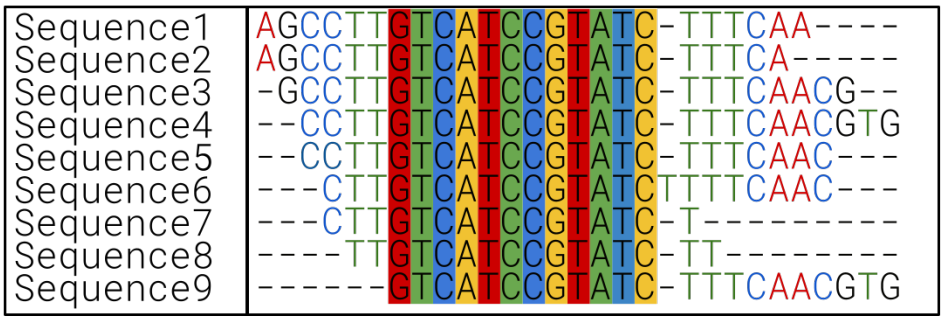


Figure 2.6: Result of a MSA with 9 DNA sequences where the sequences are ordered based on the alignment score.

After two or more sequences have been aligned, possible SNPs can be found in the range of these sequences. As previously stated a SNP is a single base-pair variation of a gene, the smallest sized allele. SNP calling is the process of pinpointing SNPs, which is based on single locus variations in aligned sequences. It is however, much easier to correctly call an SNP using a reference sequence that represents the neutral genetic composition of an organism [30]. A number of methods for SNP calling have been developed which have been implemented in multiple tools, e.g., Genome Analysis Tool Kit (GATK) [31], Samtools/mpileup [32], FreeBayes [33], Platypus [34], SNVer [35], VarDict [36], and VarScan [37].

Figure 2.7 shows a SNP at a single locus of multiple aligned sequences. Two individuals together can pass on one of the two possible alleles or SNPs to their offspring.

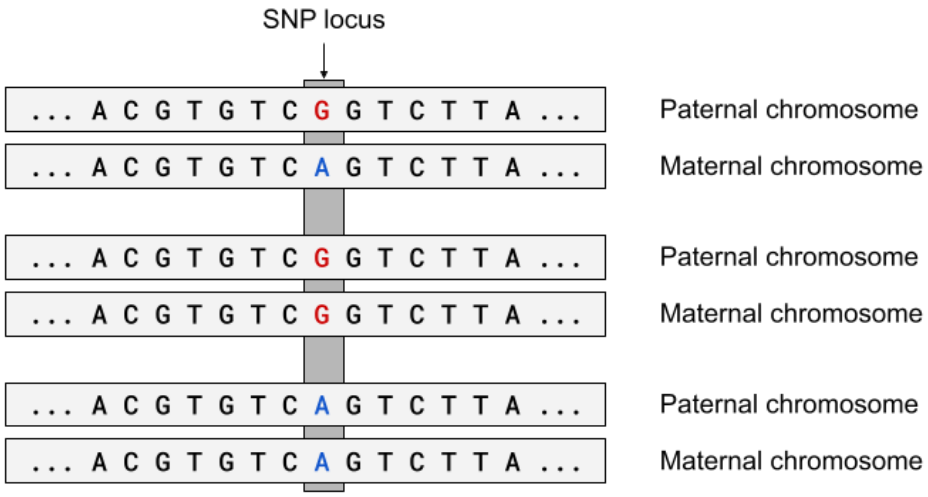


Figure 2.7: Illustration of a SNP at a locus of six aligned sequences of individuals, where G and A are the two possible alleles for this position (adapted from [38]).

2.1.2 Natural selection and selective sweeps

Natural selection is one of the four main evolutionary processes that cause genetic variation. It occurs when one phenotype is preferred over an other, resulting in an increase of frequency of that phenotype. Positive selection is a form of natural selection where an allele is favoured over other alleles and its frequency will shift in the direction of that phenotype and eventually sweep the population, i.e., completely omitting the non favoured alleles. When this process happens fast the frequencies of alleles at closely linked loci, loci in the same chromosome, will increase due to the so-called 'hitch-hiking effect' [39]. This process is called a selective sweep. Because of the sweep-induced reduction of genetic variations, a subgenomic region will appear that exhibits a reduced number of SNPs when compared to the rest of the chromosome.

When a single recent strongly beneficial mutation has occurred in a population, a selective sweep can occur on this rare allele. The frequency of this allele will increase rapidly due to natural/positive selection which can trigger a selective sweep through genetic hitch-hiking. Typically this is called a classic or hard selective sweep [39], [40]. A soft selective sweep occurs when beneficial mutations appear in multiple individuals in a population, which results in a lower reduction of genetic variation and has a less pronounced hitch-hiking effect [41]–[43]. Figure 2.8 illustrates the process of both a soft and hard selective sweep. The remainder of this work will focus on hard selective sweeps.

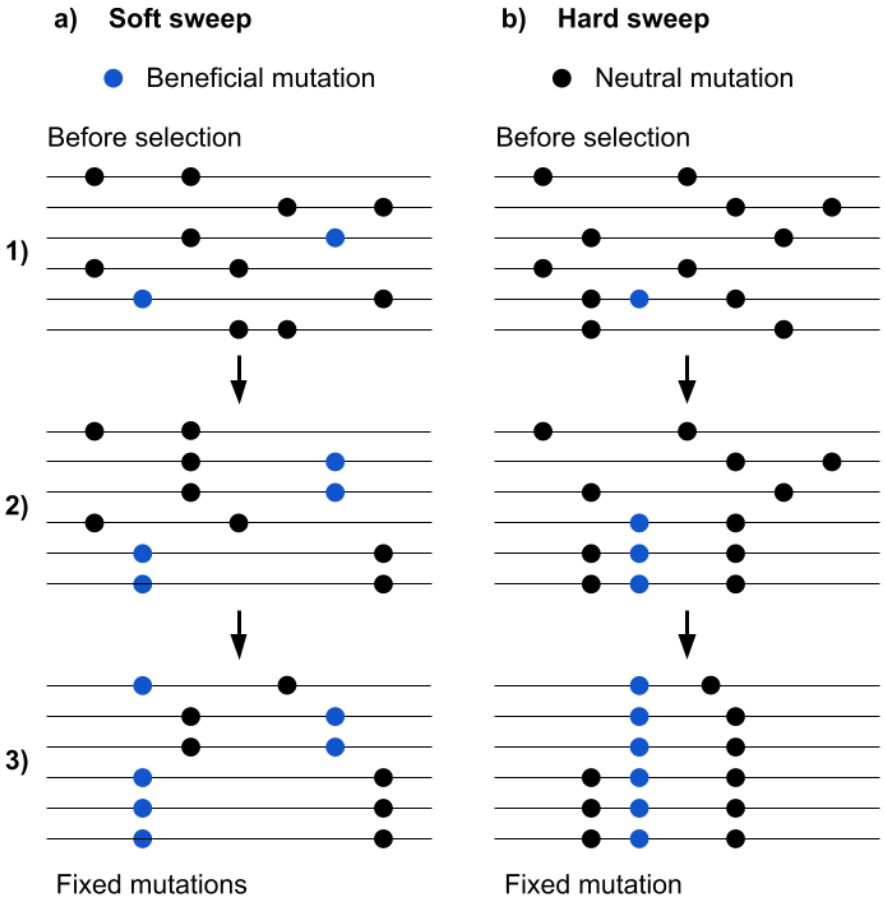


Figure 2.8: Illustration of a) soft selective sweep and b) hard selective sweep. 1) Beneficial mutation(s) has/have occurred. 2) The frequency of the beneficial mutation(s) increase(s) in the population 3) All individuals have either of the beneficial mutations (soft sweep) or the same mutation (hard sweep). The regions on the left and the right side of the selection site comprise pairs of SNPs with high LD values. Pairs of SNPs on different sides have low LD

2.1.3 Signatures of selective sweeps

A selective sweep can be characterised by three distinct signatures in genomes. The first signature is, as previously stated, a reduced number of SNPs that can be observed in a subgenomic region [39]. The second signature is a directional shift in the Site Frequency Spectrum (SFS), i.e., the distribution of allele frequencies in DNA sequences, toward low- and high-frequency derived variants [44]. The last signature that can be observed is a localised pattern of Linkage Disequilibrium (LD) levels. LD is the non-random association of alleles at different loci in a population, and appears when genotypes at two loci are dependent. This dependency is present when the frequency of association of their different alleles is higher or lower. Kim and Nielsen [45] found that increased levels of LD are observed on neighbouring regions on both sides of a beneficial mutation, whereas the level of LD between loci

that are located on different sides of the beneficial allele remains low.

The most commonly used measure of LD relies on Pearson's correlation coefficient, r^2, as provided for the calculation of LD between SNPs A and B in Equation 2.1:

$$r^2_{AB} = \frac{D^2_{AB}}{p_A(1-p_A)p_B(1-p_B)}, \tag{2.1}$$

where p_A and p_B are the frequencies of the derived allele at SNPs A and B, respectively, while D_{AB} is the coefficient of LD, defined as follows:

$$D_{AB} = p_{AB} - p_A p_B, \tag{2.2}$$

where p_{AB} is the frequency of occurrence of the derived allele at both SNPs A and B. Whenever $D_{AB} \neq 0$, A and B are said to be in linkage disequilibrium, otherwise A and B are in linkage equilibrium, i.e., SNPs A and B occur independently of each other.

Figure 2.9 gives an illustration of LD in two different populations with both two SNPs on two loci. The figure shows SNPs A, a for the colour of the shape and SNPs B, b for the shape itself, where the LD is only calculated for the association of A and B. In the situation with two loci consisting both of two SNPs, also called biallelic, the restrictions are so strong that one measure of LD represents all the other LD relationships $D_{AB} = -D_{Ab} = -D_{aB} = D_{ab}$ with:

$$\begin{aligned} D_{AB} &= p_{AB} - p_A p_B, \\ -D_{Ab} &= p_{Ab} - p_A p_b, \\ -D_{aB} &= p_{aB} - p_a p_B, \\ D_{ab} &= p_{ab} - p_a p_b. \end{aligned} \tag{2.3}$$

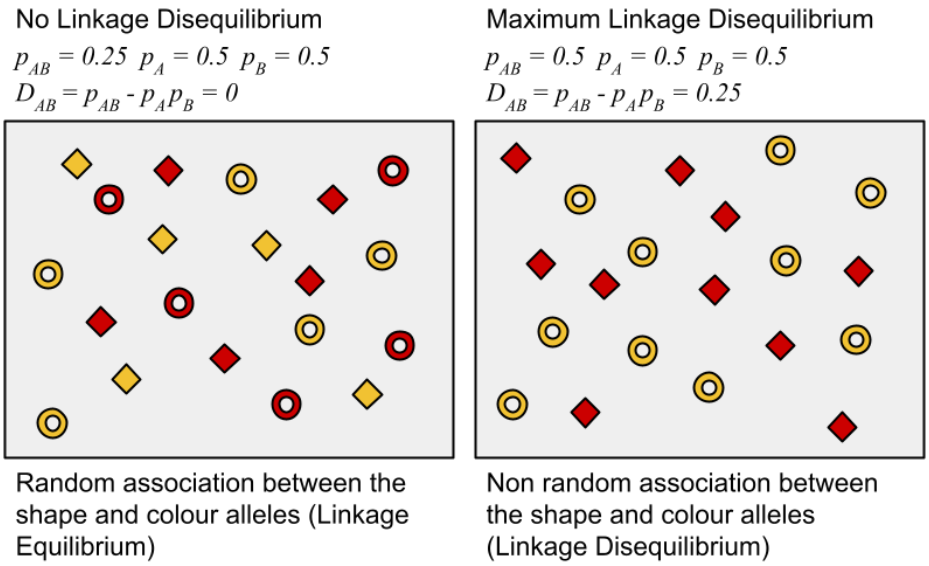


Figure 2.9: Illustration of LD in two different populations, where in the left population the association between the shape and colour allele is random (no LD) and in the right population the association is non-random, every circle is yellow and every diamond is red (maximum LD) (adapted from [46]).

The significance of LD, as can been observed from titles of population genetics papers declaring "Population genomics: Linkage disequilibrium holds the key" [47], has motivated various studies for high-performance LD computation.

Kim and Nielsen developed the ω-statistic to detect the genomic pattern LD exhibits.

Every ω is computed at a specific location in a genomic region with a total of W SNPs. This genomic region is split up into a left L and right R subgenomic region which consist of l and $W - l$ SNPs respectively. Then ω at a location in the region is computed as follows:

$$\omega = \frac{\left(\binom{l}{2} + \binom{W-l}{2}\right)^{-1}\left(\sum_{i,j\in L} r_{ij}^2 + \sum_{i,j\in R} r_{ij}^2\right)}{\left(l(W-l)\right)^{-1}\sum_{i\in L, j\in R} r_{ij}^2}. \tag{2.4}$$

High ω-statistic values indicate regions for positive selection since the average LD is high within the L and R subgenomic regions but not across the beneficial mutation.

2.2 Graphics processing unit

This section will give background information about the general architecture of a GPU, an explanation about programming a GPU through so-called GPU programming frameworks and an introduction about the main acceleration techniques for the GPU architecture.

2.2.1 GPU architecture

There are two main manufacturers that produce GPUs used for high performance computing, i.e., AMD and Nvidia, both with their own specific architectures. The general architecture however applies to both of these GPUs, where the naming convention differs slightly. In this section names used by AMD and Nvidia will both be given respectively, a single name applies to both manufacturers. The GPU architecture has evolved through the years, this model GPU architecture applies to GPUs of the past 10 years.

Where a CPU can consist of 2 to even 32 cores per chip, a GPU typically consists of more than 2000 cores nowadays which are grouped in N_{CU} Compute Units (CUs) or Streaming Multiprocessors (SMs). Each CU/SM is built up with N_{SP} Stream Processors (SPs) or CUDA cores, with N_{SP} typically being 64 or a multiple of 64. The SPs or CUDA cores are effectively the Arithmetic and Logic Units (ALUs) of the GPU which execute instructions of scheduled threads. Within a CU/SM, N_G number of SPs/CUDA cores are grouped in N_{SC} SIMD Cores (SCs) or Processing Blocks (PBs), where each group of SPs/CUDA cores can execute threads independently. The total number of ALUs in a GPU is then equal to: $N_{CU} * (N_G * N_{SC})$.

The massive parallelism of a GPU comes from the fact that a multiple number of threads W_s, together called a wavefront or warp, execute the same instruction on different data from the memory, called the Single Instruction Multiple Threads (SIMT) exectution model. During kernel execution on a GPU, N_W number of wavefronts/warps are hosted on the GPU for executing instructions, where a wavefront/warp executes independently on a SC/PB. On older AMD architectures (GCN) a wavefront consisted of 64 threads, on newer architectures (RDNA, RDNA 2) this number is reduced to 32. In Nvidia architectures 32 threads form a warp, the equivalent of a wavefront. The size of a wavefront or warp is fixed, so in order to fully utilise the computing power of a GPU, all threads in a wavefront or warp should have a data element to execute an instruction on.

In order to operate, a GPU needs a host, which is typically a CPU. The host transfers data from its own memory to the global memory of the GPU and can instruct the GPU to execute a kernel, which is a pre-compiled routine that calls the instructions. Within a GPU architecture there are 5 main types of memory and additional caches. From slowest to fastest memory, the 5 are organised as follows:

- Global and local memory
- Texture memory
- Constant memory
- Shared memory
- Registers

Global memory is off-chip memory in the order of gigabytes nowadays. It allows for read and write access and it is allocated by the host. All threads can access global memory but accesses are slow due to it being off-chip. Local memory is an abstraction of global memory used when the compiler determines there is not enough space for variables in the registers. The same rules of global memory apply to local memory. Texture memory is a L2 cached read-only off-chip memory similar to global memory. It is allocated by host on the device. All threads can access texture memory. Constant memory is also a L2 cached read-only off-chip memory. Constant memory is allocated by host on device. It is in the order of tens of kilobytes. Constant memory can be accessed by all threads and is fast as long accesses to the memory are executed in parallel. Constant memory is managed by the compiler. Shared memory is on-chip memory shared by all threads in a CU or SM and is also in the order of tens of kilobytes. Shared memory is explicitly managed by the programmer but is allocated by the GPU itself. Registers are assigned to each thread on-chip. Besides that each CU or SM has its own register file. Register allocation however, is not managed by the programmer but managed by the compiler.

As with a CPU, memory accesses on a GPU typically go through cache. Memory transfers between off and on-chip memory are all cached by L2 cache, which is connected to off-chip memory via several memory controllers. L2 cache is on-chip and can be accessed by all CUs or SMs. Within every CU or SM, L1 cache is used for reading and writing L2 cache. Shared memory and registers are able to read from and write to global memory through L2 and L1 cache. Every cache has a certain line size which is the chunk of bytes a cache line transfers in one iteration determined by the hardware. Line sizes are typically 32, 64 or 128 bytes. Whenever a read or write is issued the cache line transfers a complete chunk of aligned memory, even though less data elements are needed. Whenever data accesses are misaligned and elements are spaced more than the line size, multiple transfers are needed.

Cache hierarchy can differ between current GPU architectures but the concept of reading and writing is generally the same.

A graphical representation of a general GPU architecture can be found in fig. 2.10, where the AMD naming convention is used. A detailed view on a GPU architecture can be found in the work by Cheramangalath et al. [48].

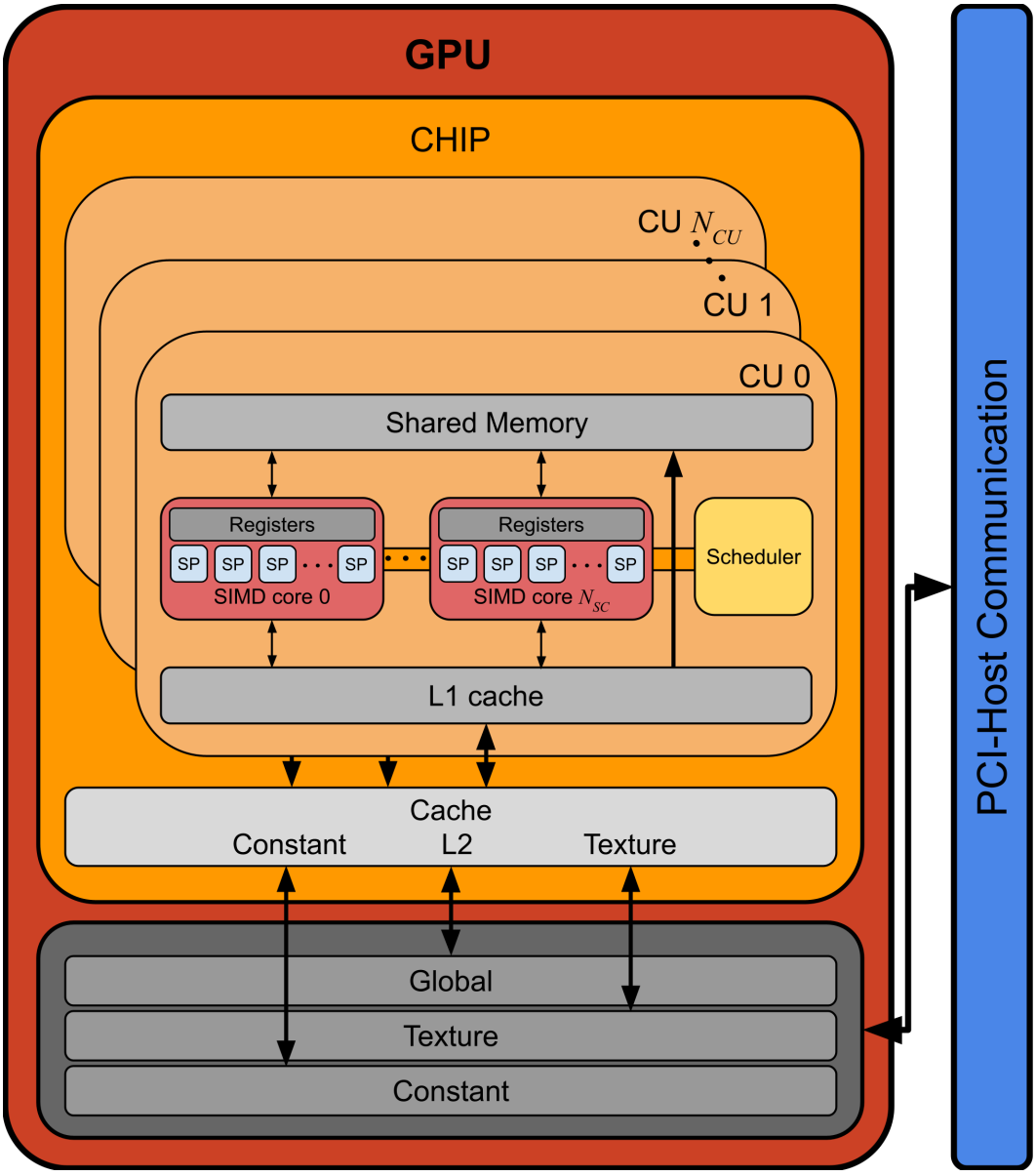


Figure 2.10: General GPU architecture using the AMD naming convention. The figure shows the placement of the different memories/registers, caches, Compute Units, SIMD cores and Stream Processors on the GPU and the communication with the host processor.

2.2.2 GPU frameworks

A GPU can be used and programmed via certain GPGPU frameworks. These frameworks make it possible to transfer data between the GPU and CPU memory, compile, develop and run kernels on the GPU and can be used to request information of the device. Well known frameworks are OpenCL [3] and CUDA [49], where OpenCL is vendor independent and can also be used on different architectures (GPU, CPU, FPGA), and CUDA being specifically for Nvidia CUDA supported GPU's. Both of these frameworks are C-based and mainly used in GPU High-Level

Computing (HLC). For this project it was decided to use OpenCL due to its flexibility with regards to multiple vendors, which can benefit researchers. The following information will be applicable to the OpenCL framework.

After transferring data to the global memory of the GPU, pre-compiled kernels can be called that perform computations on this data through various instructions. The kernels can be developed using a C-based language dialect and are very similar to regular functions annotated with `__kernel`. The function parameters can either be pass by reference (pointer) or pass by value, where a reference is a location on the GPU its global memory and a value can be a single value used in the kernel. Pointers are annotated with the `__global`, `__constant`, `__local` and `__private` qualifiers reflecting the memory hierarchy, where `__local` variables are stored in shared memory and `__private` variables in the registers. Default variables declared within the kernel are stored in the thread registers and are `__private`.

In order to set pointer arguments to an allocated space on the memory or to specify a pass by value argument, the `clSetKernelArg` function can be used. Whenever a global memory pointer argument is used, i.e., `__global` and `__constant`, a space on the memory should be allocated with `clCreateBuffer`. After allocation, this space can be written and read with `clEnqueueWriteBuffer` and `clEnqueueReadBuffer` respectively. The kernel itself can be executed with the `clEnqueueNDRangeKernel` function, which has three important arguments that specify the kernel name, the number of threads and the thread group size. The number of threads passed to the function determines how many threads or work-items will be deployed to execute the kernel on and is called the `global` size, G_s. The number of wavefronts/warps, N_W, executing on the GPU is then equal to G_s/W_s, with W_s being the number of threads in a wavefront/warp. The thread group size specifies how many work-items together are scheduled on a CU or SM and is called the `local` or work-group size, L_s, which should be a multiple of W_s. The `global` size should be a multiple of the `local` size in order to create an integer number, $N_{WG} = G_s/L_s$, of so-called work-groups. All `Enqueue` functions expect an OpenCL command queue as argument in which the issued event can be placed for execution. The issued event in the queue can also be added to an OpenCL event wait list which can be used for waiting on certain events to finish.

In listing 2.1 an example kernel is shown where array `x` of length G_s is multiplied with a scalar `y` and stored in array `z`. `x` is stored in constant memory since its values are only read, where `z` is read/write since it is in global memory. The `get` functions return the current `global` work-item id, the current `local` work-item id, the current work-group id and the specified `local` size, L_s, respectively. Hence the result of the `get_global_id` function can be rewritten as the return values of the other functions

$$get\_global\_id = wid * lsz + lid \tag{2.5}$$

These functions can be used to read specific data elements per work-item and perform many calculations in parallel.

```
__kernel void vecscal(__constant uint *x,
                      uint y, __global uint *z)
{
    uint gid = get_global_id(0);            // Global id
    uint lid = get_local_id(0);             // Local id
    uint wid = get_group_id(0);             // Group id
    uint lsz = get_local_size(0);           // Local size

    uint scalar = x[wid * lsz + lid] * y;
    z[gid] = scalar;
}
```

Source Code 2.1: Kernel example

2.2.3 Acceleration techniques

The performance of a GPU kernel is typically limited by memory bandwidth. Calculations are always performed on data that is initially transferred to the GPU's global memory, which is the slowest memory on the GPU. Thus, it is of great importance to optimise data transfers between global memory and local memories, i.e., shared memory and registers.

Minimize data transfer A very simple but efficient acceleration technique is to minimise data transfer. For example, instead of adding a certain value to an element on global memory multiple times, it is good practice to use a temporary register variable that accumulates the values and write it to global memory once.

Coalesced memory access An global memory access pattern that results in high transfer rates is called a coalesced memory access. A coalesced memory access refers to combining accesses to multiple data elements into a single memory transfer. As previously stated the cache line size determines the number of bytes that are transferred in one iteration. Given a line size of 64 bytes, a coalesced memory access would mean 16 consecutive work-items in a wavefront or warp access 16 consecutive single precision words (4 bytes/word) from global memory. This sequential and aligned access would translate into a single transfer by using a single cache line and is therefore coalesced. Whenever an access is misaligned, multiple cache lines are addressed and transfers may be serialised due to a limited number of cache lines. Current architectures do combine an aligned but nonsequential access into a single coalesced memory access. The same holds for uncached memory, whenever

an access is misaligned this results in more memory transfers. A GPU only features a limited number of memory controllers per CU or SM, thus more memory transfers can result in serialising transfers and degrading performance. The main difference between cached and uncached memory transfers is the chunk size determined by a cache line or memory controller. Figure 2.11 shows a graphical representation of an aligned but nonsequential memory access hitting two cache lines, which results in a coalesced memory access. Figure 2.12 shows a graphical representation of an misaligned and nonsequential memory access hitting three cache lines, which results in an uncoalesced memory access.

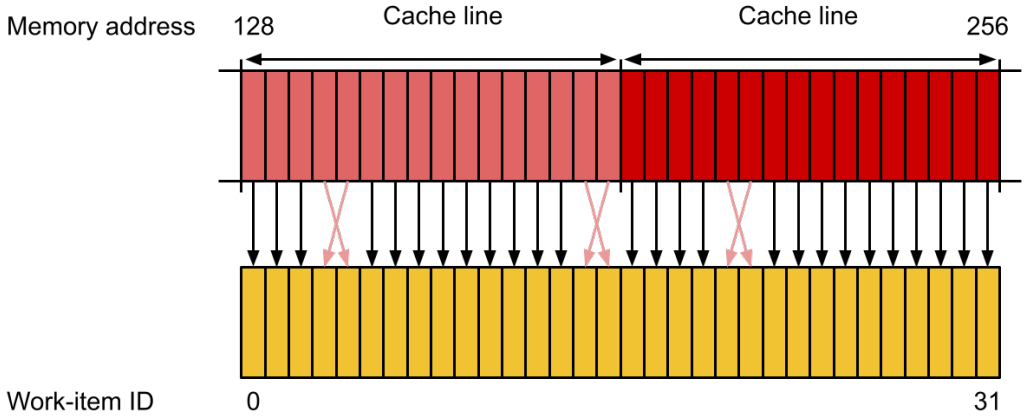


Figure 2.11: Aligned but nonsequential access resulting in a coalesced access

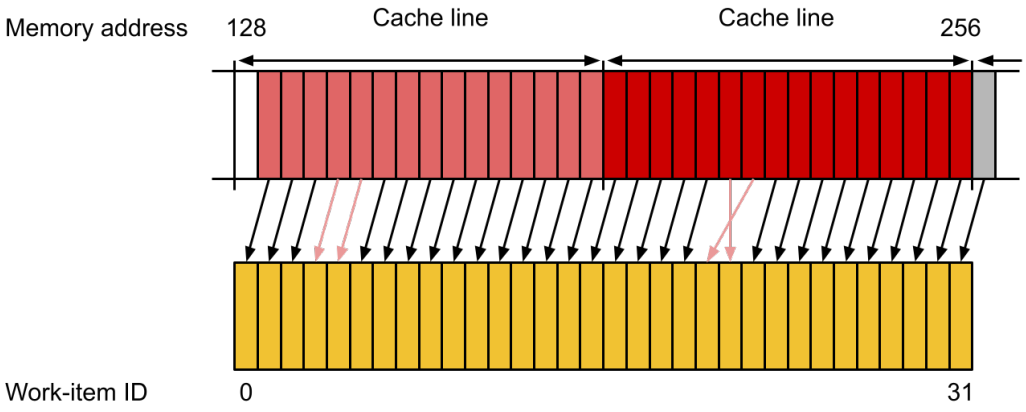


Figure 2.12: Misligned and nonsequential access resulting in a uncoalesced access

Constant memory utilization In many cases elements from arrays are not altered during kernel execution. For these cases it is good practice to use the `__constant` qualifier, which informs the compiler that the elements from an array can be stored in constant memory which is cached. As previously stated, constant memory has a limited size in the order of tens of kilobytes, when a variable qualified as `__constant`

exceeds this size it is stored on global memory which results in lower throughput. When multiple work-items in a wavefront or warp access the same element in a constant array this access is hardware accelerated and is as fast as accessing a register. However if multiple work-items in a wavefront or warp access different elements in a constant array, these accesses are serialised resulting in global memory performance. As such, the `__constant` qualifier is used best when work-items in the same wavefront or warp accesses only a few distinct locations.

Shared memory utilization Whenever work-items in the same work-group access different elements of an array multiple times, using shared memory can benefit performance massively. Shared memory has a much higher bandwidth than global memory has (typically 10x higher) and about twice the bandwidth L1 cache has with far lower latency. Furthermore shared memory doesn't require coalesced accesses, shared memory can thus be filled from global memory using coalesced accesses and afterwards read by work-items in a nonsequential misaligned manner. Shared memory is built up with so-called banks, where each bank has a certain depth and accesses to the different banks can be executed simultaneously. Typically a shared memory has 32 banks of 512 elements, with each element consisting of 4 bytes resulting in a memory size of 64kB. Maximum performance is achieved when `n` memory accesses fall in `n` distinct memory banks, which are then executed simultaneously. If memory accesses of multiple work-items in a work-group fall in the same bank a so-called bank conflict occurs. These accesses to the same bank are serialised which degrade the performance by the number of conflict-free accesses necessary. A typical access pattern that prevents bank conflicts is for each work-item in a work-group access an element from shared memory with the work-item id. This results in each work-item writing or reading a subsection of an array, with all the work-items executing in parallel the whole array is processed. When applying such a parallel access structure it is necessary to synchronise the work-items afterwards. This is done by using `barrier(CLK_LOCAL_MEM_FENCE)`, this function ensures correct ordering of memory operations, where this specific parameter is used for shared memory.

Cache utilization Instead of using shared memory, certain applications can benefit from using L1 cache. L1 is part of the read path on current architectures and offers high performance when cache hit rates are high but is typically smaller than shared memory. Choo et al. [50], however found that L1 cache hit rate is substantially lower when compared to CPU caches. As with shared memory, to achieve high throughput L1 cache needs to be filled from global memory using coalesced memory access patterns, after which a random accesses come at no extra cost. Where

shared memory is not capable of sharing its data across multiple work-groups, L1 is. Data in L1 cache is independent of work-group execution, so whenever another work-group is executing on a CU or SM, data from L1 can be reused in the form of cache hits. However, it is not possible to explicitly control this sharing across multiple work-groups. One last advantage of relying on L1 cache is the fact that it is not needed to synchronise the work-items, as is the case for shared memory.

Hardware utilization In order to maximise performance it is key to utilise available hardware in CUs or SMs and limit practices that cause SPs and CUDA cores to wait for further executions. The number of scheduled wavefronts/warps per CU/SM is a key metric, often called occupancy, that influences the hardware utilisation. On a GPU, kernel instructions are executed sequentially, where multiple work-items execute these instructions in parallel. Previous sections have shown that certain design choices can introduce latencies for work-items, which cause stalls. A solution to hide these latencies and keep the hardware busy, is to schedule multiple wavefronts/warps on a single CU/SM. This allows for pipelining different wavefronts/warps on a single SC/PB which can prevent instruction stalling. Higher number of wavefronts/warps per CU/SM doesn't automatically result in higher performance. Every application has its optimum which is related to shared memory and register usage in the kernel. This is due to the fact that every wavefront/warp executing on a CU/SM requires its own portion of memory to make alternate scheduling possible. Furthermore the work-group size is also an important factor in hiding latencies, this is due to the fact that work-items in a work-group can share data through shared memory which reduces global memory accesses. AMD and Nvidia present a number of heuristics that provide a guideline on these parameters. For AMD architectures it is good practice to schedule at least 4 wavefronts/CU since a typical CU has 4 SCs, 8 to 32 wavefronts/CU is desirable for pipelining however. This can vary depending on the kernel complexity and kernel memory usage. On Nvidia architectures a typical SM also has 4 PBs, resulting in the same 4 warps/SM, with a multiple of that being desirable for pipelining. The optimal work-group size is an integer multiple of the wavefront/warp size, i.e., 64 and 32 work-items. Generally larger work-groups perform better when the scheduled `global` size, G_s, is big.

Figure 2.13 shows an example of scheduling a high enough number of wavefronts/warps per CU/SM, good occupancy, for hiding memory latencies and therefor improve hardware utilization.

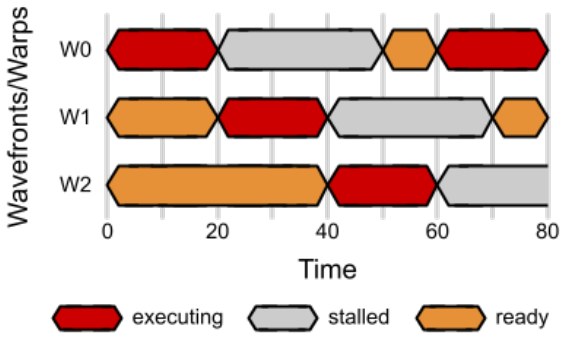


Figure 2.13: Example of scheduling enough wavefronts/warps on a single CU/SM to hide memory latencies and improve hardware utilization.

Prevent thread divergence Conditional statements can severely decrease instruction throughput by causing work-items in the same wavefront or warp to diverge, i.e., to follow different execution paths. These execution paths need to be executed separately which can heavily increase the number of instructions that need to be performed by this warp. Whenever an application requires control flow, the conditional statement should be written such that the number of divergent wavefronts or warps is minimised.

Execution overlapping In order to hide CPU-GPU transfer latency it can be useful to overlap transfers with GPU computations. This can be applied by executing a non-blocking transfers for a portion of the data and launch the kernel on this part of data. Then while the kernel is executing a new chunk of data can be written to a different location on the global memory, thus hiding the transfer latency. Figure 2.14 shows a graphical representation of this technique.

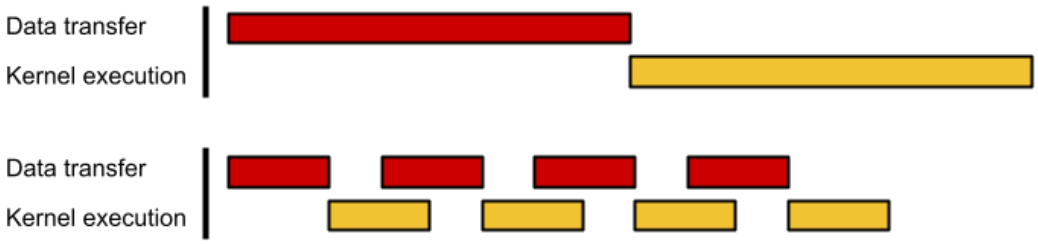


Figure 2.14: Overlapping transfers with execution

General techniques Generally, typical CPU techniques as loop unrolling, compiler optimisations and using specific *math* instructions can improve performance of a GPU kernel.

More information about acceleration techniques for both AMD and Nvidia architectures can be found in the respective optimization guides (AMD [51], Nvidia [52]).

More information about OpenCL programming can be found in [53].

Chapter 3

Related work

This chapter will elaborate on work related to this project with regards to optimisation and acceleration of Bioinformatics software. Section 3.1 focuses on phylogenetics and discusses solutions for the two main scoring functions of phylogenetic trees, i.e., the phylogenetic parsimony function (Section 3.1.1) and the phylogenetic likelihood function (Section 3.1.2). Section 3.2 focuses on population genetics and discusses solutions for the calculation of linkage disequilibrium (Section 3.2.1) and epistasis (Section 3.2.2). Section 3.3 focuses specifically on selective sweep detection software, where multiple tools with their respective methods are discussed. section 3.4 concludes this chapter with a performance overview followed by the decision on the selective sweep detection tool.

3.1 Phylogenetics

Phylogenetics reconstruct the evolutionary history of organisms (taxa) based on shared ancestry. After an MSA has been created, phylogenetic inference methods can construct a phylogenetic tree given an optimality criterion. Due to advances in DNA sequencing technologies, the field of phylogenetics is currently in need for accelerated solutions; tree reconstruction increasingly becomes more computationally intensive with an increasing number of taxa. As with the field of population genetics, phylogenetics have also attracted the attention for optimization and hardware acceleration. Both of the fields also process MSAs to gather information about the genetic composition. The search for the maximum likelihood phylogenetic tree is NP-hard [54], which is why heuristics are inevitably employed to search the tree space for the optimal tree (given some optimality criterion).

Tree-scoring functions are used for evaluating phylogenetic trees. One method for this purpose is the Phylogenetic Parsimony Function (PPF) [55], which is a discrete function that aims to find the phylogenetic tree based on the least number of

mutations among taxa. Another method for evaluating phylogenetic trees is the Phylogenetic Likelihood Function (PLF) [56]. The PLF is the most widely used method for tree construction, and is employed in several phylogenetics tools like RAxML [57], GARLI [58], and MrBayes [59]. Yang and Rannala [60] provide a detailed review of the major methods and principles in phylogenetics.

3.1.1 Phylogenetic Parsimony Function (PPF)

The PPF is used when the main objective is to find the tree topology that requires the least number of mutations to explain the data. Figure 3.1 depicts two possible topologies and the required mutations to explain the observed data at the tips; the tree that requires only one character change is the most parsimonious one. In comparison with the PLF, which relies on likelihood calculation (discussed in the next section), the PPF is considerably less compute- and memory-intensive [61] due to a simpler scoring function that produces an integer output representing the number of evolutionary changes. Because of this, the PPF is highly suitable for large-scale analyses. Various studies have already focused on optimizing the calculation of the PPF. Two algorithms can be used to implement the PPF, i.e., Fitch's algorithm [62] and Sankoff's algorithm [63], with Fitch's algorithm being more commonly used because of lower computational complexity [64].

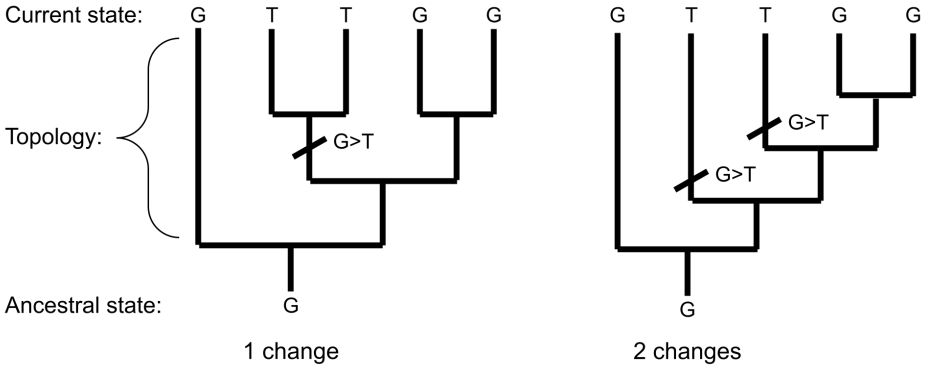


Figure 3.1: Parsimony tree scoring visualisation: less changes in sequence data from the ancestral state to the current state result in a higher parsimony score, and thus a more parsimonious topology.

CPU

Alachiotis and Stamatakis [65] employ an optimized, in-house parallel implementation, dubbed *parsimonator*, as the reference software for evaluating performance of a hardware accelerator (discussed in a following section). This reference software implementation deploys vector instructions such as the Intel SSE (128-bit streaming

SIMD extensions) and AVX (256-bit advanced vector extensions) intrinsic instructions. The study shows (based on execution time comparisons) that the AVX implementation is 1.14x to 1.73x faster than the SSE implementation, yet between 5.6x and 9.65x slower than a FPGA accelerator.

Santander et al. [66] present an optimized solution for CPU, which is used for performance comparisons with their hardware-accelerated PPF implementation. The software (SW) implementation is parallelized and optimized using the OpenCL framework in combination with SIMD instructions. The solution is compared to an unoptimized sequential implementation and a newer version of the *parsimonator* (1.0.2) (available at: `cme.h-its.org/exelixis/web/software/parsimonator/index.html`) tool that is parallelized using OpenMP. All performance values are acquired using two Intel Xeon E5-2630v3 CPUs. The optimized parallel solution is between 8.7x and 77x faster than the sequential implementation, and between 1.1x and 9.3x faster than the parallel AVX implementation of *parsimonator*.

Block and Maruyama [67] present a CPU-optimized solution developed using C++, which is used as reference for their hardware accelerated PPF implementation (discussed in a following section). The C++ solution implements a local search algorithm that relies on the Progressive Neighborhood (PN) [68] search method. A PN search starts with a large neighborhood and takes more topologies into account to ensure a more intensive search. The C++ software solution is compared with the respective FPGA implementation and with TNT [69], a highly optimized tool for parsimony analysis. Using an Intel core i7-860 CPU running at 2.8 GHz as test platform, the software solution is considerably slower than both TNT and the FPGA accelerator, with TNT achieving, on average, comparable execution times with the FPGA accelerated implementation (inferred from Block and Maruyama [67], Table 3).

FPGA

Kasap and Benkrid [70], [71] presented an FPGA implementation based on a systolic-array architecture for the acceleration of the PPF using Sankoff's algorithm [63]. The authors applied fine- and coarse-grained parallelism on multiple FPGAs, with each device hosting a systolic array with 20 Processing Elements (PEs). The accelerated system only supported up to 12 taxa. To evaluate performance, the authors used the Maxwell supercomputer [72] with Intel Xeon processors running at 2.8GHz, and 8 processing nodes, each hosting a Virtex-4 FPGA clocked at 157 MHz. Speedups between 5x and 32,414x were reported based on comparisons with PAUP [73], a software tool for phylogenetics analyses that was executed on an Intel Centrino Duo CPU running at 2.2 GHz.

Alachiotis and Stamatakis [65] accelerated the PPF using a vector-like pipelined architecture on an FPGA. The size of the vector of PEs could be adjusted based on the available FPGA resources. As previously mentioned, the authors employed an optimized software implementation as a reference for assessing performance. Profiling revealed that ancestral-vector computations and final-score computations at the virtual root account for 99% of the total execution time, which were subsequently accelerated. In comparison with the AVX-based implementation, the reported speedups ranged between 5.6x and 9.6x using a Xilinx Virtex-6 FPGA.

Block and Maruyama [67] also employed FPGAs for the PPF, accelerating the complete tree-search algorithm using a versatile approach that is not limited by the number of taxa. As with the SW solution previously mentioned, the accelerated solution uses a local search algorithm that relies on the Progressive Neighborhood (PN) [68] search method. Every algorithmic step was translated into a dedicated hardware unit, thereby allowing them to operate in parallel in a pipelined architecture. The solution is implemented on a Xilinx Kintex-7 FPGA operating at 157 MHz. The performance comparison, already presented in section 3.1.1, shows that the FPGA accelerator achieves speedups in the order of thousands over the respective unoptimized software solution, but only matches the execution times of TNT, even though TNT constructs and evaluates 5x more trees than the FPGA implementation.

The more recent works of Block and Maruyama [64], [74] extend the accelerator design to also implement the Indirect Calculation of Tree Lengths [75] search method. When using the Indirect Calculation of Tree Lengths, the time required to visit all the internal nodes of a tree is fixed at $1/T$, where T is the number of taxa. The benefit of this search method is that the time required for this search does not increase with T [75]. The same Kintex-7 FPGA is used running at 156.25 MHz. When compared to the previous work the approach achieves speedups between 34x and 45x per tree, and between 2x and 6x for the whole local search. When compared with TNT (version 1.1), the speedups per tree range from 2x to 4x, and from 18x to 112x for the whole local search.

GPU

Santander et al. [76] employed a GPU to accelerate phylogenetic tree inference based on the PPF. The authors used OpenCL [3], a framework for parallel programming of heterogeneous systems, and assessed performance on various GPU cards. In a subsequent study, Santander et al. [66] presented a comparative assessment of various parallel-programming frameworks (OpenCL [3], CUDA [49], and OpenACC [77]) on different GPU architectures. The PPF was computed by organizing the entire computational task into independent sub-tasks that you could be pro-

Work by	Year	System details	Achieved speedup/Reference
Satander et al. [78]	2020	CPU + GPU	22x-299x / CPU (single-core)
Satander et al. [66]	2019	CPU + GPU	8.8x-324x / CPU (single-core)
Satander et al. [66]	2019	CPU	8.7x-77x / CPU (single-core)
Block and Maruyama [64], [74]	2017	CPU + FPGA	18x-112x / CPU (multi-core)
Block and Maruyama [67]	2013	CPU	N/A
Block and Maruyama [67]	2013	CPU + FPGA	1x / CPU (multi-core)
Alachiotis and Stamatakis [65]	2011	CPU	1.14x-1.73x / CPU (SSE, multi-core)
Alachiotis and Stamatakis [65]	2011	CPU + FPGA	5.6x-9.6x / CPU (AVX, multi-core)
Kasap and Benkrid [70], [71]	2010	CPU + FPGA	5x-32,414x / CPU (single-core)

Table 3.1: Overview of high-performance computational solutions for the Phylogenetic Parsimony Function (PPF).

cessed in parallel; each sub-task corresponded to a partial calculation of the PPF. A tree topology is stored on the GPU's constant memory since this data structure has to be read only, while the sequences are stored in global memory due to their size. A row-major layout is adopted to ensure coalesced memory accesses. The performance results of the GPU-accelerated solution are compared with a sequential software implementation. Of the three GPUs used for evaluation, the Nvidia GeForce GTX TITAN X achieves the highest performance with speedups ranging from 8.8x to 324x. The lowest performance gain is measured on a dataset with short sequences (759 characters), where the memory transfer overhead is large. Santander et al. [66] shows that implementing the solution using the CUDA toolkit outperforms the OpenCL implementations while both CUDA and OpenCL considerably outperform OpenACC, with OpenCL being between 2.3x to 3x faster than OpenAcc.

Santander et al. [78] present a new solution that solely focuses on processing protein sequences, which results in higher complexity due to the larger number of states (20 amino acids). This work also presents a comparative view on multiple GPU architectures and different implementation frameworks. Profiling the accelerated design revealed that nearly 40% of the execution time is spent on data transfers and pre-processing tasks. By overlapping data transfers with pre-processing tasks, this time can be reduced. Overall, the CUDA implementation on the Nvidia GeForce RTX 2080 Ti achieves the highest performance, with speedups ranging from 22x to 299x when compared to a sequential software implementation.

A summarized overview of the performance-driven solutions for the phylogenetic parsimony function that were reviewed in this section are provided in table 3.1.

3.1.2 Phylogenetic Likelihood Function (PLF)

The PLF is used by Maximum Likelihood and Bayesian inference tools like MrBayes [79] and RAxML [80] to evaluate phylogenetic trees by calculating the likelihood of the tree. The calculation of the PLF is both computationally and memory intensive, and takes approximately between 85% and 95% of the runtime [80]. This amounts to several CPU hours and is therefore of great importance to be accelerated. The PLF is recursively applied, starting from the tips and proceeding toward a virtual root, calculating likelihood vectors at the inner nodes of the tree topology. A detailed explanation of the PLF is provided by Malakonakis et al. [81].

CPU

Pratas et al. [82] improved performance of the PLF on a CPU, a Cell Broadband Engine, and a GPU. The authors developed an OpenMP implementation and parallelized the outermost loop of the PLF to minimize synchronization overheads. The solution is implemented on 3 different systems with multiple CPUs: 2x Intel Xeon Quad-core CPUs (System A), 4x AMD Opteron Quad-core CPUs (System B), and 8x AMD Opteron Dual-core CPUs (System C), and performance comparisons are performed with sequential execution on an Intel Core2 Duo E8400 CPU. Overall System B achieves the highest speedups, ranging from 4x to 11x depending on the dataset. System C achieves the same maximum speedups but lower on-average performance. System A delivered lower performance than the other two systems under test (speedups between 6x and 7x), but performance was more consistent over varying dataset sizes.

Flouri et al. [83] present the Phylogenetic Likelihood Library (PLL), a highly optimized application programming interface for developing likelihood-based phylogenetic inference and post-analysis software. Similarly, Ayres et al. [84], [85] present BEAGLE, an optimized software library that implements both likelihood-based and Bayesian-based development. BEAGLE implements solely the likelihood calculation while the PLL also implements the tree data structure. The PLL implementation employs Intel 128-bit SSE and 256-bit AVX intrinsic instructions, whereas BEAGLE only employs 128-bit SSE instructions. Both libraries support parallel processing, with the PLL relying on Posix threads while BEAGLE uses OpenMP. In the latest work of Ayres et al. [85], a performance comparison is presented between the AVX PLL (Version 2) and SSE BEAGLE (Version 3.1.2) implementations. Both solutions are executed on a single thread of an Intel Core i7-2600 CPU, where the PLL achieves speedups of up to 3.1x over BEAGLE.

FPGA

Alachiotis et al. [86] implement a subset of the PLF functions required to conduct a full real-world tree search on an FPGA, limiting functionality to fixed tree topologies. The solution is evaluated using trees with between 4 and 512 taxa, and compared with an efficient parallel C code for multicore CPUs (a stripped down version of RAxML). A Xilinx Virtex-5 SX240T was used as the target FPGA, which achieved speedups between 3x and 13.5x faster than a single CPU core. When compared with 8 CPU cores, the FPGA implementation led to a slowdown of 0.96x for the 16-taxon tree, and speedups up to 5.08x for the rest of the tree sizes.

Zierke and Bakos [87] present a FPGA-accelerated solution based on MrBayes [79] using the Bayesian Metropolis-Coupled Markov Chain Monte Carlo (MC^3) method. In addition to the PLF, the normalisation and log-likelihood steps of MrBayes are also accelerated. For likelihood calculations, the internal nodes of the tree are processed via a post-order traversal with minimal intervention from the host to reduce CPU-FPGA communication overheads. The solution utilizes on-board memory to cache the output vectors of the computations to minimize host-FPGA communication. A deep pipeline architecture is devised and the solution is implemented on a Xilinx Virtex-6 SX475 FPGA running at 310 MHz, and a Xeon 5500-series CPU is used as the host processor. For performance evaluation, the solution is compared with the software implementation of MrBayes executed on the same Xeon CPU, resulting in speedups between 4.7x and 8.7x.

Recently, Malakonakis et al. [81] implemented the complete RAxML algorithm on a hybrid system. The calculation of the PLF is done on a FPGA while the rest of the algorithm runs on the host CPU. Two different target systems are expored: a Xilinx ZCU102 development board, which consists of a system-on-chip that combines reconfigurable logic with a quad-core ARM A53 general purpose processor, and a cloud-based Amazon AWS EC2 F1 instance that hosts multiple FPGAs connected to Intel Xeon E5-v4 CPUs through PCIe. The first system deploys at most two PLF accelerators at a frequency of 250 MHz, due to memory bandwidth limitations. This implementation is 7.7x faster than a pure software implementation run on a AWS EC2 F1 instance. The AWS-based accelerated system is about 5.2x faster than the software implementation. In comparison with previous work by Alachiotis et al. [86], the implementation on the Xilinx development board is about 2.35x faster.

GPU

Pratas et al. [82] accelerate the calculation of the PLF within MrBayes on a GPU. A fine-grained architecture is adopted where parallelization is done at the likelihood vector entry level. Calculation of each vector entry is assigned to one independent

thread to minimize thread synchronization. To improve performance, data partitioning is done on three levels: a) global partitions are created when the data is larger than the GPU's global memory, b) block partitioning is used to distribute the likelihood array elements among processing engines which are processed independently, and c) thread partitioning for a set of computations. For every call to the PLF, the input data is transferred to the GPU's global memory, and the results are returned when the computation is finished. The percentage of time spent on calculating the PLF is reduced from more than 90% to 5-10%. The CPU-GPU communication however, severely limits overall performance. The resulting speedup, when executed on a Nvidia GeForce GTX 285, is approximately 1.5x over the respective single-thread CPU implementation executed on an Intel Core2 Duo E8400. The presented speedup is scaled with respect to the clock frequencies of the GPU and CPU, which are 1.48 GHz and 3.0 GHz respectively.

Zhou et al. [88] propose an improvement over the work by Pratas et al [82]. While the work presented by Pratas et al. mainly focuses on the GPU-side computations, this work adopts a more hybrid approach where the CPU performs computations in parallel with the GPU. Besides that, the authors employ pipelining in order to reduce the idle time of both platforms and overlap CPU-GPU communication with computations. In order to further improve performance, shared memory is exploited and thread idle operations are reduced to the minimum. When compared with the fastest–at the time of publication–CPU multi-core implementation, the solution achieves speedups between 0.9x to 5.4x. In comparison with the work by Pratas et al., the performance gain is between 7.5x and 12.6x.

Ayres et al. [84], [85] also include GPU acceleration in BEAGLE and present various optimizations in version 3.1.2 [85] to further improve GPU acceleration. Both CUDA and OpenCL have been used in different implementations to target a wide range of GPUs, as well as solutions implementing single- and double-precision arithmetic. Fine-grained parallelization of the likelihood calculation is applied. BEAGLE version 3.1.2 optimized thread utilisation by identifying additional opportunities for parallelization in order to prevent thread idle operations. Data partitioning has been improved to prevent sequential execution of BEAGLE instances on the GPU. Overall, the optimizations in version 3.1.2 focus on higher utilization of the massively parallel architecture of the GPU. When executed on a Nvidia Tesla P100 GPU and two Intel Xeon E5-2690v4 CPUs as host, the solution achieves a 32x speedup over the single-thread PLL [83] software implementation on the same CPU.

A summarized overview of the performance-driven solutions for the phylogenetic likelihood function that were reviewed in this section are provided in table 3.2.

Work by	Year	System details	Achieved speedup/Reference
Malakonakis et al. [81]	2020	CPU + FPGA	7.7x / CPU (multi-core)
Ayres et al. [84], [85]	2019	CPU + GPU	32x / CPU (AVX, single-core)
Flouri et al. [83]	2015	CPU	3.1x / CPU (SSE, single-core)
Zhou et al. [88]	2011	CPU + GPU	0.9x-5.4x / CPU (multi-core)
Zierke and Bakos [87]	2010	CPU + FPGA	4.7x-8.7x / CPU (multi-core)
Pratas et al. [82]	2009	CPU	4x-11x / CPU (single-core)
Alachiotis et al. [86]	2009	CPU + FPGA	3x-13.5x / CPU (single-core)
Pratas et al. [82]	2009	CPU + GPU	1.5x / CPU (single-core)

Table 3.2: Overview of high-performance computational solutions for the Phylogenetic Likelihood Function (PLF).

3.2 Population genetics

As previously stated population genetics studies the genetic composition within one and among different populations. This includes the detection and understanding of footprints caused by evolutionary phenomena such as positive selection, epistasis, recombination, linkage disequilibrium, and genetic drift, among others, which can explain changes in the frequencies of genes over space and time. The current section focuses on computational solutions for linkage disequilibrium and pairwise epistasis.

3.2.1 Linkage Disequilibrium

As the number of sequenced genomes increases and more genetic variation is discovered, the calculation of LD becomes increasingly compute- and memory-intensive. Computational and memory requirements increase quadratically with the number of Single-Nucleotide Polymorphisms (SNPs), while computational requirements also increase linearly with the number of genomes (sample size).

CPU

Chang et al. [89] present an optimized version of PLINK [90] which is a widely used tool for whole-genome association studies and population genetics. The optimized version (PLINK1.9) calculates both Pearson's correlation coefficient and D', as measures of LD. PLINK1.9 implements various improvements, such as bit-level parallelism, vector instructions, and higher memory efficiency than its predecessor. For pairwise LD computations, PLINK1.9 is between 754x and 8,450x faster than PLINK1.07 (initial release), as can be inferred from the reported execution times by

the authors (Chang et al. [89], Table 5).

Tang et al. [91] present LDkit, a parallel computing toolkit for linkage disequilibrium analysis. The tool implements both Pearson's correlation coefficient and D'. Using task-level parallelism to deploy multiple threads/cores, the authors report speedups of up to 12.8x (over single-thread execution) with 32 threads. Performance comparisons with other tools reveal than LDkit does not outperform PLINK1.9 [89], which is between 1.3x and 25x faster. LDkit, however, has a user-friendly graphical user interface.

Zhang et al. [92] present PopLDdecay, a C++ tool for LD decay analysis that can be used to study the rate of recombination in a population. Similarly to the previous tools, PopLDdecay implements both Pearson's correlation coefficient and D'. PopLDdecay does not outperform the second-generation of PLINK, which is approximately 2.7x faster, but it achieves higher memory efficiency, utilizing up to 12x less memory than PLINK, on average.

Alachiotis and Pavlidis [93] present a series of parallelization strategies to overcome the problem of load imbalance when computing LD on multi-core processors. A fine-grained parallelization approach is suitable for large sample sizes, achieving up to 11.1x speedup with 16 threads/cores, whereas a coarse-grained approach is proposed for better parallel performance on long genomes. Because the coarse-grained approach is particularly sensitive to load imbalance (varying SNP density along the genome), a generic algorithm is proposed that achieves up to 2.5x faster processing than the coarse-grained approach on 16 threads/cores. All parallelization alternatives are implemented in the open-source software OmegaPlus [1].

Alachiotis et al. [17] demonstrate that the calculation of LD can be cast in terms of Dense Linear Algebra (DLA) operations. The authors describe the caclulation of LD as a series of Basic Linear Algebra Subprograms (BLAS) [94]–[96] operations, and show that the GotoBLAS [97] approach (now maintained as OpenBLAS [98]) can be used to compute LD as a high-performance General Matrix Multiplication (GEMM). The proposed approach is implemented based on BLAS-Like Instantiation Software (BLIS) [99], i.e., a high-performance framework for rapidly implementing DLA operations using the GotoBLAS approach, and is up to 17x and 6.7x faster than PLINK1.9 [89] and OmegaPlus [1], respectively.

FPGA

Alachiotis et al. [100] present an FPGA accelerator for computing Pearson's correlation coefficient as a measure of LD, using the Infinite Sites Model (ISM) [101]. The hardware architecture is automatically generated based on a number of parameters that were used to explore the accelerator design space. The study reports that

throughput improves when a moderate amount of wide, pipelined population count[1] operators are used instead of a larger number of narrow operators. A host CPU runs an iterative algorithm that schedules execution on the accelerator hardware based on the available number of accelerator instances on the FPGA. To evaluate performance, the proposed solution is mapped on a Xilinx Virtex-7 VX980T-2 FPGA with a clock frequency of 137Mhz. When compared with PLINK1.9 [89] running on a workstation with an Intel Xeon E5-2630 hexa-core 2.6 GHz CPU, the FPGA achieves 50x faster processing than 12 CPU threads, and 200x faster processing than 1 CPU thread.

Bozikas et al. [102] also implement the Pearson's correlation coefficient as a measure of LD, with the architecture supporting the more compute-intensive Finite Sites Model (FSM). An accelerator architecture that supports any number of samples is presented and mapped to a system with four FPGAs. The authors observe that transferring SNPs to the FPGAs is limiting performance, and propose a memory layout that facilitates the parallel retrieval of SNPs through multiple memory controllers. The Convey HC-2ex platform with 4 Xilinx Virtex-6 LX760 FPGAs is used. When compared with PLINK1.9 running on an Intel Xeon E5-2630 CPU at 2.3 GHz, one FPGA is 4.7x faster than 12 CPU threads, whereas processing becomes up to 12.7x faster than 12 CPU threads when all 4 FPGAs are used. Despite using FPGA technology as well, the speedups by Bozikas et al. [102] are lower than the 50x speedup previously achieved by Alachiotis et al. [100] because of the additional support of the FSM model that requires more computations and hardware resources.

GPU

Xian et al. [103] present a GPU-accelerated solution for LD, computing Pearson's correlation coefficient under the ISM model. The authors employ the *__popc* instruction of the CUDA Toolkit API [49] for faster bit counting (population count operation). Furthermore, a data reorganization scheme and atomic instructions are used for reducing memory footprint and latency. The proposed solution is implemented on a cluster of Nvidia Tesla C2075 GPUs (two GPUs per node), achieving speedups between 906x and 1,589x in comparison with a sequential software implementation on an Intel Xeon E5410 quad-core CPU running at 2.33 GHz. The overall processing capacity of the cluster (number of nodes, CPU cores, and GPUs) is not specified.

Theodoris et al. [104], [105] present quickLD, an optimized software for computing LD statistics using either a CPU or a GPU. The GPU implementation is based on the OpenCL framework. The authors focus on handling large-scale datasets by

[1] Population counting describes the operation of counting the number of set bits ('1') in a computer register.

Work by	Year	System details	Achieved speedup / Reference
Tang et al. [91]	2020	CPU	12.8x / CPU (single-core)
Theodoris et al. [104]	2020	CPU + GPU	20x / CPU (SSE, multi-core)
Zhang et al. [92]	2019	CPU	N/A / N/A
Binder et al. [16]	2019	CPU + GPU	7.8x / CPU (multi-core)
Bozikas et al. [102]	2017	CPU + FPGA	4.7x-12.7x / CPU (SSE, multi-core)
Alachiotis et al. [93]	2016	CPU	2.5x-11x / CPU (multi-core)
Alachiotis et al. [17]	2016	CPU	17x / CPU (SSE, multi-core)
Alachiotis et al. [100]	2016	CPU + FPGA	50x / CPU (SSE, multi-core)
Chang et al. [89]	2015	CPU	754x-8450x / CPU (multi-core)
Xian et al. [103]	2013	CPU + GPU	906x-1589x / CPU (single-core)

Table 3.3: Overview of high-performance computational solutions for computing Linkage Disequilibrium (LD) statistics.

introducing a two-step process that separates parsing from processing. This allows for more flexibility in scheduling computation between distant SNPs without increasing memory requirements. For performance evaluation, quickLD is compared with PLINK1.9 on two different computing systems: a) a personal computer with an Intel Core i5-8300H 2.3 GHz CPU and a Nvidia GeForce GTX 1050-M GPU, and b) the Aris supercomputer (*https://hpc.grnet.gr/en/*) with two Intel Xeon E5-2660v3 2.6 GHz CPU and a Nvidia Tesla K40 GPU per node. Using datasets with up to 100,000 samples and 10,000 SNPs on the supercomputer, the authors report up to 29x faster processing than PLINK1.9 (20 threads).

Binder et al. [16] present a portable framework for performing CPU-based SNP comparison algorithms on a GPU. Comparing SNPs is the core of LD calculations. For portability, the implementation of LD is based on OpenCL [3] and maps the BLIS [99] framework onto the GPU. The SNP-comparison framework is evaluated on a Nvidia TITAN V GPU, a GeForce GTX 980 GPU, and an AMD Radeon Vega GPU, and performance comparisons with a BLIS-based CPU implementation [17] are performed. The authors report that the GPU implementation is up to 7.8x faster than the CPU implementation on an Intel Xeon E5-2620v2 6-core CPU running at 2.10 GHz.

A summarized overview of the performance-driven solutions for computing linkage disequilibrium statistic that were reviewed in this section are provided in table 3.3.

3.2.2 Epistasis

Epistasis is the phenomenon where interaction between different genes is antagonistic in such a way that one gene overrules or interferes with the expression of another gene. This section focuses on pairwise epistasis (direct gene-gene interaction). An example of pairwise epistasis is the interaction between genes that control hair color and genes responsible for total baldness. The gene that is responsible for total baldness is epistatic to the gene that controls hair color because the gene for total baldness supersedes the effect of the gene that controls hair color. The gene that controls hair color is called hypostatic to the gene for total baldness.

Detecting pairwise epistasis consists of two stages: a) creation of contingency tables that contain the (multivariate) frequency distribution of the variables, and b) statistical testing of each created table. A contingency table is created for every SNP pair, which leads to excessive compute and memory requirements. Because of this, approximate statistical tests [106], [107] have been proposed in order to shorten analysis times when conducting Genome-Wide Association Studies (GWAS). Commonly used tools for epistasis detection are BOOST [108], MB-MDR [109], and iLOCi [110]. Cordell [111] provides a detailed explanation of epistatis and related statistical methods.

CPU

Wienbrandt et al. [112] present an optimized implementation of the BOOST [108] algorithm, which performs an exhaustive pairwise analysis using statistical regression and is used by PLINK [90]. To improve performance, sample covariance is not supported, and a logistic regression test based on contingency tables is used. This optimization reduces the computational complexity from $O(NT)$ to $O(N+T)$, where N is the number of samples and T is the number of iterations required for a single test. When executed on a system with two Octa-core Intel Xeon E5-2667v4 3.2 GHz CPUs, the optimized version is between 10x and 15x faster than the original PLINK BOOST implementation.

González-Domínguez et al. [113] also optimize the PLINK BOOST algorithm using logistic regression, targeting the Intel Xeon Phi 5110P co-processor with between 57 and 61 simplified Intel CPU cores running at 1.0-1.2 GHz. Optimizations are mainly focused on exploiting the available 512-bit-wide vector instructions, including the *popcount* instruction. In addition, an embarrassingly parallel workflow is adopted to employ the underlying many-core architecture. Moreover, the authors present a heterogeneous CPU/GPU implementation that additionally deploys a Nvidia Tesla K20m GPU. This heterogeneous implementation is between 8x and 33x faster than the Phi-only software implementation, as can be inferred from the

reported execution times by the authors (González-Domínguez et al. [113], Table 3).

FPGA

Wienbrandt et al. [114] present an FPGA-accelerated GWAS epistasis detection tool. The solution combines fine- with coarse-grained parallelism through systolic arrays on multiple FPGAs, resulting in a large number of PEs operating in parallel. The systolic array architecture is used for both the creation of large contingency tables and the application of a statistical test that is adopted from iLOCi [110]. The authors introduce a nearly redundant-free SNP pairing scheme while maintaining load balance among a large number of FPGAs. The proposed solution is implemented on the RIVYERA [115] system that features 128 Xilinx Spartan 6-LX150 FPGAs and two Intel Xeon E5-2620 CPUs as host processor. All FPGAs run at a clock frequency of 150 MHz, and each one of them implements 100 PEs. The accelerated implementation achieves up to 285x faster processing than the iLOCi software executed on two Intel Xeon quad-core 2.4 GHz CPUs.

González-Domínguez et al. [116] also target the RIVYERA [115] system for implementation of the commonly used BOOST [108] algorithm, including its statistical tests. A similar systolic architecture as in the work by Wienbrandt et al. [114] is used for large-scale parallel pairwise contingency table creation and preparation, followed by the statistical tests. A total of 128 FPGAs running at 133 MHz are deployed, with each device hosting 56 PEs. For performance evaluation purposes, the authors created an optimised parallel software implementation using PThreads. The FPGA solution achieves a speedup of 190x when the software implementation is run on an Intel Core i7-3930K using 12 threads.

GPU

Hemani et al. [117] propose a GPU-accelerated pairwise epistasis analysis tool called epiGPU, which uses the OpenCL framework [3]. The solution performs an exhaustive pairwise analysis where each SNP is statistically tested against all other SNPs, resulting in a two-dimensional search grid with calculations distributed over the massive parallel architecture of the GPU. To increase performance, the program does not consider the effect of covariates in the analysis. Also, the authors observe that the slow access to global memory limits performance considerably, and introduce optimizations to the regression algorithm in order to achieve higher utilisation of the faster shared memory on the GPU. Moreover, the CPU-GPU communication overhead is minimised by using bit-packed compression, while the memory access time is minimised by using coalesced memory accesses. The optimizations result in

a 15x speedup over the unoptimised implementation. For performance evaluation, the Nvidia GeForce GTX 580 was used with an Intel Core i7-970 CPU as host processor. In comparison with the respective parallel software implementation running on the host CPU (6 CPU cores), the accelerated epiGPU solution is 15.7x faster.

Yung et al. [118] present GBOOST, a GPU-accelerated implementation of BOOST [108] using the CUDA toolkit. The creation and preparation of the contingency tables as well as the statistical test are performed on the GPU. The log-linear filter, however, is executed on the CPU to avoid thread divergence. The performance of the statistical test calculation is improved through coalesced memory accesses to the global memory. The solution omits the effects of covariates. For performance comparison GBOOST is tested on a Nvidia GeForce GTX 285 and compared to BOOST, which is executed on an unknown CPU running at 3 GHz. GBOOST achieves a 40x speedup compared with BOOST.

Wang et al. [119] present an optimised version of GBOOST [118] called GBOOST 2.0. The solution achieves higher true positive rates than GBOOST through the implementation and consideration of covariates. Performance comparison is performed using the Nvidia GeForce GTX 285, but no host CPU is reported. The authors report a speedup of 1.5x speedup of GBOOST 2.0 over GBOOST.

González-Domínguez et al. [116], in addition to the FPGA implementation previously discussed, also present a hybrid CPU-GPU implementation of the BOOST algorithm. The same contingency table creation and statistical test calculation steps are performed as with the FPGA implementation. The GPU solution, however, uses a single kernel to perform the whole analysis on a batch of SNP-pairs. This improves performance since large tables do not need to be stored in global memory. The resulting SNP information is stored in binary form in global memory before being transferred back to the host. As with the FPGA implementation, the GPU implementation is also compared with the same optimized software solution that employs PThreads, executed on an Intel Core i7-3930K using 12 threads. The GPU solution is also compared with BOOST [108] executed on the same CPU but using a single thread. Using a Nvidia GeForce GTX TITAN GPU, speedups of 269x and 31x are achieved over BOOST and the PThreads implementation, respectively.

A summarized overview of the performance-driven solutions for epistatis evaluation that were reviewed in this section are provided in table 3.4.

3.3 Methods and tools for selective sweep detection

There are a number of tools that apply a variety of methods to detect selective sweeps, these methods rely on the sweep signatures to indicate an affected region. This section will discuss a number of tools and their applied method.

Work by	Year	System details	Achieved speedup / Reference
Wienbrandt et al. [112]	2019	CPU	10x-15x / CPU (multi-core)
Wang et al. [119]	2016	CPU + GPU	1.5x / GPU
González-Domínguez et al. [113]	2015	CPU	N/A
González-Domínguez et al. [116]	2015	CPU + FPGA	190x / CPU (multi-core)
González-Domínguez et al. [116]	2015	CPU + GPU	31x / CPU (multi-core)
Wienbrandt et al. [114]	2014	CPU + FPGA	285x / CPU (multi-core)
Hemani et al. [117]	2011	CPU + GPU	15.7x / CPU (multi-core)
Yung et al. [118]	2011	CPU + GPU	40x / CPU (single-core)

Table 3.4: Overview of high-performance computational solutions for calculating pairwise epistasis.

Nielsen et al. [120] developed a tool called Sweepfinder which applies a parametric test based on composite likelihood, similar to the method of Kim and Stephan [121]. Instead of solely focusing on identifying regions with aberrant frequency spectra, this method is based on considerations of the way the spatial distribution (along the chromosome) of frequency spectra is affected by a selective sweep. The test differs from previous composite likelihood methods in that the null hypothesis is derived from the background patterns of variation in the data itself, rather than a specific population genetic model. The SNP ascertainment process is explicitly taken into account to correct for concomitant biases.

DeGiorgio et al. [122] continued on the work of Nielsen et al. and developed an improved version called Sweepfinder2. Besides the effect of positive selection, Sweepfinder2 also takes background selection and local recombination into account in order to increase sensitivity and robustness. Background selection is a loss of neutral variation due to negative selection [123]. The effect of local recombination is taken into account because background selection may be pronounced in regions of low recombination.

Pavlidis et al. [124] implemented a new method in SweeD, a tool that uses the SFS in order to pinpoint regions of selective sweeps. The SweeD code is based on the work of Nielsen et al. and can be used for whole genome rapid detection of sweeps. SweeD implements calculation of the SFS analytically for demographic models that comprise instantaneous population size changes and, optionally, also an exponential growth as the most recent event. Thereby, a neutral SFS can be obtained without the need to compute the empirical average SFS for the genome. This makes the result more robust.

Voight et al. [125] presented a new test statistic they denote iHS (integrated haplotype score). The statistic identifies loci where strong selection has driven new alleles up to intermediate frequency. This event can be followed by a selective sweep where fixation will take place, or the alleles become balanced polymorphisms. The

iHS relies on the first two signatures, i.e., a shift in the SFS and a subgenomic region with a reduction of the polymorphisms level. The iHS statistic uses the EHH (extended haplotype homozygosity) statistic proposed by Sabeti et al. [126], to determine a measure of how unusual the haplotypes around a given SNP are, relative to the genome as a whole. In other words, at each SNP the iHS measures the strength of evidence for selection acting at or near that SNP.

Alachiotis et al. [1] presented OmegaPlus, a high-performance dynamic programming implementation of the ω-statistic proposed by Kim and Nielsen [45]. The ω-statistic measures the specific localised pattern of LD, the third signature, to accurately pinpoint selective sweeps. OmegaPlus is used for rapid detection of selective sweeps in whole genome data. Overall the LD-based selective sweep detection method has showed to be more fruitful than other detectable signatures due to accuracy and computation complexity. Crisci et al. [127] observed that when compared to other tools for detecting selective sweeps (SweepFinder, SweeD, and iHS), OmegaPlus was found to be the most sensitive to various model parameters, and exhibits the highest true positive rates of the tools.

Alachiotis and Pavlidis [128] present RAiSD (Raised Accuracy in Sweep Detection) that implements a novel, to the authors knowledge, and parameter-free detection mechanism that relies on multiple selective sweep signatures via the enumeration of SNP vectors. The authors introduce the μ statistic, a composite evaluation test that scores genomic regions by taking all three signatures into consideration. Due to the use of SNP vectors to detect the SFS and LD changes, the computational requirements are considerably reduced. A SNP vector is an entire alignment column, which the μ statistic employs as a unit. To compute the μ statistic, a SNP-driven, sliding-window algorithm is employed that reuses calculated data between overlapping windows.

3.4 Discussion and Conclusion

Due to advances in DNA sequencing technologies in the past years, Bioinformatics gradually transformed into a computational discipline that requires scalable algorithms and high-performance processing systems. The use of hardware acceleration and high-performance computing solutions in Bioinformatics is found to be a viable solution for this trend. In this literature research. heterogeneous FPGA-/GPU-based systems and CPU-based algorithmic solutions that boost performance of compute-intensive kernels in the fields of phylogenetics and population genetics have been reviewed, providing insights into the potential of these accelerator technologies in Bioinformatics. Based on the reviewed literature we observe that CPU optimizations can lead to performance improvements up to 18x faster than unoptimized implemen-

tations, while hardware-accelerated solutions empowered by FPGAs and GPUs are capable of reducing analyses times further, achieving speedups up to 77x and 86x, respectively.

From the multiple selective sweep detection tools reviewed, OmegaPlus uses the more fruitful LD signature to locate the selective sweep. When compared to RAiSD, more extensive literature is available that evaluates OmegaPlus as superior in comparison with other tools [127], [129]. Furthermore OmegaPlus is more compute intensive then the other reviewed tools since it examines the LD signature more thoroughly, where RAiSD for example examines the three signatures with a rough approximation. The rough approximation of the three signatures results in high performance but may be suboptimal, as the different signatures might be partially correlated since they depend on the same underlying coalescent trees [45], or lead to conflicting outcomes [128]. OmegaPlus is the chosen tool to accelerate as it implements a full likelihood based evaluation of LD, where LD has been extensively reviewed and found to be more accurate [45].

Chapter 4

OmegaPlus and quickLD

This chapter will give information about the chosen selective sweep detection tool, OmegaPlus, in section 4.1. The target tool for GPU acceleration is decomposed in three main parts, i.e., input data representation, LD computation and ω-statistic computation, which will be explained in the OmegaPlus section. Furthermore a high-performance LD computation implementation, including the knowledge that lies at the foundation of this implementation, will be described in section 4.2.

4.1 OmegaPlus

This section will elaborate on the chosen selective sweep detection tool OmegaPlus [1], [2]. Section 4.1.1 will explain the input data of the tool and how this data is represented within the tool. Section 4.1.2 explains the LD calculation within OmegaPlus from the stored data and how the results are prepared for calculating the ω-statistic. Section 4.1.3 elaborates on the calculation of the ω-statistic.

4.1.1 Input data representation

OmegaPlus can process two types of input data, DNA in the FASTA and VCF (Danecek et al., [130]) format and binary data in the ms (Hudson, [131]) or MaCS (Chen et al., [132]) format. The DNA data formats comprise of nucleotide data with four possible states, 'A', 'C', 'G', and 'T', where one is ancestral and the remaining are derived. The binary data format relies on the Infinite Sites Model (ISM) [133], where individuals can either carry an ancestral state (0) or a derived state (1). An ancestral state corresponds to no mutation while a derived state is used when a mutation has occurred. Where the FASTA format consists of MSA(s), and the VCF (Variant Calling Format), ms and MaCS formats consist of sites of interest, i.e., allele variations. The FASTA format can be seen as an MSA $N * m$ matrix, consisting of N

rows representing the sequences/samples and m columns representing the alignment sites. An example of an MSA with 4 sequences (individuals), each consisting of 29 alignment sites is shown in fig. 4.1, where 5 SNPs are highlighted in red.

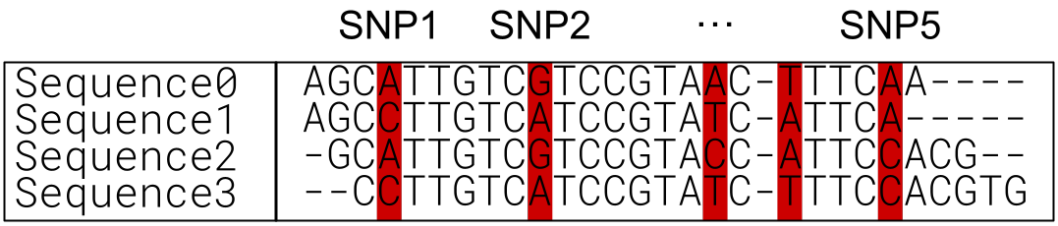


Figure 4.1: MSA example with with 4 individuals, each consisting of 29 alignment sites. The 5 SNPs in this dataset are highlighted in red.

Before storing the data, the FASTA MSA format is pre-processed by removing all the alignment sites that do not contain a SNP, also called the monomorphic sites. This results in the dataset represented by a reduced matrix of dimension $N * W$ only comprising of polymorphic sites, with N sequences/samples and W remaining segregating sites or SNPs.

Now this $N * W$ matrix containing either binary or DNA data is first mapped from characters to 32-bit unsigned integers, where these unsigned integers can be seen as bit vectors. This is an iterative process where 32 characters from a sequence are converted to either a single or multiple bit vectors, which are stored in the `compressedArrays` matrix. For the binary data this process is straightforward, every 0 state character translates to a '0' bit and every 1 state character to a '1' bit. For the DNA data four 32-bit unsigned integers represent the data, one for each of the possible states. For example, when an 'A' state is read, this translates into a '1' bit in the 'A' 32-bit unsigned integer with the other three are set to '0' for that bit position. Figure 4.2 illustrates the data representation for the DNA format. For binary data only the first row of the `compressedArrays` matrix is filled with ancestral (0) or derived (1) states, making it a one-dimensional array of unsigned integers.

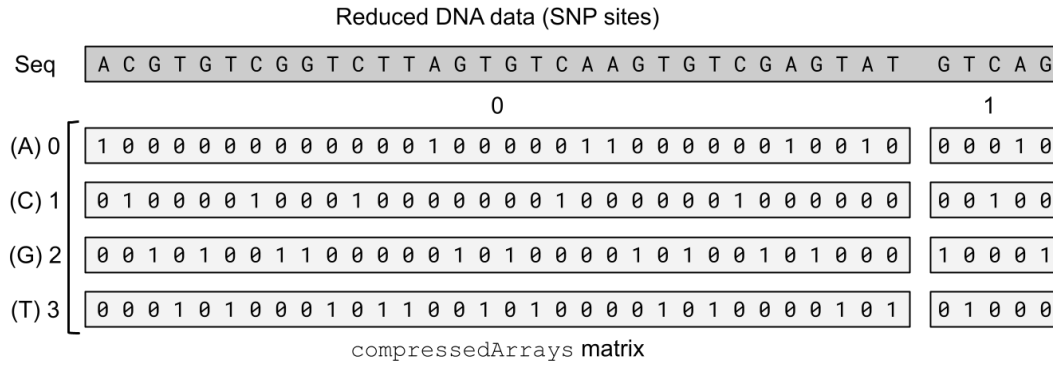


Figure 4.2: Illustration of the data representation in OmegaPlus for the DNA format input data after filtering out the monomorphic sites and compressing the data in 32-bit unsigned integers.

Assume an alignment with a total of W SNPs or segregating sites and N sequences, then the number of elements in each used row of the `compressedArrays` matrix is equal to: $\lceil \frac{N}{32} \rceil W$.

Each of these one-dimensional rows can be seen as a matrix itself with N_{int} rows and W columns. In this matrix each column represents a SNP that is stored as a group of N_{int} 32-bit unsigned integers, where N_{int} is defined as:

$$N_{int} = \lceil \frac{N}{32} \rceil, \tag{4.1}$$

with zero padding the columns if $N \ \texttt{mod}\ 32 \neq 0$.

Effectively the stored data can then be looked at as a genomic matrix named G, with dimension $(N_{int} * 32) * W$. Where the $N_{int} * 32$ padded rows represent the sequences/samples, and the W columns represent the SNPs or sites of interest. The genomic matrix G is often referred to in the remainder of this chapter.

For DNA data, four of these genomic matrices can be formed, one for each row in the `compressedArrays` matrix.

4.1.2 Linkage Disequilibrium computation

After the input data has been processed and stored in the `compressedArrays` matrix the LD computation will execute.

Assuming the ISM with binary data, p_i, p_j are the frequencies of alleles that have the state '1' at SNPs i and j, respectively and p_{ij} is the frequency of the alleles that have the state '1' at both SNPs i and j. Assume a total of N sequences where the states of the input data are denoted by S ($\{$A, C, G, T$\}$ for DNA, $\{0, 1\}$ for binary). First $r^2_{s_i s_j}$ is calculated as follows:

$$r^2_{s_i s_j} = \frac{(p_{s_i s_j} - p_{s_i} p_{s_j})^2}{p_{s_i} p_{s_j} (1 - p_{s_i})(1 - p_{s_j})}, \tag{4.2}$$

where $s_i, s_i \in S$, p_{s_i} is the number of s_i '1' states in SNP i divided by the total number of sequences N, p_{s_j} is the number of s_j '1' states in SNP j divided by the total number of sequences N, and $p_{s_i s_j}$ is the number of $s_i s_j$ combined '1' states divided by the total number of pairs of sequences N. When the input data are in binary format, then $r^2_{ij} = r^2_{s_i s_j}$ since only one mutation array in the `compressedArrays` matrix represents the states. For DNA input data, $r^2_{s_i s_j}$ is calculated for each of the four possible states, where s_i and s_j are extracted from the corresponding state array (0, 1, 2, 3) in the `compressedArrays` matrix. Then r^2_{ij} is calculated as follows, according to [134]:

$$r^2_{ij} = \frac{(v_i - 1)(v_j - 1)v_{ij}}{v_i v_j} \sum_{s_i s_j \in S} r^2_{s_i s_j}, \tag{4.3}$$

where v_i is the number of valid states in SNP i ($v_i \leq 4$), v_j is the number of valid states in SNP j ($v_j \leq 4$), and v_{ij} is the number of valid pairs of states ($s_i, s_j \in S, v_{ij} \leq 16$). When input DNA data is gapless, the number of valid pairs of states is equal to the number of sequences N.

The counting of the '1' states in the `compressedArrays` matrix is done using a lookup table called `bits_in_16bitsLocal`, which is populated using an iterative bit-counting function.

Previous experimental results revealed that LD calculations on binary data require approximately 7–9 times less operations than on DNA data, therefore significantly reducing exection times for the LD calculations [93]. To force the deduction of a DNA alignment to binary within OmegaPlus, the *-binary* option can be used.

In the input parameters of OmegaPlus the user defines a grid size D, which determines a number of equidistant locations C_i ($1 < i < D$) to be assessed. The size of the genomic region, centered at C_i, is determined by the length of the input data and the user specified minimum and maximum window size. Between all possible SNP pairs i and j in the genomic regions, the Pearson's correlation coefficient is calculated, r^2_{ij}. The coefficients are stored in a lower triangular matrix M using a Dynamic Programming (DP) algorithm based on eq. 4.4:

$$M_{i,j} = \begin{cases} 0 & 1 \leq i \leq W, j = i \\ r^2_{ij} & 2 \leq i \leq W, j = i - 1 \\ M_{i,j+1} + M_{i-1,j} - & \\ M_{i-1,j+1} + r^2_{ij} & 3 \leq i < W, i - 1 > j \geq 0 \end{cases}, \tag{4.4}$$

where W is the total number of SNPs in the genomic region. The lower triangular matrix M has `correlationMatrix` as variable name with matrix size $W(W-1)/2$.

Typically, the number D of locations to be assessed is in the order or thousands, which can lead to extended overlapping areas between neighbouring genomic re-

gions, depending on the length of the input data. This overlapping results in redundant computations which are avoided using a data-reuse optimization. For grid position i, genomic region C_i and corresponding matrix M_i, the subsequent matrix M_{i+1} is calculated in two steps. The first step consists of copying the lower n rows of M_i to the higher n rows in M_{i+1}, where n is the number of SNPs in the overlapping area between subsequent genomic regions C_i and C_{i+1}. Step two is the calculation of the remaining rows of M_{i+1}. This optimization can achieve up to an order of magnitude faster overall execution [1].

4.1.3 Omega statistic computation

Every genomic region centered at C_i, consists of W SNPs and is split up into a left L and right R subgenomic region. These subgenomic regions consist of l and $W-l$ SNPs respectively, where ω can be computed for every C_i as follows:

$$\omega = \frac{\left(\binom{l}{2} + \binom{W-l}{2}\right)^{-1}\left(\sum_{i,j\in L} r_{ij}^2 + \sum_{i,j\in R} r_{ij}^2\right)}{\left(l(W-l)\right)^{-1}\sum_{i\in L, j\in R} r_{ij}^2}. \tag{4.5}$$

The numerator in the ω-statistic quantifies the level to which average LD is increased on the left and right side of selective sweep, against the level across the site of the selection in the denominator. For every selective sweep center C_i, ω is assessed for all SNP intervals in subgenomic regions L and R in $l(W-l)$ number of steps, illustrated in fig. 4.3. These subgenomic regions typically consist of a few thousand of SNPs resulting in multiple million steps in which ω is computed. The goal is to find the maximum ω value and the corresponding l of all the steps that lie within the borders of the candidate region. All $\sum_{i,j\in L} r_{ij}^2$, $\sum_{i,j\in R} r_{ij}^2$, and $\sum_{i\in L, j\in R} r_{ij}^2$ values required by the ω-statistic are retrieved from matrix M.

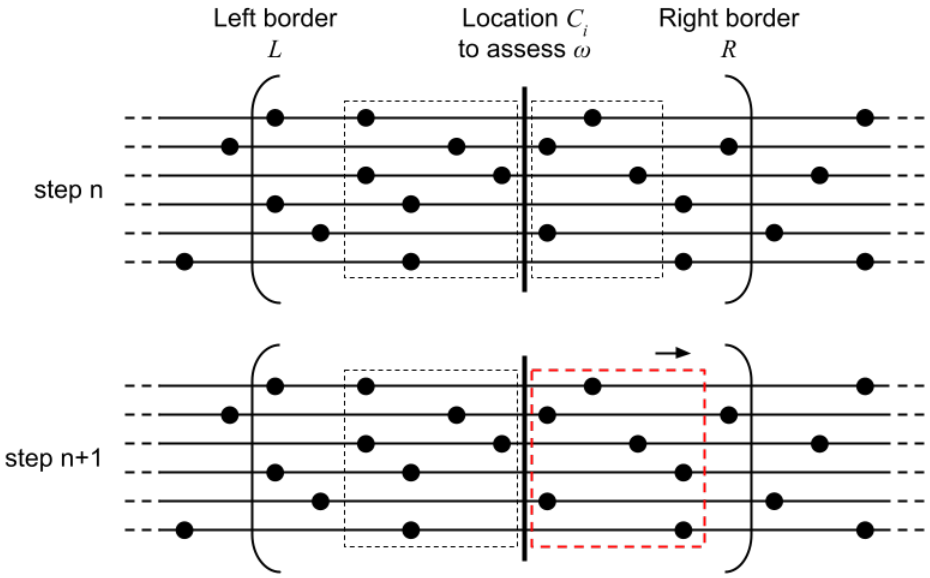


Figure 4.3: Illustration of two consecutive ω computations in the subgenomic regions L (left border) and R (right border) at center location C_i (thick line). The SNPs in the dashed squares are included in each computation step with LD values taken from M. At step $n+1$ note the shift of the dashed square with the smallest interval (adapted from [2].

Figure 4.4 shows a general view of the work-flow in OmegaPlus as described in this chapter.

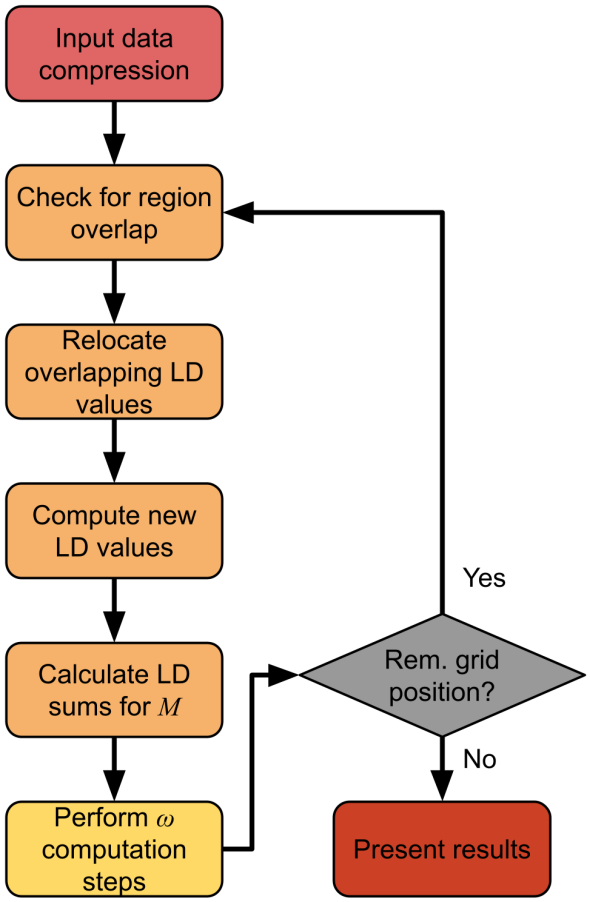


Figure 4.4: General view of the work-flow in OmegaPlus with the described steps. First input data is processed and compressed, which after LD values are relocated in matrix M where needed and new values are computed. With the computed LD values the ω-statistic is computed. This process is repeated for all the grid positions specified by the user.

4.2 High-performance LD

In this section a deeper understanding of the computation of LD is given with an corresponding implementation on both the CPU and GPU architecture. In section 4.2.1, the computation of LD is explained in terms of Dense Linear Algebra (DLA) operations. With this knowledge a high-performance implementation is developed for both the CPU and GPU architecture, named quickLD and described in section 4.2.2.

4.2.1 LD as Dense Linear Algebra operations

Alachiotis et al. [17] present a deeper understanding of LD and observed that the calculation of LD can be rewritten in terms of Dense Linear Algebra (DLA) operations. This allows for high-performance implementations for various microprocessor

architectures due to collective knowledge in the high-performance computing (HPC). The rest of this section explains the casting of LD as DLA operations.

When assuming the ISM where '0' represents the ancestral state and '1' the derived state, the allele and haplotype frequencies can be computed using linear algebra. When given N sequences, the allele frequency p_i of SNP s_i can be computed using linear algebra operations as follows:

$$p_i = \frac{s_i^T s_i}{N}. \tag{4.6}$$

The counting of the number of derived states in s_i is possible by calculating the dot product of the bit vector s_i with itself. The haplotype frequency with SNPs s_i and s_j can then be computed the same:

$$p_{ij} = \frac{s_i^T s_j}{N}. \tag{4.7}$$

Then using eq. 4.6 and 4.7, the standard coefficient of LD is equal to:

$$\begin{aligned} D_{ij} &= p_{ij} - p_i p_j \\ &= \frac{1}{N}(s_i^T s_j) - \frac{1}{N^2}(s_i^T s_i)(s_j^T s_j). \end{aligned} \tag{4.8}$$

As with the LD computations in OmegaPlus, D_{ij} should be computed for for all possible pairs of SNPs, s_i and s_j, in a region of W SNPs. Alachiotis et al. note that every dataset that consists of more than one SNP can be regarded as a genomic matrix G, where each column in the matrix is a SNP. Now the LD computations can be written as matrix multiplications with the following sequence of DLA operations:

$$\begin{aligned} H &= \frac{1}{N} G^T G \\ D &= H - pp^T, \end{aligned} \tag{4.9}$$

where H is a matrix with all the possible haplotype frequencies, P_{ij} and the matrix D is then formed by subtracting the product of allele frequencies from H. When using the Level 3 Basic Linear Algebra Subprograms (BLAS3) [135] operations, which are essentially matrix multiplications of different forms, these DLA operations can be mapped efficiently on modern day computer architectures with hierarchy of caches for memory. Computing the haplotype frequency matrix H is of $O(n^3)$ complexity, as it is a matrix multiplication. The subtraction of the allele frequencies is of $O(n^2)$ complexity since it an outer product of vector p with itself. The higher complexity of calculating H results in it dominating the overall required operations, and thus optimizations focused on computing H have the biggest impact. The remainder of this

chapter will focus on optimizing the computation of the haplotype frequency matrix H.

The computation of H is essentially a General Matrix Multiplication (GEMM) operation, which can be implemented using the GotoBLAS [97] approach for high-performance GEMM. The core of the GotoBLAS approach is a highly optimized GEMM kernel, this kernel has a particular shape and implementation.

With matrices A, B and C with dimensions $m * k$, $k * n$ and $m * n$ respectively, the optimized GEMM operation computes the following:

$$C = \alpha AB + \beta C. \tag{4.10}$$

The input matrices, A, B and C, are partitioned in the k dimension, reducing the GEMM operation size to $m * k_c$, $k_c * n$ and $m * n$ respectively, with $m, n \gg k_c$. This smaller GEMM operation is the actual optimized GEMM kernel in the GotoBLAS approach. The performance of the GotoBLAS GEMM kernel can reach close to the peak performance of the architecture ($\approx 90\%$), if the kernel parameters are tuned to the specific computer architecture.

In order to fully utilize all cache levels in the architecture, each matrix is further partitioned in specific manner. This is achieved by implementing each GEMM kernel as a series of block-panel multiplications, where these block-panels are packed into contiguous memory. The resulting matrix multiplication is implemented as a blocked-dot product.

Given the genomic matrix G with N number of sequences and W number of SNPs, with typically W being much larger than N, the computation of the haplotype frequency matrix:

$$H = \frac{1}{N} G^T G, \tag{4.11}$$

can be seen as a GEMM operation with A being G^T and B being G, with dimensions $m * k$ and $k * n$ respectively. Now k can be seen as the number of sequences with $m = n \gg k$, as m and n are the number of SNPs. With $\alpha = N^{-1}$ and $\beta = 0$ the computation of H can be cast to the highly optimized GEMM kernel as every input matrix and output matrix are already of the correct shape and can be partitioned in the k dimension. As the number of sequences (k) increases due to advances in DNA sequencing technologies, the GotoBLAS approach does not need to be changed because the partitioning is happening in the k dimension. Thus, the GotoBLAS approach to implementing the GEMM operation is suitable for LD computations, now and in the future.

4.2.2 BLIS based implementation

Theodoris et al. [104], [105] present quickLD, a highly optimized GEMM implementation using the GotoBLAS approach for LD computation. The software utilizes the BLAS-Like Instantiation Software (BLIS) [99] for rapid implementation of this GotoBLAS approach for high-performance DLA on both CPU and GPU. The GPU-accelerated version extends the work in BLIS to computing LD on the GPU, as described by Binder et al. [16]. With the BLIS framework, only a highly efficient micro-kernel needs to be implemented, which is a much smaller GEMM operation than the GotoBLAS approach. This results in high-performance GEMM implementation with existing parallelization schemes available in the framework. The developed tool can reach up to 95% and 97% of the theoretical peak performance of a CPU and a GPU respectively.

CPU

Where the GotoBLAS GEMM kernel expected dimensions $m * k_c$, $k_c * n$ and $m * n$ for matrices A, B and C respectively, the BLIS micro-kernel performs the GEMM operation on much smaller dimensions $m_r * k_c$, $k_c * n_r$ and $m_r * n_r$, with $k_c \gg m_r, n_r^2$. The partitioning to the smaller matrix dimensions is performed in a total of five loops around the micro-kernel itself.

In the fifth loop the output matrix C and input matrix B are partitioned to $m * n_c$ and $k * n_c$ respectively, in the fourth loop input matrices A and B are partitioned to $m * k_c$ and $k_c * n_c$ respectively, in the third loop C and A are partitioned to $m_c * n_c$ and $m_c * k_c$ respectively, in the second loop C and B are partitioned to $m_c * n_r$ and the final size $k_c * n_r$ respectively and in the first loop around the actual micro-kernel C and A are partitioned to their respective final sizes $m_r * n_r$ and $m_r * k_c$.

As with the GotoBLAS approach, the BLIS implementation performs partitioning in the k dimension, this is executed in the fourth loop, after which the block-panel multiplications are performed. The k dimension is partitioned into chunks of size k_c where the resulting packed matrices, called A_p and B_p, are stored in a new contiguous memory. These matrices are showed in the third lowest layer of fig. 4.5 and the following loops refer to this new contiguous memory. The BLIS micro-kernel itself is shown in the top layer of fig. 4.5.

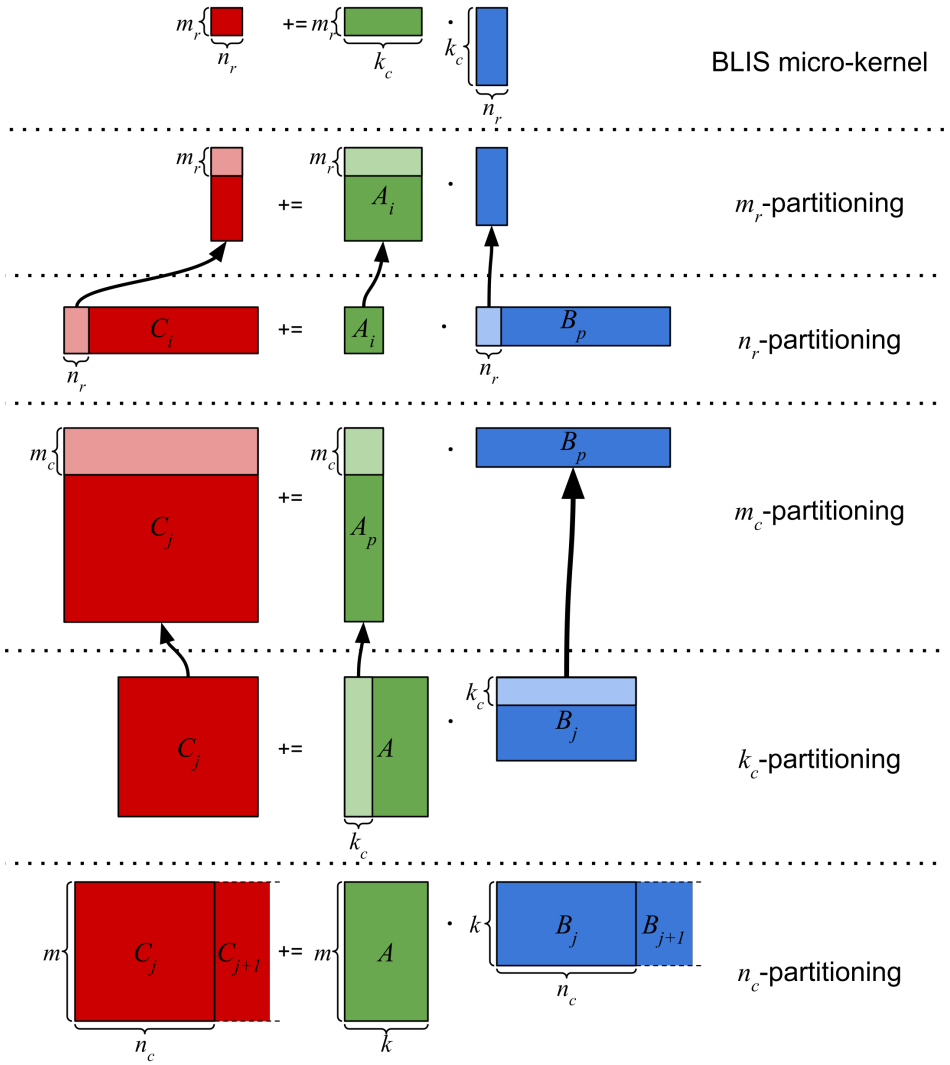


Figure 4.5: Illustration of the GotoBLAS approach of the GEMM operation with the BLIS micro-kernel. From bottom to top, matrices C and B are first partitioned in the n dimension, after which the input matrices are partitioned in the k dimension and stored in contiguous memory. The matrices are then partitioned into cache optimal dimensions for block-panel multiplications. At the top of the figure the BLIS micro-kernel is showed which is a smaller kernel implementation than the GotoBLAS approach (adapted from [17] and [136]).

Recall the genomic matrix G, where SNPs are stored as groups of N_{int} 32-bit unsigned integers or vectors in OmegaPlus. QuickLD uses the same data representation where the SNPs are stored in 64-bit vectors for the CPU implementation. The BLIS micro-kernel can now be used for the computing the haplotype frequencies as we earlier stated that its computation:

$$H = \frac{1}{N} G^T G, \tag{4.12}$$

can be cast as GEMM operation. The BLIS micro-kernel however is designed for double-precision floating-point matrix multiplications instead of binary data. Thus

the kernel needs to be adapted to this by changing the matrix data types to 64-bit long unsigned integers.

The core of the micro-kernel however, can also be rewritten due to the binary data. Where the computation of the haplotype frequency of SNPs s_i and s_j is as follows:

$$p_{ij} = \frac{s_i^T s_j}{N}, \tag{4.13}$$

which has the same result as the following, computer architecture optimal, operations:

$$p_{ij} = \frac{1}{N}\texttt{POPCNT}(s_i \& s_j), \tag{4.14}$$

where the a bit-wise AND (&) operation between bit vectors s_i and s_j and then counting the number of '1' bits in the resulting vector with an optimized bit-count instruction, has the same effect as calculating the dot product between the two vectors. However, each SNP is stored as a group of N_{int} unsigned integers, which requires the following computation for retrieving the haplotype frequency:

$$p_{ij} = \frac{1}{N}\sum_{k=0}^{N_{int}} \texttt{POPCNT}(s_i \& s_j). \tag{4.15}$$

QuickLD however, omits the division by the number of sequences N in the core of the adapted BLIS micro-kernel. The computation of the actual haplotype frequencies, as well as the allele frequencies, is performed together with the Pearson correlation coefficient r_{ij}^2. These computations are executed outside of the BLIS micro-kernel using the results from matrix C, consisting of the absolute haplotype '1' states, $s_i^T s_j$, and another matrix consisting of the absolute allele '1' states, $s_i^T s_i$.

GPU

The GPU implementation of quickLD is an extension of BLIS to map the computation of the absolute haplotype '1' states, $s_i^T s_j$ or matrix C, on the GPU, as described by Binder et al. [16]. The Pearson correlation coefficient computations are performed on the CPU as described with the CPU implementation of quickLD. The approach of Binder et al. is to first cast a model GPU hardware abstraction onto the CPU abstraction underlying BLIS. Then software parameters are determined that guide how the GPU kernel is to be written using the BLIS framework, this is done by leveraging the analytical models in Low et. al. [137].

An abstraction that is often used for mapping the GPU architecture to the CPU architectures is the SIMD/SIMT abstraction [138]. Where the SIMD (Single Instruction Multiple Data) execution model applies a single instruction on similar datasets,

resulting in simultaneous execution by multiple execution units. SIMT (Single Instruction Multiple Threads) is the thread equivalent of SIMD, where instruction-level parallelism is improved by mapping the same instruction on multiple threads to execute on different datasets.

On current CPUs, multiple SIMD functional units are present in every core which allows for multiple SIMD instructions to be executed in parallel. Queuing sufficient number of SIMD instructions increases performance due to pipelining of these instructions.

For the explanation of the abstraction recall section 2.2, in which we stated that multiple wavefronts/warps can execute independently and on different SIMD Cores (SCs)/Processing Blocks (PBs) at the same time. Each SC/PB on the GPU can then be seen as a SIMD functional unit on the CPU. When taking the pipe-lining of a SIMD functional unit into consideration, this abstraction is further confirmed. Recall that in order to hide latencies for work-items or Stream Processors (SPs)/CUDA cores, a multiple number of wavefronts/warps should be scheduled on a SC/PB, lets call this number L_{fn}. This number is equal to the latency of an instruction, where L_{fn} is made equal for all instruction for simplicity of the abstraction. Now wavefronts/warps can be seen as the equivalent of SIMD instructions, as the multiple scheduled wavefronts/warps can be seen as multiple pipelined SIMD instructions, where every wavefront/warp has independent outputs.

On the basis of the above abstraction, we can assume that a GPU Compute Unit (CU)/Streaming Multiprocessor (SM) is the equivalent of a CPU core. This is a logical continuation due to the fact that every CPU core contains multiple SIMD functional units and every CU/SM contains multiple SCs/PBs.

Now with this CPU/GPU abstraction the authors describe how the GPU kernel is designed for LD computation within the BLIS framework.

The CPU BLIS framework implementation uses the cores to perform both the n_r and m_c partitioning in the second and third loop around the micro-kernel in a hierarchical fashion, resulting in each core computing an independent $m_c * n_r$ matrix of C [139]. Similarly, Binder et al. appoint matrices of size $m_c * n_r$ of C to each CU/SM on the GPU after parallelizing the second and third loops around the micro-kernel amongst the available GPU CUs/SMs.

The $m_c * n_r$ matrices assigned to the CUs/SMs are further partitioned to smaller sizes and computed by the wavefronts/warps scheduled on that particular CU/SM. The partitioning is performed as follows, each $m_c * n_r$ matrix is divided into a $m_r * (n_r/L_{fn})$ sub-matrix, which is computed by multiplying two input matrices, A and B, of sizes $m_r * k_c$ and $k_c * (n_r/L_{fn})$. Wavefronts/warps scheduled on the same SC/PB are assigned sub-matrices in the same row. Wavefronts/warps which are executing simultaneously, and thus scheduled on different SCs/PBs, are assigned

sub-matrices from the same column.

The developed GPU framework from Binder et al., implements all the content from the second loop around the micro-kernel to the BLIS micro-kernel itself on the GPU. The fifth to third loop are executed on the CPU where data is transferred to the GPU in order to correctly set up all the parameters and needed data. This implemented parameterized GPU kernel first loads a sub-matrix of A into shared memory, which after computations are performed assuming that A resides in shared memory while B is retrieved from global memory. The parameterization is done via C macros which are present in a C header file. The four variables in the header file are m_c, m_r, k_c and n_r which correspond with the required BLIS framework values. The values of these variables should be tuned to the available hardware resources on the GPU, such as memory sizes, cache line width, wavefront/warp size, instruction latency and number of CUs/SMs and SCs/PBs.

4.3 Acceleration target selection

This chapter has given an insight in the chosen selective sweep detection tool OmegaPlus. The workflow of the tool has been explained on the basis of three main parts, i.e., input data representation, LD computation and at last ω-statistic computation.

In order to achieve the best results the following sub-question was set up to target the most time consuming parts:

- Which parts of the state-of-the-art tool are the most computationally intensive to target for acceleration?

In this chapter we gained an insight in the computational intensity of computing LD and the ω-statistic, which are the expected compute intensive parts to accelerate. After profiling the main parts of the tool, with a dataset comprising of 7,000 samples and 13,000 SNPs, the LD and ω-statistic computation together take up 98.2% of the total execution time. Solely the LD computation takes up 50.3% of the execution time and solely the ω-statistic computation takes up 47.9% of the execution time. Table 4.1 shows the complete profiling results of the OmegaPlus tool for the used dataset.

	Time (s)	Time (%)
Parsing input data	4.14	1.8
LD computation	114.7	50.3
ω computation	109.1	47.9

Table 4.1: Overview of OmegaPlus profiling result

From the profiling together with the information provided in this chapter, we can conclude that both the LD and ω-statistic computation are the most computational intensive parts of OmegaPlus.

Furthermore, in section 4.2 a deeper understanding of the computation of LD is given together with a high-performance implementation using the BLAS approach. This implementation, quickLD, which extends the BLIS framework, utilizes the computing power of the GPU architecture and achieving up to 97% of the theoretical peak performance. Given this implementation, the choice has been made to implement an adaptation of this already existing high-performance tool in order to reduce the workload of this project and prevent reinventing something that is already researched and developed. The following chapter will elaborate on this adaptation as well as on the design of the ω-statistic GPU implementation.

Chapter 5

Designs

This chapter will give an insight into the adaptation of quickLD as well as the developed implementation for GPU-accelerated ω-statistic computation. First an introduction will be given about the general idea of the implementation in section 5.1. Section 5.2 will elaborate on the quickLD adaptation and which exact parts of the existing tool are used for the GPU-accelerated LD computation. Section 5.3 will describe the GPU kernel itself and the design choices that have been made to improve performance. In section 5.4 the developed C-code on the host CPU will be described, i.e., the preparation of the data, setting up arguments and launching of the actual GPU kernel. The last section, section 5.5, will give an complete overview of the design to give a clear understanding of the final implementation.

5.1 Introduction

The overall idea of the GPU-accelerated implementation of OmegaPlus is to extend the original tool in order to boost performance, improve throughput and reduce execution times. This is achieved by introducing a specific OmegaPlus executable called `OmegaPlus-GPU`, which utilizes GPU acceleration. This version of OmegaPlus can be compiled with the `Makefile.GPU.gcc` makefile which will create the corresponding executable. The GPU-accelerated functions are inserted into the original `main` function and general header file `OmegaPlus.h` by using the `#ifdef` and `#infdef` directives, which react to the defined `_USE_GPU` flag in the corresponding makefile. All inserted GPU specific functions or adapted functions from OmegaPlus are located in a new C-file named `OmegaPlus_gpu.c`.

The previous chapter stated that the LD and ω-statistic computation are the two compute intensive parts within OmegaPlus and are therefore targeted for GPU acceleration.

Chapter 4 also concluded that the LD computation can be accelerated using

an adaptation of the high-performance BLIS implementation in quickLD, which will further be elaborated on in section 5.2. This will essentially be a stripped down version of the tool that fits well into the existing work-flow and code of OmegaPlus, thus preserving most of the original tool.

The accelerated ω-statistic computation is an adaptation of the original function, `computeOmegas`. This function performs all the ω computation steps per grid position, shown in fig. 4.3, iteratively. The general idea of the accelerated function is to perform these multiple ω computation steps simultaneously on the GPU, leveraging its massive parallel capabilities. For this approach the variables needed for each computation step are packed in contiguous memories and transferred to the GPU for kernel execution. The kernel computes many ω values in parallel, which after they are sent back to the CPU for further processing.

Dynamic kernel execution Within the sliding window algorithm for computing ω-statistics, the number of computation steps can vary a lot from grid position to grid position. These large variations are the result of varying number of SNPs in either the left or right subgenomic region. This is due to a non-uniform SNP distribution along the genome, of which the effect is more profound at the edges of the complete genome. This non-uniformity results in subgenomic regions that can exhibit a relatively low number of SNPs which are taken into account for the ω-statistic computation.

Depending on the user specified window parameter values, the number of SNPs exhibited in a subgenomic region, can vary from a few tens to thousands due to the non-uniformity. Given the number of ω computation steps is the product of the number of SNPs in the subgenomic regions, the number of steps can vary from a few thousands to millions from grid position to grid position.

These two situations, where the number of computations differ massively led to the decision of making two different kernels. One suitable for a low number of simultaneous computations and one for a high number of simultaneous computations.

Original omega computation Recall section 4.1, where the OmegaPlus ω-statistic computation for every genomic region centered at C_i was explained. For every computation step, seven sliding window dependable variables are needed for computing an ω value which are either read from memory or computed.

As stated in section 4.1, all the needed Pearson correlation values are retrieved from the lower triangular correlation matrix M. The left, L, and right, R, sub-region correlation values, $\sum_{i,j \in L} r_{ij}^2$ and $\sum_{i,j \in R} r_{ij}^2$ are stored in respective floating point variables `LS` and `RS`. The third correlation value needed for the ω-statistic computation, $\sum_{i \in L, j \in R} r_{ij}^2$, is calculated by subtracting the left and right sub-region correlation

values from the complete region correlation value. Lets assume the complete grid position centered genomic region is called S, which is the combination of the two sub-regions, L and R. The complete genomic region correlation value, $\sum_{i,j \in S} r_{ij}^2$, from M is stored in `TS`, where $\sum_{i \in L, j \in R} r_{ij}^2$, is then calculated as follows: $\texttt{TS} - \texttt{LS} - \texttt{RS}$.

The number of SNPs in the left and right subgenomic region window, l and $W - l$, are stored in integer variables `k` and `m` respectively, with W being the number of SNPs in the genomic region ω is computed for. In every step, one or both of these values change with the smallest interval as shown in fig. 4.3. The values of $\frac{l}{2}$ and $\frac{W-l}{2}$ are computed from `k` and `m` and are stored in integer variables `ksel2` and `msel2` respectively.

Left sub-region variables, `LS` and `k`, and right sub-region variables, `RS` and `m` only depend on the state of the sliding window in their corresponding sub-region. Variable `TS` depends on the sliding window in both the left and right subgenomic region.

The sliding window algorithm is implemented as nested loop where the outer loop updates the left subgenomic region variables, `LS`, `k`, `ksel2`, and the inner loop updates the right subgenomic region variables, `RS`, `m`, `msel2` and `TS` which depends on both regions. Every iteration performs a single computation step or ω-statistic computation and determines if this computed value is the up to the current loop maximum value. Per assessed grid position this results in l outer loop iterations, $W - l$ inner loop iterations and $l(W - l)$ total computation steps. Algorithm 1 shows pseudocode of the sliding windows algorithm with ω-statistic computation and window location storing.

Algorithm 1 Compute omega values algorithm

function COMPUTEOMEGA($LS, RS, TS, k, ksel2, m, msel2$)
  $num \leftarrow (LS + RS)/(ksel2 + msel2)$
  $den \leftarrow (TS - LS - RS)/(k * m)$
  $omega \leftarrow num/den$
  return $omega$
end function
procedure COMPUTEOMEGAVALUES
  $MaxOmega \leftarrow 0$
  for each left region SNP interval **do**
    $LS \leftarrow LeftRegionCorrelation$
    $k \leftarrow LeftWindowSNPs$
    $ksel2 \leftarrow k * (k - 1)/2$
    for each right region SNP interval **do**
      $RS \leftarrow RightRegionCorrelation$
      $m \leftarrow RightWindowSNPs$
      $msel2 \leftarrow m * (m - 1)/2$
      $TS \leftarrow RegionCorrelation$
      $TmpOmega \leftarrow$ COMPUTEOMEGA$(LS, RS, TS, k, ksel2, m, msel2)$
      if $TmpOmega > MaxOmega$ **then**
        $MaxOmega \leftarrow TmpOmega$
        $LeftWindow \leftarrow LeftSNP$
        $RightWindow \leftarrow RightSNP$
      end if
    end for
  end for
  $RegionMaxOmega \leftarrow MaxOmega$
  $RegionLeftWindow \leftarrow LeftWindow$
  $RegionRightWindow \leftarrow RightWindow$
end procedure

In section 5.3 the design choices of the GPU kernels are explained, where section 5.4 describes the preparing of the data, data transfer and kernel launch of the two different kernels. Both of these are closely related as a specific data access pattern within a kernel, requires the host CPU to pack the data in the correct way.

5.2 quickLD adaptation

The GPU LD computation in OmegaPlus is an adaptation of the GPU-accelerated BLIS framework implementation in quickLD. This section will describe which parts of quickLD are adapted and implemented and which are omitted from the adaptation.

As described in the work by Theodoris et al. [104] and in section 4.2, quickLD uses the same compressed data representation as OmegaPlus does. The genomic matrix G is used and SNPs are stored as groups of N_{int} unsigned integers which can be seen as bit vectors. The data parsing and preparation algorithm of quickLD can thus be omitted since this is similarly implemented in OmegaPlus.

The tools only differ in the size of each bit vector the SNPs are stored in, where OmegaPlus uses 32-bit unsigned integers, quickLD uses 64-bit unsigned integers. Every function adopted from quickLD that has the genomic matrix G as parameter, `compressedArrays` in OmegaPlus, is thus adapted to process 32-bit unsigned integers. Any depending parameters are taken from the OmegaPlus parsing algorithm.

Another difference between OmegaPlus and quickLD is the steps around the LD computation and the computation itself. OmegaPlus computes a number of LD values per grid position and sums the values where needed to form the lower triangular matrix M. Furthermore in every grid position iteration, the last and current genomic region is checked for overlap in order to prevent redundant LD values computation, which is accomplished with a data-reuse optimization.

The quickLD implementation only computes the Pearson correlation values, r_{ij}, and computes the values at once for all possible SNP pairs. These correlation values are stored in an W^2 resulting matrix, with W being the number of SNPs in the complete genomic region. This resulting matrix is effectively the full matrix M, instead of the lower triangular matrix, without the computation of the summed values.

The GPU-accelerated version of OmegaPlus is adapted to this approach. The quickLD implementation is called outside of the for-loop that iterates over the grid positions, and the needed values are copied from the quickLD resulting matrix and stored in the lower triangular matrix M. These correlation values stored in M are then summed using the original OmegaPlus function. Furthermore, the original data-reuse optimization is also retained in order to prevent redundant data transfer and correlation value summing.

In order to give a good understanding of the adaptation a short overview of the implemented parts will be given.

The GPU adaptation of quickLD in OmegaPlus is limited to a stripped down version of the quickLD `correlate_gpu` function including its child functions. This stripped down function receives the compressed input data of OmegaPlus, denoted as genomic matrix G, all absolute allele '1' states $s_i^T s_i$, the number of SNPs, W, the

number of unsigned integers for each SNP, N_{int}, and the number of sequences N. Within the function the input data matrix is first transposed and stored in a new contiguous memory location. After transposing, the `gpu_gemm` function is called in which the GEMM operation is executed to compute the absolute haplotype '1' states. In this function the transposed and original input matrices are packed, partitioned and transferred to the GPU for kernel execution, as described in section 4.2.2. With the resulting matrix C and the number of sequences N the Pearson correlation coefficients, r_{ij}^2, are computed and stored in the quickLD resulting $m * n$ matrix which is used for the ω-statistic computation.

Besides the computational function, `correlate_gpu`, the GPU initialization and release functions of quickLD are adapted and extended for the ω-statistic GPU computation.

Figure 5.1 shows a general flow chart of the GPU-accelerated OmegaPlus tool in which the two GPU kernels of both quickLD and the ω-statistic computation are shown. The quickLD GPU adaptation will further be referred to as the GEMM-based LD implementation.

5.3 GPU kernels

In this section the two designed kernels will be described in detail. The general idea of the kernel will be elaborated on, as well as acceleration techniques, memory types and memory access patterns used in the kernel. The kernel described in the following section, is designed for a low number of ω-statistic computations and referred to as "Kernel I", where the second kernel is designed for higher number of computations and referred to as "Kernel II".

5.3.1 Kernel I: GPU kernel for low computational load

As previously stated, certain grid positions in the genomic region reveal a low number of SNPs in either the left of right subgenomic region, resulting in a relatively low number of computation steps. This section describes the GPU kernel designed for this situation. Every scheduled work-item for this kernel computes a single step, where the number of scheduled work-items should always be a multiple of the work-group size. This results in a global size of:

$$G_s = \left\lceil \frac{l(W-l)}{L_s} \right\rceil * L_s. \tag{5.1}$$

These G_s scheduled work-items each perform a single computation step, resulting in the minimum number of performed computations on the GPU with a wavefron-

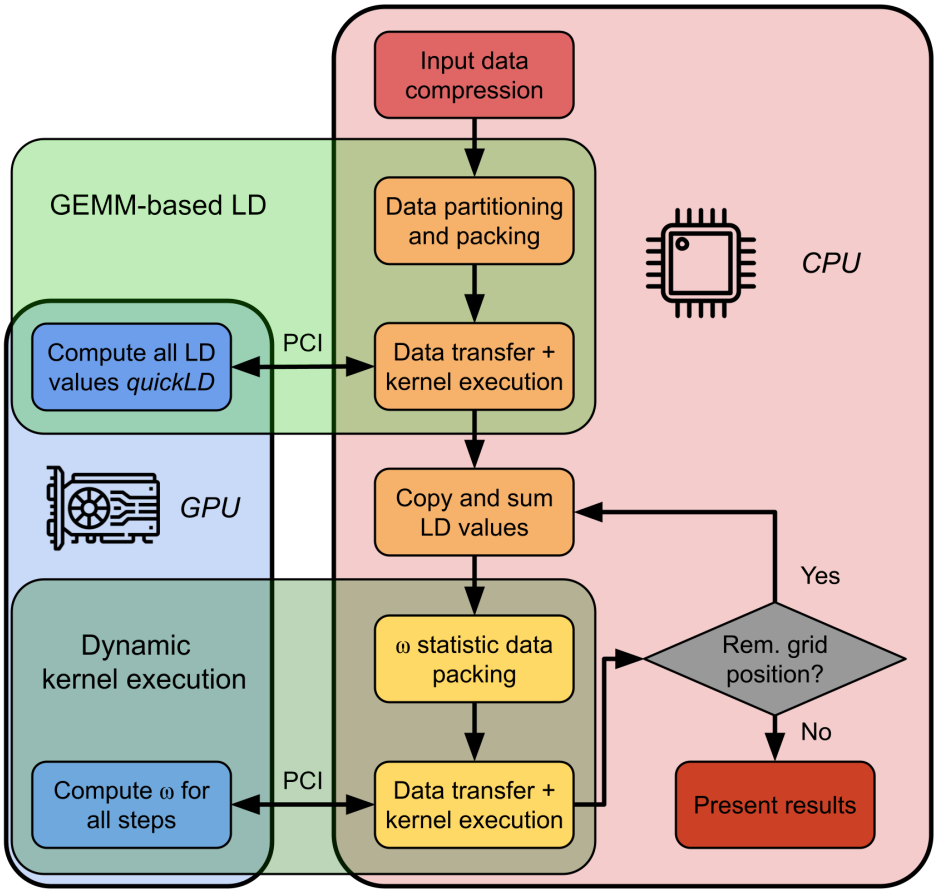


Figure 5.1: Illustration of general design flow chart, where the data is first compressed on the CPU, which after the GEMM-based LD implementation partitions and packs the data before transferring it to the GPU for BLIS kernel execution. After computing all the LD values, data is packed per grid position and transferred to the GPU for ω-statistic kernel execution in which all steps are performed. Depending on the remaining grid positions results are presented or new grid position computations are performed.

t/warp multiple sized work-group. This kernel will be executed when the number of scheduled work-items, and thus needed computation steps, is smaller than:

$$N_{thr} = N_{CU} * W_s * 32, \tag{5.2}$$

with N_{CU} being the number of Compute Units/Streaming Multiprocessors on the GPU and W_s being the wavefront/warp size of the GPU architecture. Recall from section 2.2 that it is good practice to schedule multiple wavefronts/warps per CU/SM, with the optimal being between 8 and 32 wavefronts/warps per CU/SM. This kernel will thus be executing up to the maximum of the optimal occupancy metric. This methodology is optimal for low number of computation steps since the occupancy and hardware utilization is maximized this way. Work-items computing multiple steps would reduce the total number of schedulable work-items resulting in underutilization

of the hardware and longer latencies. With typical GPUs consisting of multiple tens of CUs/SMs and wavefront/warp sizes of 32 to 64, multiple thousand or a couple of ten thousand work-items can be scheduled for this kernel performing the equal number of computation steps. Grid positions with more computation steps require a different approach as higher occupancy does not further improve performance necessarily.

This kernel effectively mimics the behaviour of the OmegaPlus sliding window nested loop, with the parallel approach of each work-item performing a single iteration/computation step. In the original OmegaPlus code the inner loop would take care of the sliding window in the right subgenomic region where the outer loop would slide over the left region. The number of inner loop and outer loop iterations are then equal to $W - l$ and l, the number of SNPs in the right and left sub-region respectively. Recall that grid positions close to either the left of right border of the complete genomic region can exhibit a low number of SNPs in the corresponding subgenomic region. In order to ensure an equal number of outer and inner loops for every kernel execution, the sub-region exhibiting the highest number of SNPs is placed in the inner loop. This dynamic sub-region sliding placement in the nested loop gives the opportunity for the kernel to be specifically designed for a higher number of inner loops and a lower number of outer loops. The number of inner loops, either l or $W-l$ for sub-region L and R respectively, is transferred to the GPU kernel as parameter `in_cnt`. This parameter is used for indexing the kernel buffers.

In order to reduce data transfer overhead and increase overall execution speed the calculation of `ksel2` and `msel2` is performed on the GPU. This results in five remaining variables that need to be transferred to the GPU for ω computation.

To further reduce data transfer overhead, buffers `LS` and `RS` and buffers `k` and `m` are merged into a single floating point buffer and single integer buffer named `LR` and `km` respectively.

Depending on the dynamic sub-region sliding placement, the values of `LS`/`RS` and `k`/`m` are stored in the respective merged buffer with an offset equal to the number of inner loop iterations, in_{cnt}.

All three input buffers, `LR`, `km` and `TS` are stored in global memory on the GPU using the `__global` qualifier. The use of constant memory for the `LR` and `km` buffers, using the `__constant` qualifier, could enable higher kernel optimization with this implementation. This would however, make the maximum allowed size of the sub-regions dependent on the size of the GPUs constant memory, which can be as small as 16kB.

Within the kernel the global work-item ID is first retrieved, using the `get_global_id` function. The index for the outer loop variables is retrieved using the following equation:

$$O_i = \frac{G_i}{in_{cnt}} + in_{cnt}, \tag{5.3}$$

with G_i being global work-item ID. The outer loop index, O_i, will increase by 1 if the global work-item ID is greater than the number of iterations in the inner loop, in_{cnt}. The variable index has an offset equal to the number of iterations in this inner loop.

The index for the inner loop variables is retrieved using the following equation:

$$I_i = G_i \mod in_{cnt}, \tag{5.4}$$

where the inner loop index I_i, will increase with the global work-item ID until it overflows the number of iterations in the inner loop.

The index for variable `TS` is equal to the global work-item ID, G_i.

Figure 5.2 shows how the buffers for kernel I are built up, with the sub-buffers offset, used size depending on the number of SNPs and index variables, I_i and O_i.

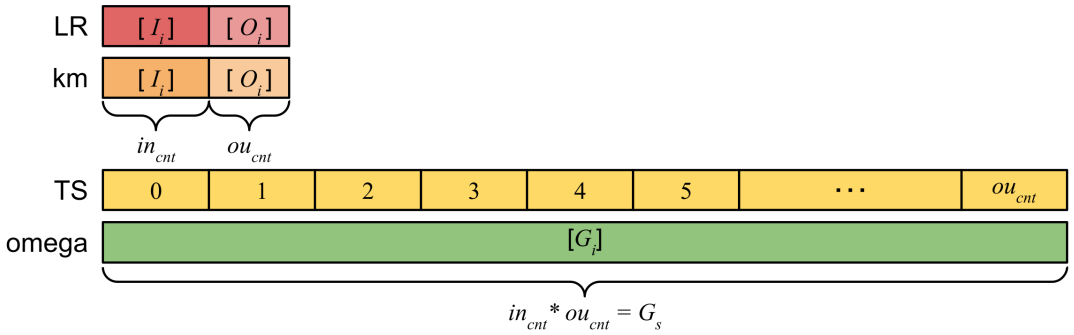


Figure 5.2: Illustration of how the inner and outer loop merged buffers, `LR` and `km`, region correlation values buffer `TS` and output buffer `omega` are built up for kernel I, with their respective sizes and index values. The inner and outer loop sub-buffers in `LR` and `km` are of sizes in_{cnt} and ou_{cnt} respectively and region correlation values buffer `TS` and output buffer `omega` are sized equal to the number of computation steps, or the product of the number of loop iterations.

As stated earlier, this approach mimics and parallelizes the flow of the original nested loop in OmegaPlus, with every work-item performing a single iteration/-computation step and where O_i is updated after the inner loop is finished and I_i is updated every iteration of the inner loop but reset to 0 after finishing. With this approach, G_i can be seen as the current computation step index.

Due to the dynamic sub-region sliding placement in the nested loop, the left and right sub-region values are also dynamically placed in the `LR` and `km` buffers. The sub-region values of the inner loop are placed at the beginning of the buffers, where the sub-region values of the outer loop are placed at an offset of `in_cnt` elements in the buffers. This dynamic value placement is possible due to the fact that the left

and right subgenomic region values are interchangeable in the ω-statistic. This interchangeability allows for a single kernel suitable for both nested loop configurations. In the case of $W - l \ll l$, the inner loop index, I_i, accesses the values of `RS` and `m`, and the outer loop index, O_i, accesses the values of `LS` and `k`. For when $l \ll W - l$, this is the other way around.

Using the global work-item ID, G_i as index value for buffer `TS`, results in a sequential and aligned coalesced memory access for the complete buffer. This is the most optimal access pattern for `TS` as it resides on global memory. Such an optimal access pattern for `TS` is of great importance as it is by far the largest buffer for ω computation, with up to N_{thr} elements.

This kernel is designed to perform the minimum number of computations, which is equal to the number of ω computation steps $l(W - l)$ extended to a multiple of the work-group size, L_s. This requires the minimum number of transferred values to the GPU, no duplicates, no padding. Together with the coalesced memory access approach for `TS` this results in suboptimal memory accesses to both `LR` and `km`.

Inner loop values, indexed with I_i, are largely accessed through a misaligned and therefore uncoalesced memory access. This is due to the fact that the number of inner loops is not a multiple of the wavefront/warp size. Memory accesses with I_i are sequential, aligned and therefore coalesced, for the initial in_{cnt} scheduled work-items. However, when I_i is reset to 0, $G_i \equiv in_{cnt}$, misalignment occurs as shown in fig. 2.12. Due to the misalignment more cache lines are addressed than optimally needed. This approach however, does not degrade performance by a great part due to data reuse in these cache lines [52]. Unused data from a cache line fetched by work-group n can be reused by the adjacent work-group $n + 1$. Furthermore, global work-items $(in_{cnt} * n)$ to $(in_{cnt} * n) + in_{cnt} - 1$ repeatedly access the same memory elements for $n = 0$ to $n = ou_{cnt} - 1$. With ou_{cnt} being the number of outer loop iterations. Whenever the number of work-items in simultaneously executing wavefronts/warps is larger than in_{cnt}, different wavefronts/warps access identical memory elements simultaneously, causing memory access latencies. As modern day GPUs typically consist of dozens of CUs/SMs, each hosting single or multiple wavefronts/warps, the number of simultaneously executing work-items can quickly be over $N_{CU} * W_s = 20 * 64 = 1280$. As the number of SNPs in a subgenomic region, the number of inner loop iterations, is in the same range, latencies can quickly occur. Furthermore due to the misaligned memory accesses, work-groups scheduled on the same CU/SM are barely able to reuse previously read elements by that CU/SM which are stored in shared memory.

A single outer loop value is accessed at an offset of in_{cnt} elements and by in_{cnt} consecutive work-items, where the next in_{cnt} consecutive work-items access the following outer loop values. An access pattern where all work-items in a wavefront

or warp access the same element in global memory requires a single transaction via cache, which after the value is broadcast to all work-items [52]. As the first in_{cnt} work-items access the same element in global memory, all the wavefronts/warps that host these work-items will perform this optimal access. However, due to in_{cnt} not being a multiple of the wavefront/warp size, certain wavefronts/warps will access two different elements from global memory as G_i/in_{cnt} will overflow to the next index.

An obvious solution to these suboptimal memory accesses is to make in_{cnt} a multiple of the work-group size. As this automatically makes in_{cnt} a multiple of W_s and enables work-groups to reuse previously read elements stored on the CU/SM its shared memory. This however, would require buffers `LR`, `km` and `TS` to be padded with dummy values, resulting in higher number of computation steps and higher data transfer overhead. Given the low number of computation steps this kernel is designed for, the advantage of a more optimal memory access pattern is outweighed by the mentioned disadvantages combined with clever data reuse by the GPU.

Due to the fact that every work-item computes a single ω value, these values can be written to the `omega` buffer in a coalesced way. Every global work-item ID, G_i, is used for indexing the current ω value that has been computed and is written to this position. This results in $l(W-l)$, ω values that need to be transferred back to the GPU.

Figure 5.3 shows how the work-groups and work-items access memory and how the massive parallelism is applied to the nested loop and ω computation steps within kernel I.

5.3.2 Kernel II: GPU kernel for high computational load

This section describes the GPU kernel that is suited for grid positions with more than the N_{thr} specified computation steps. As stated earlier kernel I is optimal for grid positions with up to N_{thr} computation steps due to the optimal occupancy limit on the GPU. This kernel is designed to work optimal beyond that number, with a single work-item performing multiple computation steps or ω-statistic computations in order to keep occupancy optimal and output buffer data transfer overhead low.

For this kernel the number of scheduled work-items is set to an indicative value and fluctuates dynamically around this value depending on the number of inner loop iterations in the sliding window nested loop. The global size, G_s or number of work-items indication can be set beforehand using C macros which are present in a C header file. Section 5.4 will elaborate on adjusting this parameter which depends on a set work-group size and number of wavefront/warps per CU/SM. The set number of wavefront/warps per CU/SM should be between 8 and 32 for optimal occupancy and thus performance. Since the kernel described in this section is designed for

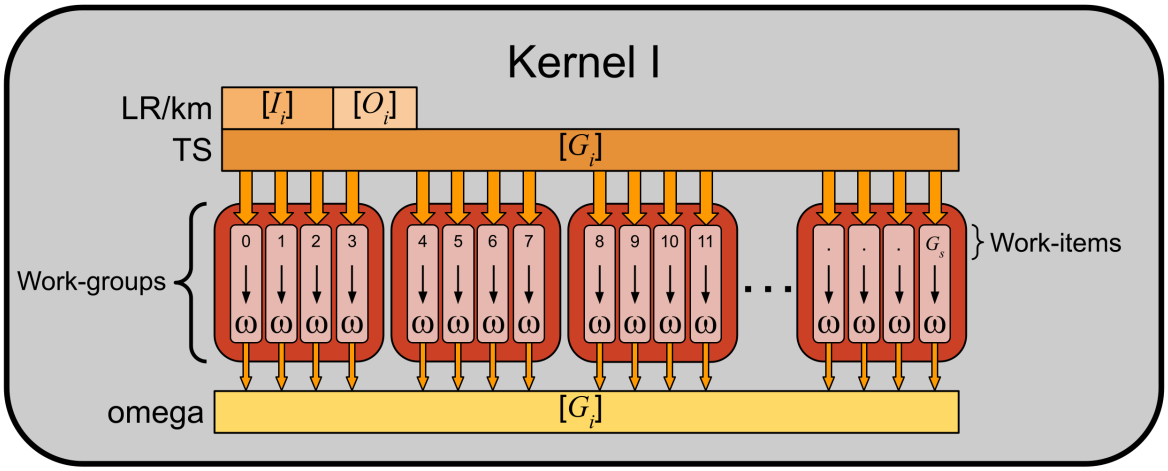


Figure 5.3: Illustration of an abstract view on kernel I. Each of the four work-groups shown host four work-items that can be scheduled on the SPs/CUDA cores in a CU/SM. The figure shows G_s work-items, which is equal to the number of computation steps and ω values that need to be computed for a specific grid position. The figure makes the abstraction that the number of computation steps is an integer multiple of the work-group size, L_s. In reality some padding is applied to ensure this requirement. Every work-item reads the required elements from the sub-buffers in `LR` and `km` and from buffer `TS` to compute a single ω value. The computed ω values together are written to the `omega` buffer in a coalesced way.

computing multiple computation steps per work-item, exceeding 32 wavefront/warps per CU/SM would violate this design requirement since the dynamic kernel transition at N_{thr}, is set to 32 wavefront/warps per CU/SM. The default values for the work-group size and wavefront/warps per CU/SM metric are 128 and 24 respectively as these values are found to perform well.

As the number of scheduled work-items narrowly fluctuates around the set fixed value, the number performed computations steps per work-item is dynamic for every grid position. This variable, the work-item load, is transferred to the kernel as the `wi_load` parameter.

Besides the different approach with regards to the work-item load, the methodology of this kernel is identical to kernel I. The nested loop dynamic sub-region sliding placement is used again, the kernel input buffers are kept the same and how data is stored is kept the same.

The difference with the first kernel, is the implementation of a for-loop that covers the work-item load. The for-loop performs WI_{LD} iterations, with every iteration computing a single ω value. The iterations together cover all the computation steps of a single grid position. A single iteration of the for-loop can be seen as a single execution of kernel I, with identical memory access patterns to the input buffers and ω-statistic computation.

To optimize the for-loop execution, loop unrolling is applied using the OpenCL

`unroll #pragma` derivative. A loop unrolling factor of 4 is applied as it was found to perform the best over the range of tested WI_{LD} values.

In order to ensure the same optimized coalesced memory access pattern to `TS`, a new variable, `G_ic`, is introduced with corresponding value G_{ic}. Within the for-loop G_{ic} is used for indexing the largest buffer `TS`, and can be seen as a virtual global work-item ID that is updated every iteration. The variable is initialized to the global work-item ID, G_i and incremented at the end of the for-loop with the number of scheduled work-items or global size, G_s. This way G_{ic} is equal to the current computation step index as G_i was in kernel I. Using G_{ic} as index value for `TS` results in a sequential, aligned and thus optimal coalesced memory access pattern which is performed WI_{LD} times.

As with `TS` index value G_{ic}, outer loop index value O_i is also updated every for-loop iteration to cover all needed values. Inner loop index value I_i is initialized once outside of the for-loop which results in high data reuse in the for-loop itself. The memory access patterns using O_i and I_i are identical to those in kernel I, but with a more optimal approach which will be described in the following paragraph.

Due to a far higher number of computation steps handled by kernel II, an optimal memory access pattern will have a more profound effect on kernel performance. Within kernel I, memory accesses to `LR` and `km` were either mainly uncoalesced, or suboptimal with single wavefronts/warps accessing different elements in global memory. These suboptimal memory accesses occurred due to the number of inner loop iterations, in_{cnt}, not being a multiple of the wavefront/warp or work-group size. For a far higher number of computation steps the advantage of performing the minimum number of computations on the GPU, and therefore requiring minimum data transfer times, is outweighed by the detrimental effect of suboptimal memory accesses on kernel performance. In order to optimize memory accesses to `LR` and `km`, the inner loop sub-buffers in `LR` and `km` of size in_{cnt}, are padded with dummy values to a multiple of the work-group size L_s. The size of the sub-buffers are then:

$$in_{cntpad} = \left\lceil \frac{in_{cnt}}{L_s} \right\rceil * L_s, \tag{5.5}$$

with in_{cntpad} automatically being a multiple of the wavefront/warp size since L_s is a multiple of W_s. This ensures solely coalesced accesses using the inner loop index variable I_i and solely wavefront/warp single element optimized accesses using the outer loop index variable O_i.

The outer loop sub-buffers in `LR` and `km`, at offset in_{cntpad} and size being equal to the number of outer loop iterations, ou_{cnt}, are padded with dummy values up to the rounded up division of the total number of computation steps performed and in_{cntpad}:

$$ou_{cntpad} = \left\lceil \frac{WI_{LD} * G_s}{in_{cntpad}} \right\rceil, \tag{5.6}$$

which results in the minimum number of needed outer loop variables to ensure that index variable O_i does not overflow buffers `LR` and `km`.

The size of the `LR` and `km` buffers are thus equal to $in_{cntpad} + ou_{cntpad}$. The size, the sub-buffers offset, the index values and grey indicated dummy values padding of `LR` and `km` are is shown in fig. 5.4.

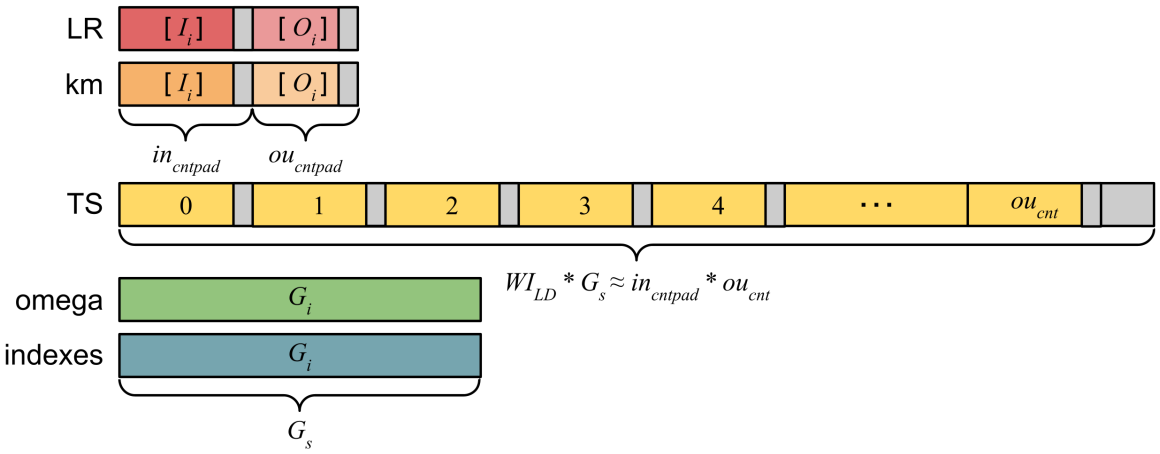


Figure 5.4: Illustration of how the inner and outer loop merged buffers, `LR` and `km`, region correlation values buffer `TS` and output buffers `omega` and `indexes` are built up for kernel II, with their respective sizes, index values and padding. The inner and outer loop sub-buffers in `LR` and `km` are of sizes in_{cntpad} and ou_{cntpad} respectively where the grey area represents the padded dummy values. Region correlation values buffer `TS` is sized equal to the product of the work-item load, WI_{LD}, and the set number of work-items, G_s, which is approximately the product of the number of outer loop iterations and the padded number of inner loop iterations. The output buffers `omega` and `indexes` are the same size as the number of set work-items as every work-items produces a single maximum ω value and corresponding index.

Due to the fact in_{cntpad} is a multiple of the work-group size and thus wavefront/warp size, the access pattern using O_i has become more optimized for the global memories `LR` and `km`. Unlike kernel I, every wavefront/warp accesses a single element from global memory resulting in a single transaction that is broadcast to all work-items in that wavefront/warp [52]. However, the access pattern also causes in_{cntpad} consecutive work-items to access the same global memory element. These latencies can be hidden by scheduling multiple work-groups and thus wavefronts/warps on every CU/SM.

A second optimization is applied to the access pattern using I_i, which focuses on data reuse by work-items and CUs/SMs. Despite the optimization with regards to using in_{cntpad} instead of in_{cnt}, work-items and therefore work-groups still repeatedly access the same memory elements as was the case in kernel I. This property of the

access pattern is exploited to maximize data reuse. This is achieved by setting the number of scheduled work-items or global size, G_s, initially set to the indicative value, to the nearest integer multiple of in_{cntpad}. The number of scheduled work-items thus fluctuates around the indicative value set by the user for every grid position. G_s being an integer multiple of in_{cntpad} ensures that every for-loop iteration requires the same inner loop memory elements, while working over the outer loop and `TS` memory elements. This is due to the fact that in every for-loop iteration the 'virtual' global work-item ID G_{ic} is incremented with G_s. Since the inner loop access pattern, identical to kernel I, using I_i, is determined as follows:

$$I_i = G_{ic} \mod in_{cntpad}, \tag{5.7}$$

I_i remains the same value every for-loop iteration as G_{ic} is incremented with a value which is a multiple of in_{cntpad}. This enables every scheduled work-item to require a single inner loop value from both `LR` and `km`, which are solely accessed through a coalesced pattern by the work-group on the CU/SM.

Given G_s is an integer multiple of in_{cntpad}, the calculation of index variable O_i in every for-loop iteration can be simplified with an additional initialization. Since the outer loop access pattern, identical to kernel I, using O_i, is determined as follows:

$$O_i = \frac{G_{ic}}{in_{cntpad}} + in_{cntpad}, \tag{5.8}$$

the calculation of O_i can be simplified to a single constant value increment at the end of the for-loop. This is due to the fact that G_{ic} is incremented every for-loop iteration with a value which is a multiple of in_{cntpad}. This makes $\frac{G_{ic}}{in_{cntpad}}$ a constant, which is stored in variable `O_inc` with value O_{inc} and used to increment the initialized value of $O_i = \frac{G_i}{in_{cntpad}} + in_{cntpad}$.

As with kernel I, the memory access pattern using inner loop index I_i, still indicates a repeated access pattern over the in_{cntpad} sub-buffer elements if $G_s > in_{cntpad}$. As stated earlier, on current GPUs the number of work-items in simultaneously executing wavefronts/warps is regularly larger than typical number of inner loop iterations, in_{cntpad} in this kernel. Given a GPU consisting of N_{CU} CUs/SMs, which can each execute a L_s sized work-group simultaneously, the number of CUs/SMs that need to access the same L_s memory elements is:

$$N = N_{CU} - \frac{in_{cntpad}}{L_s}. \tag{5.9}$$

Due to the repeated access pattern using I_i, the additional scheduled work-groups on the 'overlapping' CUs/SMs can't be used to hide memory latencies as they require the same data. However, due to the previously mentioned optimization,

work-items require a single inner loop value from both LR and km. This enables massive data reuse after every CU/SM has read the L_s elements from both LR and km for the initial N_{CU} scheduled work-groups. Work-group range:

$$n * \frac{in_{cntpad}}{L_s} \texttt{ to } (n+1) * \frac{in_{cntpad}}{L_s} - 1 \tag{5.10}$$

repeatedly access the same elements from LR and km inner loop sub-buffers for $n = 0$ to $n = \frac{G_s}{in_{cntpad}}$. Work-groups requiring the same data can easily be scheduled on the CUs/SMs that fetched the exact same data to shared memory, for the initial N_{CU} scheduled work-groups.

The access pattern using outer loop index value O_i, results in all work-groups in the following range:

$$n * \frac{in_{cntpad}}{L_s} \texttt{ to } (n+1) * \frac{in_{cntpad}}{L_s} - 1 \tag{5.11}$$

to access the same single element from both LR and km, for $n = 0$ to $n = \frac{G_s}{in_{cntpad}}$. In order to prevent high memory latencies, work-groups should be scheduled and executed in such a way that simultaneously executing work-groups are from another n range. Scheduling multiple work-groups on every CU/SM enables this. Whenever a work-group memory instruction stalls due to that element being accessed by a simultaneously executing work-group, another work-group queued on that CU/SM can take over execution and hide that high memory latency i.e., optimal occupancy.

Because of the inner loop sub-buffers padded dummy values in LR and km, dummy values are also inserted in TS to ensure that the correct elements are read from TS every iteration, using a coalesced memory access pattern. The size of TS is equal to $WI_{LD} * G_s$ due to every work-item performing WI_{LD} iterations. After every in_{cnt} elements, TS consists of dummy values up to in_{cntpad} elements. The buffer consists of the number of outer loop iterations, ou_{cnt}, of these padded sequences which after the buffer is padded with dummy values from $ou_{cnt} * in_{cntpad}$ to $WI_{LD} * G_s$. Figure 5.4 shows the size, the index values and the grey indicated dummy values padding of TS.

Due to the fact that multiple ω values are computed in a for-loop an additional variable is needed to keep track of the up to the current loop maximum value. A conditional statement checks if the current computed ω value tmpW is larger than the stored maximum value maxW, which is updated if the statement is true.

Due to the coalesced write to the omega buffer and work-items computing a single ω value in kernel I, the array index of a value in omega is the same as the corresponding computation step index, and can therefore be used to find the region that ω value is found in. Since a single work-item performs multiple computation steps or ω-statistic computations in kernel II, the array index of a found maximum ω value

no longer reveals any information about the region of that value. In order to find the region of the work-item found maximum ω value in `omega`, the current computation step of that value is stored in a second buffer called `indexes`. Variable `G_ic` holds the current computation step index which is stored in variable `maxI` when the above mentioned conditional statement is true.

After the for-loop is finished, the found maximum ω value and the corresponding computation step index are written in a coalesced access pattern to buffers `omega` and `indexes` respectively.

Figure 5.5 shows how the work-groups and work-items access memory and how the massive parallelism is applied to the nested loop and ω computation steps within kernel II.

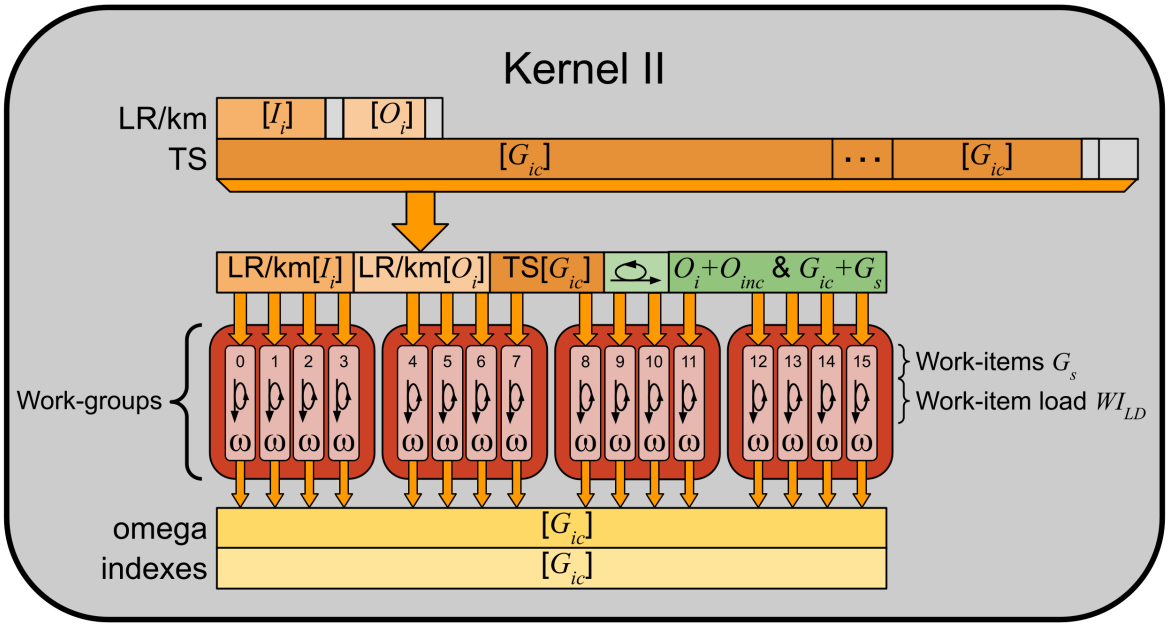


Figure 5.5: Illustration of an abstract view on kernel II. Each of the four work-groups shown host four work-items that can be scheduled on the SPs/CUDA cores in a CU/SM. 15 work-items are scheduled which can be seen as the set number of work-items G_s, which is near the indicative value set by the user. The dummy values padding of the input buffers is shown, where `TS` is made up of WI_{LD} sections of the global size, G_s. The additional buffer shown above the work-groups, indicates which elements are read to perform the work-item load in multiple iteration. The green part in this additional buffer indicates the memory access pattern increment for every work-item load iteration. Every work-item reads a single value from `LR` and `km` using the inner loop index value I_i, and WI_{LD} values from `LR`, `km` and `TS` using outer loop index value, O_i, and global index value, G_{ic}, respectively. Every work-item computes WI_{LD}, ω values, of which the maximum and its global index are written in a coalesced way to the `omega` and `indexes` buffers respectively.

5.4 Host and interfacing

In this section the host and interfacing code will be described. The preparing of the data for each of the kernels will be elaborated on, as well as some small optimizations to speed up this process.

An introduction will be given about OpenCL functions used for device information requests which are needed for optimal and dynamic kernel execution. Besides that this section will elaborate on some general variable calculation required by both kernel designs.

In the `gpu_init` function, the target GPU is selected, the kernels are compiled, the GPU buffers are allocated and the device specific specifications are requested using OpenCL functions.

In order to work with the occupancy metric, the number of CUs/SMs and wavefront/warp size on the GPU in use, should be known.

The `clGetDeviceInfo` function can be used to request the number of CUs/SMs on the GPU.

The `clGetKernelWorkGroupInfo` function can be used for requesting the wavefront/warp size and the maximum possible work-group size that can be scheduled on a CU/SM.

In order to know what size a single buffer or all buffers combined can be, the maximum buffer allocation size and global memory size are requested. These are also requested using the `clGetDeviceInfo` function.

Within `computeOmegaValues_gpu`, the GPU-accelerated ω-statistic computation function, some variable initilization is performed outside of the dynamic kernel conditional statement.

For every designed kernel, the number of SNPs in the left and right subgenomic region are calculated in order to know the number of iterations per loop. The number of SNPs in the left sub-region for a specific grid position, is the difference of the leftmost SNP position and rightmost SNP position in this sub-region. This is the other way around for the number of SNPs in the right sub-region. The number of SNPs in the left and right subgenomic regions, l and $W-l$, with W being the number of SNPs in the complete region, are stored in variables `L_SNP` and `R_SNP` respectively.

The total number of computation steps, $l(W-l)$, of a grid position is stored in `tot_step`. The dynamic kernel execution is implemented using a conditional statement that checks if the number of computation steps is larger than the specified threshold, `steps_thresh`.

Outside of the dynamic kernel conditional statement body, the found maximum ω value and the corresponding region borders are stored in the `omega` structure with the current grid position index. At the end of the `computeOmegaValues_gpu` function

the initialized pointers for the GPU buffers are freed for subsequent function calls, i.e., subsequent grid positions.

The original nested for-loop of OmegaPlus features an approximation for the ω-statistic computation per ω grid position using a variable named `borderTol`. This approximation effectively limits the maximum difference between the number of SNPs in sub-regions sliding windows. This balances the number of SNPs in the left and right sub-region sliding windows. This implementation can modify the grid position dependable borders of the right subgenomic region during nested loop execution. This would produce varying number of SNPs in the assessed genomic region resulting in buffer size changes from computation step to computation step making optimal buffer index patterns impossible. Therefore, it has been chosen to not support this implementation for the GPU-accelerated version of OmegaPlus.

5.4.1 Host implementation for kernel I

This section will describe the data preparation, GPU interfacing including data transfers and kernel execution, and result processing for kernel I.

Data preparation Kernel I requires 3 input buffers `LR`, `km` and `TS` and a single output buffer, `omega`. Floating point buffer `LR` and integer buffer `km` are of size `tot_SNP` which is the sum of `L_SNP` and `R_SNP`. Floating point buffers `TS` and `omega` are of size `tot_step`. These buffers are initialized as pointers using `malloc`.

As with the dynamic kernel execution, the dynamic sub-region sliding placement is implemented with a conditional statement, where the inner loop of the nested loop always contains the most iterations. The two nested loops variations, $l > W - l$ or $W - l > l$, are both fully written out in code instead of using a single nested loop with dependable variables. This increases code size but benefits code readability and performance.

The original nested for-loop of OmegaPlus is adopted, where instead of storing the parameters of the ω-statistic in variables, the parameters are stored in the initialized pointers/arrays eponymous to the kernel buffers. As previously stated, correlation values originally stored in `LS` and `RS` are stored in array `LR`, and window dependable number of SNPs originally stored in `k` and `m` are stored in array `km`. Left sub-region values `LS` and `k` are stored using left region index variable `L_i`. Right sub-region values `RS` and `m` are stored using right region index variable `R_i`. A third index variable, `T_i`, is used to store correlation values in array `TS`, where the index variable is incremented every computation step at the end of the inner loop.

Index variable of the outer loop placed region, either `L_i` or `R_i`, is initialized to the number of SNPs in the inner loop placed region to ensure the correct sub-buffer

offset. At the end of the outer loop this index variable is incremented.

Due to the fact that the `borderTol` approximation implementation is discarded from the nested loop, only one nested loop completion is required to loop over and store all inner loop values using either `L_i` or `R_i`. This is implemented through a conditional statement in the inner loop, that checks if the outer loop index variable is equal to the initialized offset, i.e., the first nested loop call.

GPU interfacing After populating all the GPU kernel input buffers, the Graphics Processing Unit (GPU) interfacing function for kernel I, `computeOmega_gpu1`, is called with the correct offset value, either `L_SNP` or `R_SNP`. This function call is still in the body of the conditional statement that takes care of the dynamic sub-region placement.

Within the function the offset value or number of inner loop iterations is hold by `in_cnt`. This variable is set as kernel argument using `clSetKernelArg`.

The local and global kernel execution values are determined and stored in `local` and `global` respectively, where `local` is set to the maximum supported work-group size and `global` to the nearest multiple of `local` and the number of computation steps `tot_step`.

The complete input buffers `LR`, `km` and `TS` are written to the GPU using the `clEnqueueWriteBuffer` function. The writes are issued to the `io_queue` OpenCL command queue as non-blocking. The last issued write is additionally stored in the OpenCL event wait list, `events`. By merging the buffers, less write events are issued which marginally reduces data transfer overhead.

After the writes are issued and added to the command queue, the kernel launch can be issued. The kernel launch is issued to the same `io_queue` command queue using the `clEnqueueNDRangeKernel` function. The kernel launch event is set to wait on the last write event to finish by using the event wait list. Kernel I is launched by using the OpenCL kernel variable `omega_kernel1`, and with `global` work-size and `local` work-group size. The kernel launch event is also stored in the OpenCL event wait list.

The OpenCL `clEnqueueReadBuffer` function is used to read back the output buffer, textttomega. This read is also issued to the `io_queue` command queue and is set to wait on the kernel launch event to finish.

At the end of the function, the `clFinish` function is called with the `io_queue` command queue as argument. This ensures that every event issued to this command queue is finished.

Result processing After finishing the GPU interfacing, a for-loop iterates over the computed ω values in the `omega` buffer to find the maximum value and its corresponding index.

After the for-loop is finished, the region borders where the maximum ω value is found in are calculated. The region borders are the leftmost and rightmost SNP within the window ω value was found in. In the original OmegaPlus sliding window nested loop, the current window border positions are known during ω computation. Since the GPU-accelerated version computes all the values on the GPU using buffers, only the computation step index of a computed ω value is available by looking at the `omega` array index. The positions of the corresponding window borders can be retrieved with the whole grid position region borders. The leftmost border position, `i` in the original loop, is retrieved by subtracting the outer loop index of the found ω value, O_i, from the starting position of the left window, `leftMinIndex`. The rightmost border position, `j` in the original loop, is retrieved by adding the inner loop index of the found ω value, I_i, to the starting position of the right window, `rightMinIndex`. O_i and I_i are retrieved as in kernel I, by performing the same calculations on the computation step index but without adding the offset to O_i. The eventual leftmost and rightmost SNP positions of the maximum value are retrieved by adding the leftmost SNP position of the complete grid position region.

Figure 5.2 shows how the buffers for kernel I are built up, with the sub-buffers offset and used size depending on the number of SNPs.

5.4.2 Host implementation for kernel II

This section will describe the data preparation, GPU interfacing including data transfers and kernel execution, and result processing for kernel II.

Data preparation For kernel II the global size, G_s or number of work-items indication can be set beforehand using C macros which are present in a C header file. This indicative value depends on the set work-group size and the set number of wavefront/warps per CU/SM. The default values for the work-group size and wavefront/warps per CU/SM metric are 128 and 24 respectively as these values are found to perform well with the tested GPUs. This can differ is newer or older architectures differ from current architectures. The values can be edited in the `OmegaPlus.h` header file where the relevant variables can be found at the end after the `GPU setting` comment. The work-group size variable is called `GPU_GROUP_SIZE` and the wavefront/warps per CU/SM metric variable is called `WAVE_CU`.

Kernel II requires the same 3 input buffers `LR`, `km` and `TS`, the same output buffer, `omega` and an additional integer output buffer, `indexes`. The buffers are initialized the same using `malloc` but have different sizes from kernel I as padding is applied.

The dynamic sub-region sliding placement implementation is identical to the one for kernel I, with the two nested loops variations, $l > W - l$ or $W - l > l$, both fully

implemented in code to retain readability and performance.

The two nested loop implementations are also identical to the ones for kernel I, with an additional increment for the `T_i` index variable due to the dummy values padding.

As explained in the section of kernel II, section 5.3, the inner and outer loop sub-buffers in `LR` and `km` are padded with dummy values for memory access optimizations and the number of scheduled work-items is adapted from the indicative value for data reuse optimization. With these values the work-item load and total number of computation steps are calculated, which are further used for the kernel execution.

First the the number of SNPs in the inner loop placed region or inner loop iterations is incremented to a multiple of the work-group size. This value is placed in either `R_SNP_pad` or `L_SNP_pad`, depending on the dynamic sub-region sliding placement. The incremented number is used for initializing either `L_i` or `R_i` in order to ensure the correct sub-buffer offset.

The number of scheduled work-items or global work-size, G_s, is then set to the nearest integer multiple of the incremented number of inner loop iterations and the indicative number of scheduled work-items, and stored in integer `set_wi`.

With the incremented number of inner loop iterations, stored in either `R_SNP_pad` or `L_SNP_pad`, the minimum number of computation steps needed for covering all correlation values in `TS` is calculated by multiplying the actual number of SNPs/iterations in the outer loop with the incremented number of SNPs/iterations in the inner loop. This value is stored in `tot_step_pad`.

Now the work-item load can be calculated by dividing the minimum number of computation steps, `tot_step_pad`, by the set number of scheduled work-items, `set_wi`, and rounding this number up to the nearest integer. The work-item load is stored in `wi_load` and is an additional kernel parameter besides the number of inner loop iterations.

The number of computation steps in `tot_step_pad`, is then updated to the value of the work-item load, `wi_load`, multiplied with the set number of scheduled work-items, `set_wi`. This gives the actual number of computation steps that are performed by the kernel. This is the minimum number of steps needed as the work-item load itself is calculated using the actual number of SNPs in the outer loop placed region.

Given the actual number of computation steps and the incremented number of inner loop iterations, the actual number of outer loop iterations is calculated. The actual number of outer loop iterations that will be performed is the round up division of the actual number of computation steps, `tot_step_pad`, by the incremented number of inner loop iterations, either `R_SNP_pad` or `L_SNP_pad`.

The adapted number of loop iterations/SNPs of both regions are then added and stored in `tot_SNP_pad`.

The size of the floating point buffer LR and integer buffer km is set to the total number of inner and outer loop values that are going to be accessed in the kernel, tot_SNP_pad. The size of floating point buffer TS is set to the actual number of computation steps that will be performed by the kernel, tot_step_pad. The sizes of the floating point output buffer omega, and integer output buffer indexes are set to the set number of scheduled work-items, set_wi, that all return a single a single ω and corresponding computation step index value.

As previously stated, the T_i index variable is incremented at an additional location in the nested loop in order to account for the dummy values padding in TS. At the end of the outer loop, besides incrementing the outer loop index variable, either Rm_i or Lk_i, T_i is incremented with the difference of the actual number of SNPs in the inner loop placed region and the incremented number of SNPs in that region. This ensures that the inner loop skips over the padded dummy values inserted in TS from R_SNP/L_SNP to R_SNP_pad or L_SNP_pad.

The dummy values in the different buffers are added through a series of additional for-loops. The left and right sub-region correlation values are set to 0.0, the left and right sub-region window SNP count values are set to 2 and the complete region correlation values are set to FLT_MAX, which is the maximum value a floating point number can be. These values are chosen to ensure ω values computed with the dummy values are either zero or extremely small. Work-items iterating over the dummy values will store the correct found maximum ω value instead of a outliers computed with padded dummy values.

GPU interfacing After finishing the nested loop and dummy values for-loops the GPU interfacing function, computeOmega_gpu2, is called with the correct incremented offset value, either R_SNP_pad or L_SNP_pad depending on the nested loop configuration.

Within the function the incremented offset value or number of inner loop iterations is hold by in_cnt_pad. The offset value, together with the work-item load, wi_load, are set as kernel arguments using clSetKernelArg.

The approach of queuing the writes and reads to the GPU is identical to the one in computeOmega_gpu1 with an additional read event at the end of the function for the indexes buffer. The only difference is the sizes of the buffers, which are set to the corresponding initialized sizes.

The queuing of the kernel launch is also identical, but with omega_kernel2 as kernel variable, set_wi as global work-size stored in global and C macro set work-group size stored in local.

Given the fact the same buffers are used as with kernel I, with the additional indexes buffer, the writing and reading of these buffers is identical to the approach

in computeOmega_gpu1. The indexes buffer is read after the omega buffer and before clFinish is called with the same OpenCL command queue, io_queue, as argument.

Result processing After finishing the GPU interfacing, a for-loop iterates over the computed ω values by the set_wi number of work-items in the omega buffer to find the maximum value. The array index of the maximum value is stored in integer variable indexes, which is used to retrieve the computation step index of that value from the indexes buffer. The calculation of the leftmost and rightmost SNP position of the maximum value is then identical to the one performed for kernel I, using the computation step index from indexes.

Figure 5.4 shows how the buffers for kernel II are built up, with the sub-buffers offset and used size depending on the number of SNPs. The grey areas indicate the padded dummy values needed for optimal memory access patterns.

5.5 Overview

Two kernels are designed for the GPU-accelerated version of OmegaPlus. The kernels are scheduled dynamically depending on the number of computation steps per grid position, and the number of Stream Processors (SPs) or CUDA cores available on the GPU.

For both kernels, correlation values LS and RS, and sub-region SNP count values k and m are merged together in two buffers to reduce data transfer overhead. These buffers are placed in global memory in order to enable optimized coalesced memory accesses and optimized wavefront/warp single element accesses.

Both implementations also apply a dynamic sub-region sliding placement in the nested loop to allow for a more specifically optimized memory access pattern.

Kernel I performs the minimum number of computation steps and applies a rather naive approach of mimicking the OmegaPlus sliding window nested loop approach. A single scheduled work-item performs a single computation step of the nested loop by computing an ω value. All the work-items together complete all the nested loop computation steps or iterations with additional steps to ensure the global work-size, G_s is an integer multiple of the work-group or local size, L_s. The kernel applies a full coalesced memory access pattern to correlation value buffer TS. Memory accesses to both LR and km are unoptimzed, but due to the low number of computation steps this kernel needs to perform, this approach is found to be more optimal than using padded buffers. As every work-item computes a single ω value, the returned omega buffer has the same size as the global work-size which approximately the number of computation steps of a grid position.

Kernel II applies the same sliding windows nested loop approach, but with every work-item performing multiple computation steps, the work-item load. This enables the kernel to be scheduled with an optimal number of work-items with regards to occupancy. The addition of padded dummy values in the `LR` and `km` sub-buffers, enables the kernel to perform a full coalesced, and optimized wavefront/warp single element accesses to these buffers while also increasing data reuse massively. The full coalesced memory access pattern to the largest buffer `TS` is preserved. The padded dummy values require additional dummy computation steps, but the found performance gain outweighs this disadvantage. As every work-item computes multiple ω values, an additional buffer is introduced that keeps track of the computation step of the found maximum value. Both of these output buffers, `omega` and `indexes`, are the size of the global work-size. This is substantially smaller than the number of computation steps this kernel is designed to perform, which reduces data transfer overhead.

Both the host C-code, and OpenCL GPU kernels can be found at `github.com/MrKzn/omegaplus.git`. Examples of how to compile the GPU-accelerated version of OmegaPlus and how to perform a test run can be found in the `README.md` file shown on the main page.

Chapter 6

Performance evaluation

This chapter will present an evaluation of the designed kernels as well as an performance evaluation of the kernels and the GPU-accelerated OmegaPlus tool as a whole. First the experimental setup will be described in section 6.1. This section will elaborate on the datasets used for evaluation, with different number of sequences/samples and SNPs, and on the platforms used for testing. Section 6.2 will describe the correctness evaluation of the kernels and the corresponding host code. In section 6.3 the performance evaluation will be presented in which the experimental setup is used to gather performance values to compare to the original tool.

6.1 Experimental setup

In order to get an insight into the execution time, throughput and performance scaling of the designed kernels, datasets with different number of sequences/samples and different number of SNPs will be used. These will be simulated datasets with either, fixed number of sequences/samples and varying number of SNPs, or fixed number of SNPs and varying number of sequences/samples.

To get a good insight in the performance of the complete GPU-accelerated tool and either of the GPU-accelerated compute intensive parts, LD and ω-statistic computation, three different dataset ranges are used. The datasets are used to compare execution times and throughput values of the GPU-accelerated tool and its parts to their CPU counterparts. This will give an insight in the performance gain with respect to the original sequential OmegaPlus version.

The simulated datasets are generated using ms (Hudson, [131]), which can be found at `home.uchicago.edu/~rhudson1/source/mksamples.html`. The datasets are generated using the following command:

`./ms nsam nreps -s nsites`,

which generates `nsam` sequences/samples, `nsites` SNPs/sites with `nreps` alignments. All the generated datasets consist of a single alignment.

For performance evaluation of the LD computation, datasets are used with varying number of SNPs and varying number of sequences/samples as the computational intensity of the LD computation depends on both of these factors.

For performance evaluation of the ω-statistic computation, a single dataset range is used with varying number of SNPs as the computational intensity of the ω-statistic computation only depends on the number of SNPs.

The dataset ranges used to evaluate the performance of the GPU-accelerated LD computation consists of either 10,000 sequences/samples and varying number of SNPs, from 1,000 to 10,000 with an interval of 1,000 SNPs, or 5,000 SNPs with varying number of sequences/samples from 10,000 to 100,000 with an interval of 10,000 sequences/samples. These dataset ranges give a good representation of execution times for high and low number of SNPs and high and low number of sequences/samples with executing times high enough to get a good insight in performance gain.

The dataset range used to evaluate the performance of the GPU-accelerated ω-statistic computation consists of 50 sequences/samples and varying number of SNPs, from 1,000 to 20,000 with an initial interval of 1,000 SNPs to 10,000 SNPs and the additional 20,000 SNP dataset. As the number of sequences/samples does not affect the computational intensity for the ω-statistic computation, these are kept to the minimum to reduce overall execution times. These dataset configurations give a good representation of execution times for high and low number of SNPs with executing times high enough to get a good insight in performance gain.

For the performance evaluation of the complete GPU-accelerated OmegaPlus tool the same datasets are used as for the LD computation evaluation as these datasets with varying number of SNPs and varying number of sequences/samples, put high and low load on both compute intensive parts and therefore also on the complete tool.

All the tests are performed with OmegaPlus set to compute the ω-statistic at 1,000 equidistant location with the left, L, and right, R sub-region windows set to a maximum of 20,000 SNPs and minimum of 1,000 SNPs and the alignment length set to 100,000.

```
./OmegaPlus -grid 1000 -length 100000 -minwin 1000 -maxwin 20000
```

Two platforms used for testing are shown in table 6.1

	System I	**System II**
Description	off-the-shelf laptop	Google Colab
CPU Model	AMD A10-5757M	Intel Xeon E5-2699 v3
Base Freq.	2.5 GHz	2.3 GHz
Processors	1	1
Cores/Processor	4	2*
Threads/Processor	1	1*
GPU Model	Radeon HD8750M	NVIDIA Tesla K80
Internal GPUs	1	1*
Compute Units	6	13
Streaming Processors	384	2496
GPU Memory	2 GB	11 GB

Table 6.1: Overview of the platform specifications (the corresponding device has more cores/threads available but the number is restricted within Google Colaboratory).*

6.2 GPU kernels verification

This section describes the approach of verifying the kernel produced ω-statistic values for the sake of correctness. This is achieved by adding test code to the GPU-accelerated host code that compares ω-statistic values computed on the GPU to values computed on the CPU.

Due to the fact that GEMM-based LD implementation applies a different order in computing the floating point correlation values than OmegaPlus does, the resulting values can differ. From verification it is found that the resulting ω-statistic values can differ by at most one thousandth, 0.001. Given the GEMM-based LD correlation values can differ from those computed by OmegaPlus, resulting in ω-statistic values differences (≤ 0.001), the GEMM-based LD implementation is not used for verifying the ω-statistic GPU kernels. By not using the GEMM-based LD values, the GPU-accelerated ω-statistic computation and sequential CPU ω-statistic computation are performed using the exact same input values enabling correct verification and error measurements.

6.2.1 Kernel I

As kernel I writes back all the computed ω-statistic values for a specific grid position, every value can be compared for correctness on the CPU. This is performed by computing the ω values of every computation step as in the original tool, and storing these values in an additional test array. After executing the nested loop and the

complete GPU interfacing, the kernel computed ω values in `omegas` can be compared to the CPU computed values for correctness.

The result of floating point calculations is sensitive to different approaches and different architectures, which can be the case with CPUs and GPUs [140]. The resulting ω-statistic values are thus verified taking a relative error into account that is compared to a maximum allowed relative error:

$$\left|\frac{\omega - \omega_{test}}{\omega_{test}}\right| < 0.00001. \tag{6.1}$$

If the error limit is exceeded the values are printed to indicate the faulty answer.

The resulting grid positions maximum ω-statistic values in the OmegaPlus "Report" of both the sequential and the GPU-accelerated versions are compared to each other to get an insight in the average and maximum absolute error in the found maximum ω values. For this verification the absolute error is used in order to get a good insight in the actual found error values. Two datasets have been tested which both consist of 50 sequences/samples, and one with 1,000 SNPs and one with 7,000 SNPs. Table 6.2 shows the absolute errors for the found ω-statistic values in the report. In the dataset with only 1,000 SNPs a relatively high maximum ω value is found, where as in the dataset with 7,000 SNPs a much lower maximum ω value is found. These two cases give a good insight in the kernel its correctness.

	7,000 SNPs	**1,000 SNPs**
Avg. abs. err.	9.0E-08	4.5E-07
Max. abs. err.	1.0E-06	7.8E-05
Max. ω	1.534736	96.486229
Max. ω err.	0.0	1.5E-05

Table 6.2: Kernel I average, total maximum and found ω maximum absolute ω-statistic errors for two datasets consisting of 50 sequences/samples and either 1,000 or 7,000 SNPs.

As expected, the table shows a bigger error with the larger found maximum ω value in the 1,000 SNP dataset. But even with the relatively high found maximum ω value, the maximum absolute error is still acceptably low.

6.2.2 Kernel II

Unlike kernel I, not all computed ω-statistic values of a specific grid position are written back to the CPU within kernel II. Due to this property it is much more challenging to verify the correctness of the complete kernel. In order to verify correctness of the kernel, only the values that are written back are compared to values computed on the CPU. This is performed by essentially mimicking the GPU kernel on the CPU. In

this approach the same buffers are used as those transferred to the GPU. A nested loop is used to mimic the scheduled work-items, G_s, in the outer loop and mimic the work-item load, WI_{LD} in the inner loop. The index variables, O_i, I_i and G_{ic} are determined in the outer loop identical to the work-item approach in the kernel, and the work-item load is performed in the inner loop as in the kernel its for-loop. The computed work-item maximum values are stored in an additional test array, and verified in the same way as with kernel I after finishing both the GPU interfacing and CPU kernel mimic.

This approach requires the kernel and thus the CPU version of the kernel to already work correct with respect to data accessing. In order to verify correct data accessing and verify ω values correctness, the same verification approach is used as with kernel I. The grid positions maximum values in the OmegaPlus "Report" of both implementations are compared to each other to gather absolute error values. Table 6.3 shows the absolute errors for the found ω-statistic values in the report.

	7,000 SNPs	**1,000 SNPs**
Avg. abs. err.	9.0E-08	4.5E-07
Max. abs. err.	1.0E-06	7.8E-05
Max. ω	1.534736	96.486229
Max. ω err.	0.0	1.5E-05

Table 6.3: Kernel II average, total maximum and found ω maximum absolute ω-statistic errors for two datasets consisting of 50 sequences/samples and either 1,000 or 7,000 SNPs.

Given the order of all the calculations in the ω-statistic in kernel II are kept identical to kernel I, the resulting errors are also identical.

6.3 Performance comparisons

In this section a thorough performance evaluation will be presented of both GPU-accelerated compute intensive parts separately and the complete GPU-accelerated OmegaPlus tool.

6.3.1 Omega statistic performance

In this section performance values will be presented of the GPU-accelerated ω-statistic computation with multiple graphs presenting different views on the performance.

Both the CPU and GPU model names of the two different systems are abbreviated in the various graph legends, the CPU models to their respective series names and the GPU models to their respective model names.

With the GPU execution times the `#1`, `#2` and `D` indicate the use of solely kernel I, solely kernel II or the dynamic kernel execution respectively. The dynamic kernel execution times for either the GPU interfacing and GPU kernel, are a sum of the execution times for both implementations which depend on the N_{thr} conditional statement, elaborated on in section 5.3.

All CPU execution times indicated by either the A10 (-5757M) or the Xeon (E5-2699 v3), are obtained using the original sequential OmegaPlus version using a single core.

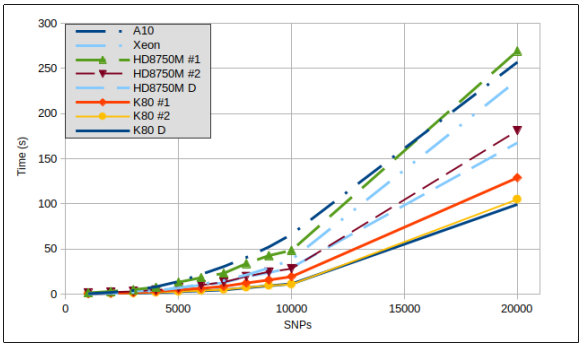
Execution times comparisons

Figure 6.1 illustrates the execution times of the ω-statistic computation on both the CPU and GPU of the two systems. The three graphs, fig. 6.1a, fig. 6.1b and fig. 6.1c, represent the execution times of the complete GPU-accelerated ω-statistic computation, the GPU interfacing, which includes the input and output data transfers and kernel launch, and the GPU kernels itself.

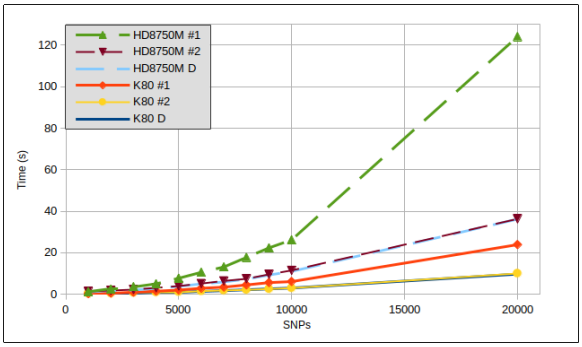
As expected, all three graphs show a quadratic increase in execution time with the number of SNPs, as the total number of computation steps grows quadratic with linear increase in SNPs.

As expected, kernel I performs well with low number of computation steps per grid position but gradually becomes slower as the number SNPs and thus computation steps increase. With 1,000 SNPs kernel I is 1.09x and 1.07x faster than kernel II on System I and II respectively.

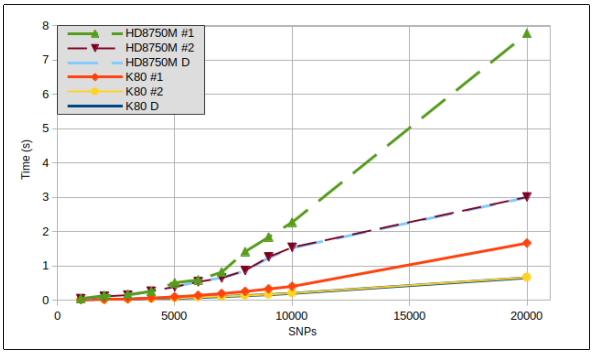
Kernel II performs well over the complete range and the dynamic kernel performs almost identical as only a few grid positions exhibit a low number of computation steps. As reference, the dynamic kernel on the K80 is 1.0x to 1.14x faster than kernel II. From 2,000 to 20,000 SNPs, the dynamic kernel execution is 1.25x to 2.59x and 1.08x to 2.54x faster than kernel I on System I and II respectively.



(a) Complete ω-statistic execution times



(b) GPU interfacing execution times



(c) GPU kernel execution times

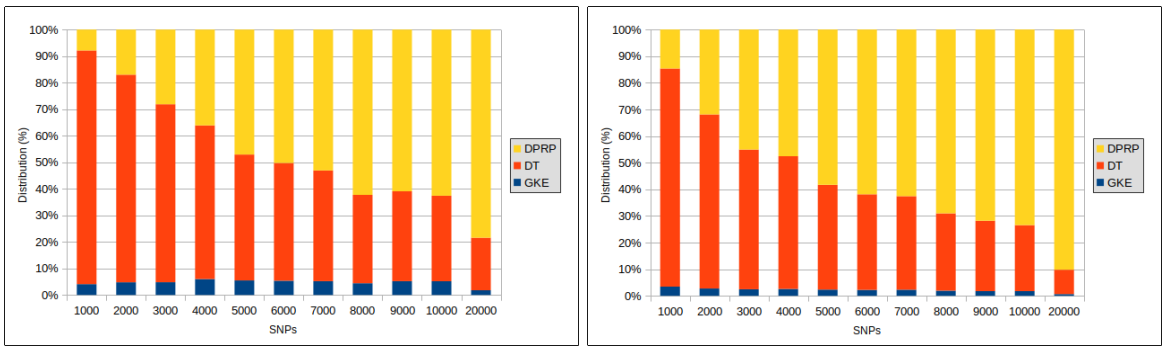
Figure 6.1: Execution times of ω-statistic computation for increasing number of SNPs (from 1,000 to 20,000) and 50 sequences/samples. Figure 6.1a shows the execution times of the complete GPU-accelerated ω-statistic computation, fig. 6.1b shows the execution times of the GPU interfacing, which includes the input and output data transfers and kernel launch, and fig. 6.1c shows the execution times of the GPU kernels itself. We observe a quadratic increase in execution time of all graphs with the number of SNPs. The execution time scales identical to the total number of computation steps or ω values of all the grid positions combined. The kernel execution times of the HD8750M GPU in fig. 6.1c only show some minor discontinuity.

Execution time distributions

Figure 6.2 illustrates the execution times distributions of only the GPU-accelerated ω-statistic computation on both systems. This distribution consists of three main parts that contribute to the total ω-statistic computation execution time (fig. 6.1a):

- **Data Preparation & Result Processing (DPRP):** Initializing of the input and output buffers needed for the used kernel, nested loop execution with eventual dummy values padding and processing of the returned ω values with eventual index values.
- **Data Transfer (DT):** Setting the kernel arguments, writing of the input buffers and reading of the output buffers of the used kernel.
- **GPU Kernel Execution (GKE):** Launching of the used kernel with the set number of work-items or global work-size and work-group or local size.

The data preparation & result processing share in total execution time increases linearly and the GPU transfer share decreases linearly with the number of SNPs. The GPU kernel share in total execution time roughly stays the same. The graphs show that with higher number of SNPs the data preparation & result processing becomes the predominant part in the GPU-accelerated ω-statistic computation with respect to execution time.



(a) ω-statistic execution time distributions on HD8750M

(b) ω-statistic execution time distributions on K80

Figure 6.2: Complete GPU-accelerated ω-statistic execution times distributions for increasing number of SNPs (from 1,000 to 20,000) and 50 sequences/samples. Figure 6.2a shows the time distributions using the HD8750M on System I and fig. 6.2b shows the time distributions using the K80 on System II. The Data Preparation & Result Processing (DPRP), Data Transfer (DT) and GPU Kernel Execution (GKE) shares are shown in the graphs. A linear increase in data preparation & result processing time can be observed with the number of SNPs in both graphs. This is as expected, as the execution times graphs in fig. 6.1 showed a large increase in execution times from the GPU interfacing to the complete ω-statistic computation with higher number of SNPs.

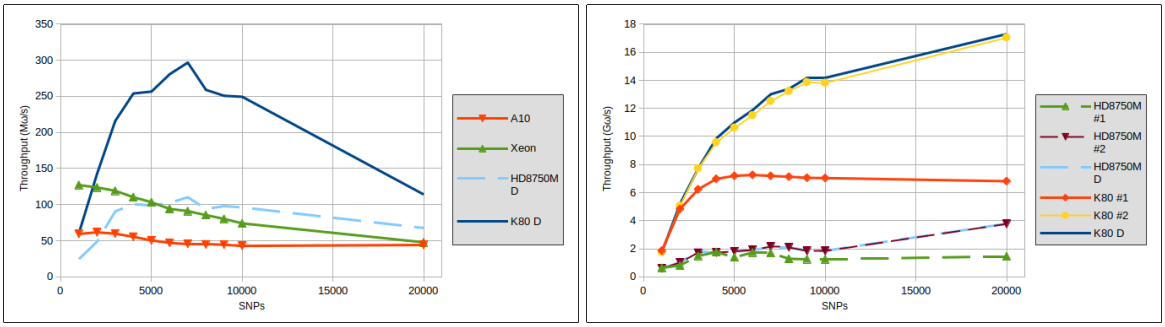
Throughput comparisons

Figure 6.3 illustrates the complete GPU-accelerated ω-statistic throughput values on both systems and architectures using the dynamic kernel (fig. 6.3a), and the kernel throughput values on both GPUs (fig. 6.3b).

The GPU kernels graph clearly shows the aforementioned speedups kernel II, and therefore the dynamic kernel, achieves over kernel I when the number of SNPs increase. After an initial steep increase in throughput of kernel I for relatively low number of SNPs, the throughput of the kernel flattens as the number of SNPs, and thus computations steps, increases. The throughput of kernel II keeps increasing until 20,000 SNPs on both systems.

Even though the throughput of kernel II and the dynamic kernel increases on both systems with the number of SNPs (fig. 6.3b), the throughput of the complete GPU-accelerated ω-statistic computation decreases on both systems after a maximum at 7,000 SNPs (fig. 6.3a). The complete GPU-accelerated ω-statistic computation achieves speedups from 0.41x to 2.41x, comparing the System I CPU and GPU, and

speedups from 0.47x to 3.37x, comparing the System II CPU and GPU, at 1,000 and 7,000 SNPs and 1,000 and 10,000 SNPs respectively. At relatively low number of SNPs (<2,500 SNPs) the GPU-accelerated ω-statistic computation performs worse compared to the OmegaPlus sequential counterpart on the same system.



(a) Complete ω-statistic computation *(b) GPU kernels*

Figure 6.3: Throughput values for increasing number of SNPs (from 1,000 to 20,000) and 50 sequences/samples. Figure 6.3a shows the throughput values (Mω/s) of the complete ω-statistic computation on both systems and architectures using the dynamic kernel and fig. 6.3b shows the throughput values (Gω/s) of the kernels on both systems.

When looking at fig. 6.2, it is clear that the data preparation & result processing on the host, linearly takes up more of the total ω-statistic computation execution time with the number of SNPs. When the number of SNPs becomes larger than 5,000, the data preparation & result processing execution time share becomes larger than 50%. From looking at fig. 6.1a and fig. 6.1b it can be deduced that there is a quadratic increase in data preparation & result processing execution time with the number of SNPs. This causes a decrease in total ω-statistic computation throughput with the number of SNPs shown in fig. 6.3a.

The increase in data preparation & result processing execution time, is due to a far higher number of SNPs per grid position. This requires initialization of larger input buffers and results in high number of nested loop iterations.

With low number of SNPs (<5,000 SNPs), it is clear from fig. 6.2 that the data transfers take up most of the GPU-accelerated ω-statistic computation time. This causes the GPU-accelerated ω-statistic computation performing worse with low number of SNPs (<2,500 SNPs), as shown in fig. 6.3a.

The slowdown with low number of SNPs and thus low number of computation steps is caused by the data transfer overhead which is too large to still benefit from the GPU its massive parallel computation.

6.3.2 Linkage Disequilibrium performance

In this section performance values will be presented of the GEMM-based LD implementation in OmegaPlus. As with the ω-statistic performance graphs, the model names are abbreviated, the CPU models to their respective series names and the GPU models to their respective model names. The GPU execution times are split up into 2 different parts which give a good representation of the time distributions:

- **Total GPU-accelerated LD execution time (Tot.):** Total execution times of both the GEMM-based LD implementation outside of the grid positions for-loop and the original correlation value summing and data reuse optimization within the grid positions for-loop.
- **GEMM-based LD implementation execution time (GEMM):** Execution times of only the GEMM-based LD implementation in OmegaPlus outside of the grid positions for-loop.

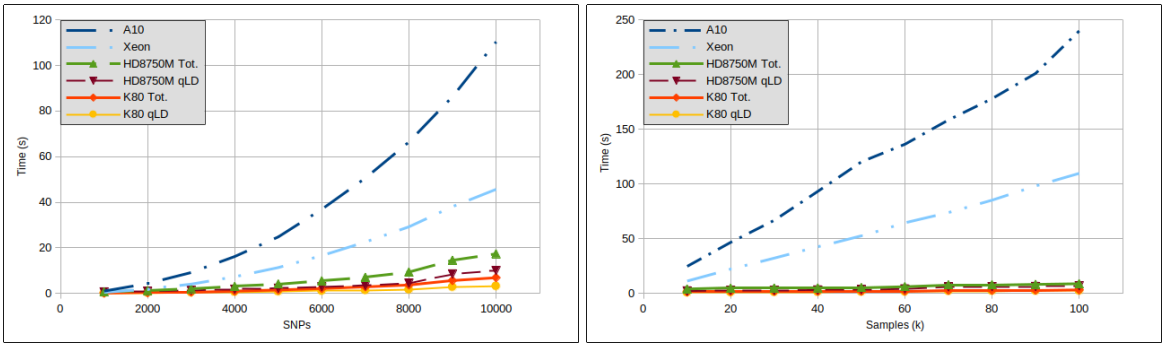
Figure 6.4 illustrates the various LD execution times for increasing number of SNPs (from 1,000 to 10,000) in fig. 6.4a and increasing number of sequences/samples (from 10,000 to 100,000) in fig. 6.4b. The execution times of the original CPU implementation are shown, as well as the execution times of the GEMM-based LD implementation. The GPU execution times show both the actual GEMM-based LD implementation and total GPU-accelerated LD computation times.

The GEMM-based implementation performs identical to the performance results presented in the work by Theodoris et al. [104]. The GPU adapted implementation is up to 7.19x faster on System I and up to 7.93x faster on System II for increasing number of SNPs, when comparing to the original CPU implementation on the corresponding system. The GPU adapted implementation is up to 27.04x faster on System I and up to 33.75x faster on System II for increasing number of sequences/samples, performing the same comparison.

6.3.3 Total performance

In this section performance values will be presented of the total GPU-accelerated OmegaPlus tool. The model names are again abbreviated, the CPU models to their respective series names and the GPU models to their respective model names.

Execution time comparisons Figure 6.5 illustrates the complete OmegaPlus execution times on both systems and both architectures, including both compute intensive parts as well as input and output data processing. Figure 6.5a shows the execution times for increasing number of SNPs (from 1,000 to 10,000) with 10,000



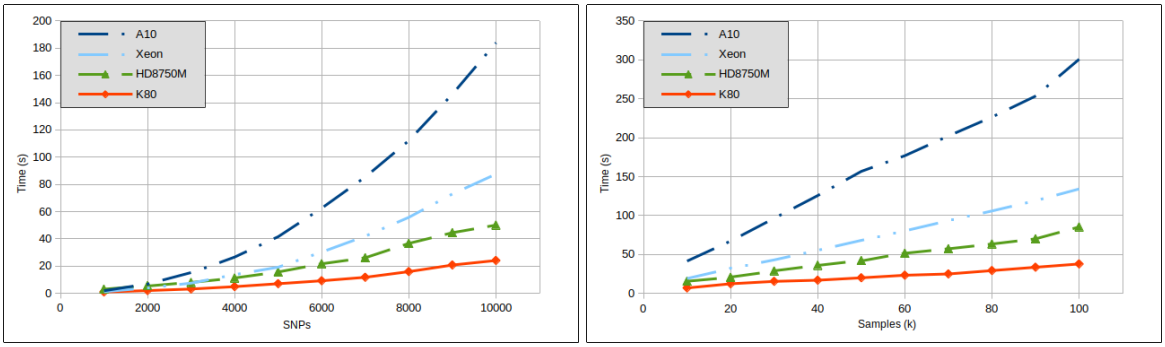
(a) Increasing SNPs *(b) Increasing sequences/samples*

Figure 6.4: Execution times of LD computation for increasing number of SNPs (from 1,000 to 10,000) and 10,000 sequences/samples (fig. 6.4a) and increasing number of sequences/samples (from 10,000 to 100,000) and 5,000 SNPs (fig. 6.4b). As presented in the original work by Theodoris et al. [104], the execution times increase quadratic with the number of SNPs and linear with the number of sequences/samples, excluding minor discontinuities. The GEMM-based LD implementation especially performs well with increasing number of sequences/samples, with speedups ranging from 7.55x to 33.75x when comparing the System II CPU and GPU and speedups ranging from 6.05x to 27.04x when comparing the System I CPU and GPU.

sequences/samples and fig. 6.5b shows the execution times for increasing number of sequences/samples (from 10,000 to 100,000) with 5,000 SNPs.

As expected, the execution times increase quadratic with the number of SNPs and linear with the number of sequences/samples.

When comparing the original sequential version to the GPU-accelerated version on System I and II respectively, speedups range from 0.62x to 3.68x and 1.24x to 3.63x for increasing number of SNPs, and 2.66x to 3.52x and 2.73x to 3.55x for increasing number of sequences/samples.



(a) Increasing SNPs *(b) Increasing sequences/samples*

Figure 6.5: Complete OmegaPlus executions times of both compute intensive parts as well as input and output data processing. Figure 6.5a shows the execution times for increasing number of SNPs (from 1,000 to 10,000) with 10,000 sequences/samples and fig. 6.5b shows the execution times for increasing number of sequences/samples (from 10,000 to 100,000) with 5,000 SNPs.

Execution time distributions

Figure 6.6 illustrates the OmegaPlus execution times distributions of the compute intensive parts on both systems and both architectures. Besides the original sequential version, also the generic parallel version is used to get a good performance comparison of the different versions using different workload distributions.

Three different workload distributions are evaluated over the sequential, generic parallel and GPU-accelerated OmegaPlus version.

Since the total execution time depends on the proportion of LD computation execution time and ω-statistic computation execution time, a performance evaluation is performed with high ω-statistic workload ($\approx$90%), high LD workload ($\approx$90%) and balanced workload ($\approx$50%/50%). For high ω-statistic workload a dataset with 15,000 SNPs and 500 sequences/samples is used, for high LD workload a dataset with 5,000 SNPs and 60,000 sequences/samples is used and for a balanced workload a dataset with 13,000 SNPs and 7,000 sequences/samples is used.

Figure 6.6a illustrates the executions with balanced workload in which the LD and ω-statistic computations take up an equal amount of time on the sequential version. The distribution on System I clearly shows the GPU-accelerated OmegaPlus tool is outperforming the sequential version as both compute intensive parts have lower execution times. The parallel version however, exhibits a lower execution time for the ω-statistic computation but a higher execution time for the LD computation resulting in an overall higher execution time. On System II we can observe the same acceleration of both parts with respect to the sequential and GPU-accelerated version. The parallel version however, exhibits a higher LD execution time while the ω-statistic execution time is lower than the sequential version. The total execution time of the parallel version on System II is therefore also higher than the GPU-accelerated version. Due to the fact that only 2 cores are available of the Xeon CPU it is expected that the speedup isn't as noticeable as on System I (1 to 4 cores increase) but the multi-core utilization on System II seems low.

Figure 6.6b illustrates the executions with a high ω-statistic workload in which approximately 90% of the execution time is used by the ω computations on the sequential version.

The time distribution on System I clearly shows a notable speedup of the GPU-accelerated ω-statistic computation with respect to the sequential version. The LD computation however shows a higher execution time using the GPU-accelerated version, indicating inefficiency in the GEMM-based LD implementation performing computations with low number of sequences/samples. Due to this inefficiency the parallel execution time is notably lower than that of the GPU-accelerated version. The GPU-accelerated ω-statistic computation is also marginally slower than the parallel ω-statistic computation. As expected from fig. 6.3a, the total GPU-accelerated

ω-statistic execution time remains fairly high for a higher number of SNPs.

System II shows similar results with respect to the sequential CPU version and the GPU-accelerated version. The GPU-accelerated LD computation execution time increases slightly due to the GEMM-based LD implementation while the GPU-accelerated ω-statistic execution time reduces. The GPU-accelerated speedup is more notably however due to the faster GPU. The LD computation in the parallel version on System II performs better with low number of sequences/samples and exhibits a lower execution time than both the GPU-accelerated and sequential OmegaPlus version.

Figure 6.6c illustrates the executions with a high LD computation workload in which approximately 90% of the execution time is used by the LD computations on the sequential version.

The distribution on System I clearly shows the efficiency of the GEMM-based LD implementation at high number of sequences/samples, as expected from fig. 6.4. Due to the massive decrease in GPU-accelerated LD computation execution time, it is much faster than the sequential version and also notably faster than the parallel version using 4 cores.

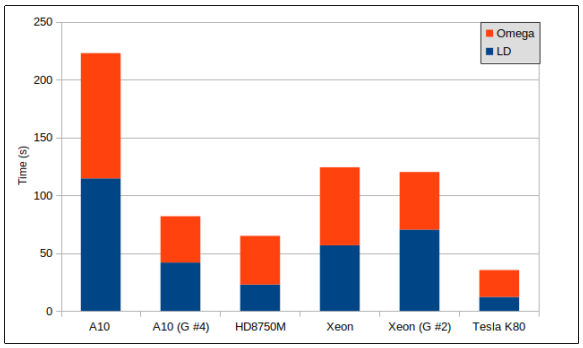
System II shows the same expected results with respect to the GPU-accelerated LD computation and with respect to the LD computation on the parallel version. As in fig. 6.6a, the parallel version on System II using 2 cores shows an increase in execution time for the LD computation with high number of sequences/samples.

When looking at both systems it can be seen that the GPU-accelerated ω-statistic execution times decrease slightly from the sequential version and are approximately on par with the parallel version on both systems. This can be expected from fig. 6.1a which showed the expected efficiency for 5,000 SNPs.

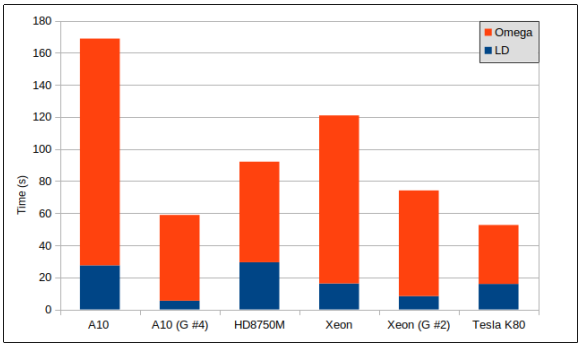
From fig. 6.6 it is clear that the ω-statistic computation generally takes up most of the total execution time in the GPU-accelerated OmegaPlus version. To further improve performance of the GPU-accelerated OmegaPlus version, improving performance of the ω-statistic computation would have the biggest impact.

As stated earlier the data preparation & result processing takes up most of the GPU-accelerated ω-statistic computation execution time with higher number of SNPs. Improving performance of this part would have the biggest impact on the overall GPU-accelerated OmegaPlus performance, given the GPU-accelerated LD computation implementation as well as the ω-statistic GPU kernels are optimized. The performance curve of the GPU-accelerated ω-statistic computation, shown in fig. 6.3a, could then be more flat towards higher number of SNPs instead of the drop.

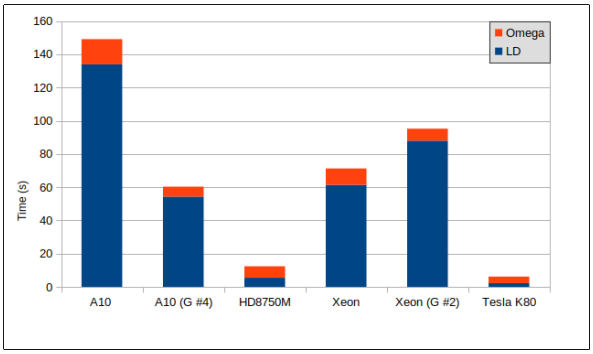
As stated earlier, with low number of SNPs ($<$5,000 SNPs), the data transfers take up most of the GPU-accelerated ω-statistic computation time. Improving performance of this part would increase throughput values in the low number of SNPs



(a) Balanced workload ≈50%/50% (7k samples/13k SNPs)



(b) High Omega workload ≈90%/10% (500 samples/15k SNPs)



(c) High LD workload ≈10%/90% (60k samples/5k SNPs)

Figure 6.6: Time distributions of both compute intensive parts for a balanced workload (≈50%/50% ω-statistic and LD computation time, fig. 6.6a), a high ω-statistic workload (≈90%/10% ω-statistic and LD computation time, fig. 6.6b) and a high ω-statistic workload (≈10%/90% ω-statistic and LD computation time, fig. 6.6c).

range and therefore reduce the slowdown of the GPU-accelerated ω-statistic computation with respect to the original sequential version in this range, shown in fig. 6.3a,

The GEMM-based LD implementation performs especially well with increasing number of sequences/samples. However, performance eventually drops significantly when the number of sequences/samples become really low and the number of SNPs remain high ($<$1000 sequences/samples and $>$10,000 SNPs).

Overall the GPU-accelerated LD computation achieves much higher speedups than the GPU-accelerated ω-statistic computation. This reveals a highly optimized approach for the LD computation and potential performance improvements for the aforementioned ω-statistic computation parts.

Table 6.4 shows the speedups between all combinations of 3 OmegaPlus versions, 2 systems and using the 3 different workload distributions. The table shows that the GPU-accelerated version achieves speedups, over either the generic parallel or sequential OmegaPlus version, between 1.26x to 6.3x with balanced workload, between 0.64x to 3.21x with high ω-statistic computation workload and between 4.86x to 24.0x with high LD computation workload. This shows the high efficiency of the GPU-accelerated LD computation which is beneficial for real-world datasets

which tend to have more and more sequences/samples resulting in an imbalanced distribution with respect to number of SNPs and number of sequences/samples. The performance of the GPU-accelerated OmegaPlus version will thus be closer to the evaluated performance using real-world datasets with imbalanced, high LD computation workload.

One example is the sample size in SARS-CoV-2 datasets which increased from a few thousand to a few million since the beginning of the global pandemic (`https://www.gisaid.org/`).

	CPU seq. vs. par.		CPU vs. GPU (same system)				CPU seq. vs. par.		CPU vs. GPU (different system)			
Dist. ω/LD	A10 vs. A10 (G #4)	Xeon vs. Xeon (G #2)	A10 vs. HD8750M	A10 (G #4) vs. HD8750M	Xeon vs. K80	Xeon (G #2) vs. K80	A10 vs. Xeon (G #2)	Xeon vs. A10 (G #4)	A10 vs. K80	A10 (G #4) vs. K80	Xeon vs. HD8750M	Xeon (G #2) vs. HD8750M
50/50	2.72x	1.03x	3.43x	1.26x	3.51x	3.40x	1.86x	1.52x	6.30x	2.31x	1.91x	1.85x
90/10	2.87x	1.63	1.83x	0.64x	2.30x	1.41x	2.28x	2.05x	3.21x	1.12x	1.31x	0.81x
10/90	2.47x	0.75x	12.02x	4.86x	11.48x	15.33x	1.57x	1.18x	24.00x	9.71x	5.75x	7.68x

Table 6.4: Table showing the speedups between all combinations of 3 OmegaPlus versions (sequential, generic parallel, GPU-accelerated), 2 systems (Laptop, Colab) and using the 3 different workload distributions. The seq. and par. affixes indicate the sequential and generic parallel OmegaPlus versions respectively. The CPU is indicated with the (G #n) affix for the generic parallel version in which the n indicates the used number of threads.

Chapter 7

Conclusions and future work

In this chapter the conclusion of the project is presented in section 7.1, which elaborates on answering the main research question and the subquestions, as well as concluding the work.

After the conclusion some future work is presented in section 7.2, which will give recommendations for future work/research that can further improve performance of the developed solution.

7.1 Conclusion

This thesis presented the acceleration of a state-of-the-art selective sweep detection software called OmegaPlus using the massive parallel GPU architecture.

The goal of the project was to boost performance of OmegaPlus by mapping the compute intensive parts of the tool optimally on the GPU architecture. The OpenCL General Purpose GPU (GPGPU) framework has been used to achieve this.

Within the project, OmegaPlus has been decomposed to elaborate on the compute intensive parts of the tool and to gather information about the execution time distribution within the tool. The LD computation, OmegaPlus is based on, was found to be a compute intensive part as well as the ω-statistic computation, which is used to localize the selective sweeps.

The thesis elaborated on an adaptation of the highly optimized LD computation tool quickLD, a GPU-accelerated GEMM-based LD implementation. This GEMM-based implementation accelerates the compute intensive LD computation using high-performance Dense Linear Algebra (DLA) operations mapped on the GPU architecture.

The ω-statistic computation is accelerated using a novel dynamic approach that distinguishes workloads for dynamic GPU kernel execution. The two GPU kernels are developed for either a high or low ω-statistic workload. The complete kernel

designs are described, as well as the host CPU code that prepares the data for optimal kernel execution and takes care of the GPU interfacing.

A performance evaluation has been conducted to verify correctness of the developed solution and to gather performance comparisons of the compute intensive parts as well as the complete GPU-accelerated OmegaPlus version.

In the following paragraphs the formulated research questions for this thesis are answered.

Sub-questions The first two sub-research questions are set up to steer the research in the right direction to achieve better results with respect to the main research question and main goal of this project.

- Which parts of the state-of-the-art tool are the most computationally intensive to target for acceleration?

As stated in the conclusion, within this project the chosen state-of-the-art selective sweep detection tool, OmegaPlus, is decomposed to describe the compute intensive parts and to perform profiling to gather execution times of the compute intensive parts. The two compute intensive parts were found to be the LD computation and ω-statistic computation which together take up $>$95% of the total execution time. With a simulated dataset of consisting of 7,000 sequences/samples and 13,000 SNPs, the LD computation takes up 50.3% of the total execution time and the ω-statistic computation takes up 47.9% of the total execution time.

- How can the computationally intensive parts be mapped optimally on the computational units and the different types of memory of the GPU?

For the compute intensive LD computation the highly optimized quickLD tool is adapted in order to apply GPU-acceleration. This decision has been made to prevent reinventing the wheel and be able to shift the focus on GPU-accelerating the compute intensive ω-statistic computation.

Due to non-uniform SNP distribution along the genome, large variation in computational workload for the ω-statistic computation can occur during execution. The ω-statistic computation has therefore been accelerated using two GPU kernels that are executed dynamically depending on the computational workload. For a low number of computations a rather naive approach is applied that mimics the original work-flow of the sequential OmegaPlus version. For a high number of computations a second kernel is applied that applies multiple acceleration techniques in order to optimize kernel execution.

The kernel designed for low computational load applies minimal data transfers with respect to the input buffers, which are stored on global memory. In order to fully

utilize the GPU its computational units, Compute Units (CUs)/Streaming Multiprocessors (SMs), with the low computational load, every scheduled work-item scheduled on the GPU computes only a single ω value. This prevents underutilization of the hardware as much as possible, in which memory latencies can be more prevalent. A fully coalesced access pattern is applied to the largest input buffer in order to minimize memory latencies. The two other input buffers are accessed through an unoptimized pattern in order to facilitate minimal input buffers data transfers. The single output buffer is written using coalesced memory accesses to maximize performance of the result storing.

The kernel designed for high computational load applies optimal occupancy through a global work-size indication based on the number of computational units, CUs/SMs, on the GPU. The number of total work-items is set closest to the set optimal occupancy metric and as an integer multiple of the number of iterations in the dynamically placed inner loop. This enables memory latency hiding and optimized memory access patterns. Input buffers are stored on global memory and accessed through either coalesced memory accesses or single element accesses. Dummy values are used to achieve the optimal coalesced memory accesses which results in dummy computations and non-minimal data transfer. A loop unrolled for-loop with factor 4 is applied to perform all the computation steps of a specific grid position. Given a single work-items performs multiple computation steps within this kernel, fewer resulting ω-statistic values need to written to the output buffer. An additional output buffer is however needed for index values of the corresponding maximum ω-statistic values. Both output buffers are written using coalesced memory accesses to maximize performance of the result storing.

- How do performance values scale for datasets with different number of genomes and datasets with different number of SNPs per genome?

As expected, the GPU-accelerated ω-statistic computation showed a quadratic increase in execution time in the performance evaluation with increasing number of SNPs. With higher number of SNPs however ($>$8,000 SNPs), the GPU-accelerated ω-statistic computation showed a decrease in throughput which resulted in lower speedups when compared to the sequential OmegaPlus version at lower number of SNPs. This is caused by an increase in data preparation & result processing execution time for the GPU-accelerated ω-statistic computation.

When looking at the performance evaluation of the LD computation the performance values scale identical to the values presented in the original work by Theodoris et al. [104], [105]. A quadratic increase of execution time with the number of SNPs and a linear increase of execution time with the number of sequences/samples.

When looking at the complete GPU-accelerated OmegaPlus version, performance values scale as expected when considering the results of the compute intensive parts separately. With increasing number of SNPs the execution time shows a quadratic increase and with increasing number of sequences samples the execution time shows a linear increase.

Main question The main research question will be answered by looking at the performance values presented with the varying workloads. This provides a good insight in the potential performance gain for different workload distributions which can be compared to real-world datasets.

- How much performance gain can be achieved when accelerating a state-of-the-art selective sweep detection tool using GPU architectures?

The performance evaluation with the balanced workload dataset, in which the execution time of both compute intensive parts is equally distributed over the two compute intensive parts, showed speedups ranging from 1.91x to 6.3x and from 1.26x to 3.4x when comparing the GPU-accelerated OmegaPlus version to the sequential and generic parallel OmegaPlus versions respectively. The generic parallel version used 4 and 2 cores for the mentioned speedups respectively.

The performance evaluation with the high ω-statistic computation workload, in which the execution time of the ω-statistic computation takes up approximately 90% of the combined execution time, showed speedups ranging from 1.31x to 3.21x and from 0.64x to 1.41x when comparing the GPU-accelerated OmegaPlus version to the sequential and generic parallel OmegaPlus versions respectively. Again 4 and 2 cores were used with the generic parallel version. The lower exhibited speedups are due to the inefficiency of the GEMM-based LD implementation with a low number of sequences/samples, 500 in the case of this dataset. The sequential and generic parallel OmegaPlus versions exhibit higher LD computation performance with low number of sequences/samples.

The performance evaluation with the high LD computation workload, in which the execution time of the LD computation takes up approximately 90% of the combined execution time, showed speedups ranging from 5.75x to 24.0x and from 4.86x to 15.33x when comparing the GPU-accelerated OmegaPlus version to the sequential and generic parallel OmegaPlus versions respectively. The mentioned speedups were again acquired using 4 and 2 cores for the generic parallel version. These speedups show the high efficiency of the GEMM-based LD implementation with a high number of sequences/samples, 60,000 in the case of this dataset. Given the fact real-world datasets tend to have more and more sequences/samples, evaluated

performance values using the high LD computation workload distribution, are more representative for real-world applications.

To conclude, performance values heavily depend on the workload distribution between the two compute intensive parts and on the number of SNPs and sequences/samples in the dataset used. Speedups can however be as high as 24.0x when datasets consist of a high number of sequences/samples and moderate number of SNPs.

7.2 Future work

This section will elaborate on future work that can be carried out to further boost performance and/or optimize the GPU-accelerated OmegaPlus version. In the paragraphs below, each recommendation is separately discussed.

As stated in section 6.3, to further improve performance of the GPU-accelerated OmegaPlus version, improving performance of the ω-statistic computation would have the biggest impact. Within this compute intensive part the data preparation & result processing takes up most of the ω-statistic computation execution time.

Execution overlapping An acceleration technique that can be applied, and was mentioned in section 2.2, is overlapping data transfers and GPU kernel execution. From fig. 6.2 we can only conclude that the GPU kernel execution time takes up less than 10% of the total ω-statistic computation execution time. Overlapping these parts would therefore result in a marginal performance gain.

Overlapping of both the GPU kernel execution and the data transfers with the data preparation would result in higher performance gains, especially with higher number of SNPs.

This can be implemented by dividing the execution of the nested loop, and therefore the input data, into smaller parts using an additional for-loop. This would enable to queue data transfers and kernel execution, using the OpenCL command queue, for the part of the data that has been prepared in a part of the complete nested loop. After queuing the data transfers and kernel execution for this part of the input data, the next part can be prepared and thereafter queued, resulting in overlap. Given the fact the queuing is non-blocking and there exist no data dependencies in the kernel, host code execution can continue while the GPU handles the queued OpenCL events.

Important is that the partial data preparation execution time, or nested loop execution time, is balanced with the data transfers and kernel execution time in order to maximize overlap and therefore performance gain. This depends on the partial size

of the input data which can be defined by the number of iterations of the additional for-loop.

After all the parts, including data transfer and kernel execution, are finished executing, only one blocking read needs to be performed to retrieve all the GPU-computed ω values as in the current approach using the OpenCL event wait list. It is also possible to read parts of the GPU-computed ω values, by reading values computed in the previous additional for-loop iteration which are ready if the execution time is balanced.

Besides improving performance, the overlapping execution also enables lower memory usage as only parts of the grid position dependable data need to be temporarily stored.

Host code parallelization An additional approach to improve performance of the GPU-accelerated ω-statistic computation, is to parallelize the CPU host code. Especially the nested loop in the implemented approach can consists of several mullion iterations in which data is stored in the input buffers.

This can however also degrade performance as the computation to thread synchronisation ratio can be unfavourable within this implementation. This should be evaluated to obtain the actual performance values.

GEMM-based LD implementation optimization The GEMM-based LD implementation can also be optimized. In the original sequential OmegaPlus version, the correlation values are computed per grid position and stored in lower triangular matrix M. The GEMM-based LD implementation omits the computation of correlation values per grid position and instead computes all the correlation values at once using GPU-acceleration. The GPU-accelerated LD computation however, implements a data transfer from the GEMM-based LD resulting correlation matrix to the original OmegaPlus lower triangular matrix. These transfers are redundant if the original OmegaPlus lower triangular correlation matrix can be bypassed, and only the GEMM-based results are used for within the data-reuse optimization and the ω-statistic computation.

Higher GPU utilization Performance gain can also be achieved by performing more computations on the GPU. The current approach accelerates the computation of the haplotype frequency matrix H, recall section 4.2, essentially a general matrix multiplication operation (GEMM) operation, and the complete computation of the ω-statistic. However, all the computations and data processing in between the two GPU-accelerated parts is performed sequentially on the CPU. The developed

solution would require a large change to enable all the computations and data processing in between the two GPU-accelerated parts to be performed on the GPU. An additional GPU kernel for computing the Pearson correlation values with haplotype frequency matrix H would be required as well as a complete different kernel for computing the ω-statistic values from the correlation values on the GPU. This approach would minimize data transfer overhead but would increase GPU kernel execution time. The potential performance improvement of this approach however, depends on whether these extra computations are suitable for GPU acceleration.

Acknowledgements

At the end of my thesis and thus the end of my time on the University of Twente, I look back on an instructive, interesting and challenging period with a bittersweet ending. The global pandemic made certain times during my graduation, and the period before that, feel very strange and provided little motivation. Luckily, I had enough people around who motivated me to keep working on my project, and enough people around to enjoy my spare time with.

First of all, I want to thank dr. ir. Nikolaos Alachiotis, my daily supervisor and the helping hand in solving the more challenging problems I encountered. Thank you for letting me work on this interesting project where I learned a lot about this specific topic which was completely new to me.

I also want to thank my roommates from my student house *B100*, which is now slowly becoming a house full of 'normal civilians'. Thank you for hearing me out about my project and giving me useful tips on the continuation.

I also want to thank my uncle who was always interested in how my study was progressing and what I was working on, no matter how difficult the topic.

Next, I want to thank my friends with whom I made great memories the last, and previous years, and will certainly make a lot more memories with. Even though most of the time we spend together is devoted to drinking beer, I can always talk to you about all sorts of stuff, for which I'm truly grateful.

At last I want to thank my loving family of which especially my parents. You have been my greatest support, motivation and inspiration during my school days, my time at the university and throughout my entire life. You gave me the resources and possibilities to work on my career path and to develop myself into the person I am today. Dankjewel!

Bibliography

[1] 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.

[2] N. Alachiotis, P. Pavlidis, and A. Stamatakis, "Exploiting multi-grain parallelism for efficient selective sweep detection," in *International Conference on Algorithms and Architectures for Parallel Processing*. Springer, 2012, pp. 56–68.

[3] J. E. Stone, D. Gohara, and G. Shi, "Opencl: A parallel programming standard for heterogeneous computing systems," *Computing in science & engineering*, vol. 12, no. 3, pp. 66–73, 2010.

[4] R. M. Ames, D. Money, V. P. Ghatge, S. Whelan, and S. C. Lovell, "Determining the evolutionary history of gene families," *Bioinformatics*, vol. 28, no. 1, pp. 48–55, 2012.

[5] A. Shen, H. Fu, K. He, and H. Jiang, "False discovery rate control in cancer biomarker selection using knockoffs," *Cancers*, vol. 11, no. 6, p. 744, 2019.

[6] E. W. Sayers, M. Cavanaugh, K. Clark, K. D. Pruitt, C. L. Schoch, S. T. Sherry, and I. Karsch-Mizrachi, "Genbank," *Nucleic acids research*, vol. 49, no. D1, pp. D92–D96, 2021.

[7] Y. Shu and J. McCauley, "Gisaid: Global initiative on sharing all influenza data–from vision to reality," *Eurosurveillance*, vol. 22, no. 13, p. 30494, 2017.

[8] C. A. Davis, B. C. Hitz, C. A. Sloan, E. T. Chan, J. M. Davidson, I. Gabdank, J. A. Hilton, K. Jain, U. K. Baymuradov, A. K. Narayanan *et al.*, "The encyclopedia of dna elements (encode): data portal update," *Nucleic acids research*, vol. 46, no. D1, pp. D794–D801, 2018.

[9] J. N. Weinstein, E. A. Collisson, G. B. Mills, K. R. M. Shaw, B. A. Ozenberger, K. Ellrott, I. Shmulevich, C. Sander, and J. M. Stuart, "The cancer genome atlas pan-cancer analysis project," *Nature genetics*, vol. 45, no. 10, pp. 1113–1120, 2013.

[10] K. S. Button, J. P. Ioannidis, C. Mokrysz, B. A. Nosek, J. Flint, E. S. Robinson, and M. R. Munafò, "Power failure: why small sample size undermines the reliability of neuroscience," *Nature reviews neuroscience*, vol. 14, no. 5, pp. 365–376, 2013.

[11] K. Katoh and D. M. Standley, "Mafft multiple sequence alignment software version 7: improvements in performance and usability," *Molecular biology and evolution*, vol. 30, no. 4, pp. 772–780, 2013.

[12] B. Morel, P. Barbera, L. Czech, B. Bettisworth, L. Hübner, S. Lutteropp, D. Serdari, E.-G. Kostaki, I. Mamais, A. Kozlov *et al.*, "Phylogenetic analysis of sars-cov-2 data is difficult," *bioRxiv*, 2020.

[13] J. M. Sá, O. Twu, K. Hayton, S. Reyes, M. P. Fay, P. Ringwald, and T. E. Wellems, "Geographic patterns of plasmodium falciparum drug resistance distinguished by differential responses to amodiaquine and chloroquine," *Proceedings of the National Academy of Sciences*, vol. 106, no. 45, pp. 18 883–18 889, 2009.

[14] A. Rambaut, O. G. Pybus, M. I. Nelson, C. Viboud, J. K. Taubenberger, and E. C. Holmes, "The genomic and epidemiological dynamics of human influenza a virus," *Nature*, vol. 453, no. 7195, pp. 615–619, 2008.

[15] 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.

[16] E. Binder, T. M. Low, and D. T. Popovici, "A portable gpu framework for snp comparisons," in *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2019, pp. 199–208.

[17] N. Alachiotis, T. Popovici, and T. M. Low, "Efficient computation of linkage disequilibria as dense linear algebra operations," in *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2016, pp. 418–427.

[18] W. S. Klug, M. R. Cummings, C. A. Spencer, M. A. Palladino, and D. J. Killian, *Essentials of genetics*. Pearson Education, Inc., 2020.

[19] C. Darwin, A. R. Wallace, S. C. Lyell, and J. D. Hooker, "On the tendency of species to form varieties: and on the perpetuation of varieties and species by natural means of selection." Linnean Society of London, 1858.

[20] "[LS4-4] Natural Selection and Adaptation," Accessed on: Sept. 28, 2021. [Online]. Available: https://biologydictionary.net/ngss-high-school-tutorials/ls4-4-natural-selection-and-adaptation/

[21] "What is a mutation?" Accessed on: Sept. 28, 2021. [Online]. Available: https://www.yourgenome.org/facts/what-is-a-mutation

[22] S. J. Gould, *The structure of evolutionary theory*. Harvard University Press, 2002.

[23] "Genetic drift," Understanding Evolution. University of California Museum of Paleontology., Accessed on: Sept. 28, 2021. [Online]. Available: https://evolution.berkeley.edu/evolibrary/article/evo_24

[24] M. Slarkin, "Gene flow in natural populations," *Annual review of ecology and systematics*, vol. 16, no. 1, pp. 393–430, 1985.

[25] "Gene flow," Understanding Evolution. University of California Museum of Paleontology., Accessed on: Sept. 28, 2021. [Online]. Available: https://evolution.berkeley.edu/evolibrary/article/evo_21

[26] D. W. Mount, *Bioinformatics: Sequence and Genome Analysis*, ser. Cold Spring Harbor Laboratory Series. Cold Spring Harbor Laboratory Press, 2004. [Online]. Available: https://books.google.nl/books?id=M8pqAAAAMAAJ

[27] T. F. Smith and M. S. Waterman, "Identification of common molecular subsequences," *Journal of Molecular Biology*, vol. 147, no. 1, pp. 195–197, 1981.

[28] S. F. Altschul, T. L. Madden, A. A. Schäffer, J. Zhang, Z. Zhang, W. Miller, and D. J. Lipman, "Gapped blast and psi-blast: a new generation of protein database search programs," *Nucleic acids research*, vol. 25, no. 17, pp. 3389–3402, 1997.

[29] J. D. Thompson, D. G. Higgins, and T. J. Gibson, "CLUSTAL W: improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position-specific gap penalties and weight matrix choice," *Nucleic Acids Research*, vol. 22, no. 22, pp. 4673–4680, 11 1994. [Online]. Available: https://doi.org/10.1093/nar/22.22.4673

[30] H. Lopez-Maestre, L. Brinza, C. Marchet, J. Kielbassa, S. Bastien, M. Boutigny, D. Monnin, A. E. Filali, C. M. Carareto, C. Vieira *et al.*, "Snp calling from rna-seq data without a reference genome: identification, quantification, differential analysis and impact on the protein sequence," *Nucleic Acids Research*, vol. 44, no. 19, pp. e148–e148, 2016.

[31] M. A. DePristo, E. Banks, R. Poplin, K. V. Garimella, J. R. Maguire, C. Hartl, A. A. Philippakis, G. Del Angel, M. A. Rivas, M. Hanna *et al.*, "A framework for variation discovery and genotyping using next-generation dna sequencing data," *Nature genetics*, vol. 43, no. 5, pp. 491–498, 2011.

[32] H. Li, "A statistical framework for snp calling, mutation discovery, association mapping and population genetical parameter estimation from sequencing data," *Bioinformatics*, vol. 27, no. 21, pp. 2987–2993, 2011.

[33] E. Garrison and G. Marth, "Haplotype-based variant detection from short-read sequencing," *arXiv preprint arXiv:1207.3907*, 2012.

[34] A. Rimmer, H. Phan, I. Mathieson, Z. Iqbal, S. R. Twigg, A. O. Wilkie, G. McVean, and G. Lunter, "Integrating mapping-, assembly-and haplotype-based approaches for calling variants in clinical sequencing applications," *Nature genetics*, vol. 46, no. 8, pp. 912–918, 2014.

[35] Z. Wei, W. Wang, P. Hu, G. J. Lyon, and H. Hakonarson, "Snver: a statistical tool for variant calling in analysis of pooled or individual next-generation sequencing data," *Nucleic acids research*, vol. 39, no. 19, pp. e132–e132, 2011.

[36] Z. Lai, A. Markovets, M. Ahdesmaki, B. Chapman, O. Hofmann, R. McEwen, J. Johnson, B. Dougherty, J. C. Barrett, and J. R. Dry, "Vardict: a novel and versatile variant caller for next-generation sequencing in cancer research," *Nucleic acids research*, vol. 44, no. 11, pp. e108–e108, 2016.

[37] D. C. Koboldt, Q. Zhang, D. E. Larson, D. Shen, M. D. McLellan, L. Lin, C. A. Miller, E. R. Mardis, L. Ding, and R. K. Wilson, "Varscan 2: somatic mutation and copy number alteration discovery in cancer by exome sequencing," *Genome research*, vol. 22, no. 3, pp. 568–576, 2012.

[38] "Single Nucleotide Polymorphisms (SNPs)," National Human Genome Research Institute, Accessed on: Sept. 28, 2021. [Online]. Available: https://www.genome.gov/genetics-glossary/Single-Nucleotide-Polymorphisms

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

[40] N. L. Kaplan, R. R. Hudson, and C. H. Langley, "The" hitchhiking effect" revisited." *Genetics*, vol. 123, no. 4, pp. 887–899, 1989.

[41] H. Innan and Y. Kim, "Pattern of polymorphism after strong artificial selection in a domestication event," *Proceedings of the National Academy of Sciences*, vol. 101, no. 29, pp. 10 667–10 672, 2004.

[42] J. Hermisson and P. S. Pennings, “Soft sweeps: molecular population genetics of adaptation from standing genetic variation,” *Genetics*, vol. 169, no. 4, pp. 2335–2352, 2005.

[43] M. Prezeworski, G. Coop, and J. D. Wall, “The signature of positive selection on standing genetic variation,” *Evolution*, vol. 59, no. 11, pp. 2312–2323, 2005.

[44] 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.

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

[46] S. Maloy and K. Hughes, *Brenner’s Encyclopedia of Genetics*. Elsevier Science, 2013. [Online]. Available: https://books.google.nl/books?id=4cj64BhrnjcC

[47] D. B. Goldstein and M. E. Weale, “Population genomics: linkage disequilibrium holds the key,” *Current Biology*, vol. 11, no. 14, pp. R576–R579, 2001.

[48] U. Cheramangalath, R. Nasre, and Y. N. Srikant, *GPU Architecture and Programming Challenges*. Cham: Springer International Publishing, 2020, pp. 123–136. [Online]. Available: https://doi.org/10.1007/978-3-030-41886-1_5

[49] NVIDIA, P. Vingelmann, and F. H. Fitzek, “Cuda, release: 11.4,” 2021. [Online]. Available: https://developer.nvidia.com/cuda-toolkit

[50] K. Choo, W. Panlener, and B. Jang, “Understanding and optimizing gpu cache memory performance for compute workloads,” in *2014 IEEE 13th International Symposium on Parallel and Distributed Computing*. IEEE, 2014, pp. 189–196.

[51] “OPENCL Optimization,” OPENCL Optimization - ROCm Documentation 1.0.0, Accessed on: Oct. 3, 2021. [Online]. Available: https://rocmdocs.amd.com/en/latest/Programming_Guides/Opencl-optimization.html

[52] “CUDA C++ Best Practices Guide,” CUDA Toolkit Documentation v11.4.2, Accessed on: Oct. 3, 2021. [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html

[53] "OpenCL Programming Guide," OpenCL Programming Guide - ROCm Documentation 1.0.0, Accessed on: Oct. 3, 2021. [Online]. Available: https://rocmdocs.amd.com/en/latest/Programming_Guides/Opencl-programming-guide.html

[54] B. Chor and T. Tuller, "Maximum likelihood of evolutionary trees: hardness and approximation," *Bioinformatics*, vol. 21, no. suppl_1, pp. i97–i106, 2005.

[55] W. H. Day, D. S. Johnson, and D. Sankoff, "The computational complexity of inferring rooted phylogenies by parsimony," *Mathematical biosciences*, vol. 81, no. 1, pp. 33–42, 1986.

[56] J. Felsenstein, "Evolutionary trees from dna sequences: a maximum likelihood approach," *Journal of molecular evolution*, vol. 17, no. 6, pp. 368–376, 1981.

[57] A. Stamatakis, "Raxml version 8: a tool for phylogenetic analysis and post-analysis of large phylogenies," *Bioinformatics*, vol. 30, no. 9, pp. 1312–1313, 2014.

[58] D. J. Zwickl, "Genetic algorithm approaches for the phylogenetic analysis of large biological sequence datasets under the maximum likelihood criterion," Ph.D. dissertation, 2006.

[59] F. Ronquist and J. P. Huelsenbeck, "Mrbayes 3: Bayesian phylogenetic inference under mixed models," *Bioinformatics*, vol. 19, no. 12, pp. 1572–1574, 2003.

[60] Z. Yang and B. Rannala, "Molecular phylogenetics: principles and practice," *Nature reviews genetics*, vol. 13, no. 5, pp. 303–314, 2012.

[61] P. A. Goloboff, S. A. Catalano, J. Marcos Mirande, C. A. Szumik, J. Salvador Arias, M. Källersjö, and J. S. Farris, "Phylogenetic analysis of 73 060 taxa corroborates major eukaryotic groups," *Cladistics*, vol. 25, no. 3, pp. 211–230, 2009.

[62] W. M. Fitch, "Toward defining the course of evolution: minimum change for a specific tree topology," *Systematic Biology*, vol. 20, no. 4, pp. 406–416, 1971.

[63] D. Sankoff and P. Rousseau, "Locating the vertices of a steiner tree in an arbitrary metric space," *Mathematical Programming*, vol. 9, no. 1, pp. 240–246, 1975.

[64] H. Block and T. Maruyama, "Fpga hardware acceleration of a phylogenetic tree reconstruction with maximum parsimony algorithm," *IEICE TRANSACTIONS on Information and Systems*, vol. 100, no. 2, pp. 256–264, 2017.

[65] N. Alachiotis and A. Stamatakis, “Fpga acceleration of the phylogenetic parsimony kernel?” in *2011 21st International Conference on Field Programmable Logic and Applications*. IEEE, 2011, pp. 417–422.

[66] S. Santander-Jiménez, M. A. Vega-Rodríguez, J. Vicente-Viola, and L. Sousa, “Comparative assessment of gpgpu technologies to accelerate objective functions: A case study on parsimony,” *Journal of Parallel and Distributed Computing*, vol. 126, pp. 67–81, 2019.

[67] H. Block and T. Maruyama, “A hardware acceleration of a phylogenetic tree reconstruction with maximum parsimony algorithm using fpga,” in *2013 International Conference on Field-Programmable Technology (FPT)*. IEEE, 2013, pp. 318–321.

[68] A. Goeffon, J.-M. Richer, and J.-K. Hao, “Progressive tree neighborhood applied to the maximum parsimony problem,” *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, vol. 5, no. 1, pp. 136–145, 2008.

[69] P. A. Goloboff, J. S. Farris, and K. C. Nixon, “Tnt, a free program for phylogenetic analysis,” *Cladistics*, vol. 24, no. 5, pp. 774–786, 2008.

[70] S. Kasap and K. Benkrid, “A high performance fpga-based core for phylogenetic analysis with maximum parsimony method,” in *2009 International Conference on Field-Programmable Technology*. IEEE, 2009, pp. 271–277.

[71] ——, “High performance phylogenetic analysis with maximum parsimony on reconfigurable hardware,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 19, no. 5, pp. 796–808, 2010.

[72] R. Baxter, S. Booth, M. Bull, G. Cawood, J. Perry, M. Parsons, A. Simpson, A. Trew, A. McCormick, G. Smart *et al.*, “Maxwell-a 64 fpga supercomputer,” in *Second NASA/ESA Conference on Adaptive Hardware and Systems (AHS 2007)*. IEEE, 2007, pp. 287–294.

[73] D. Swofford, *PAUP*. Phylogenetic Analysis Using Parsimony (*and Other Methods). Version 4.0b10*, 01 2002, vol. Version 4.0.

[74] H. Block and T. Maruyama, “An fpga hardware acceleration of the indirect calculation of tree lengths method for phylogenetic tree reconstruction,” in *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2014, pp. 1–4.

[75] P. A. Goloboff, “Methods for faster parsimony analysis,” *Cladistics*, vol. 12, no. 3, pp. 199–220, 1996.

[76] S. Santander-Jiménez, A. Ilic, L. Sousa, and M. A. Vega-Rodríguez, "Accelerating the phylogenetic parsimony function on heterogeneous systems," *Concurrency and Computation: Practice and Experience*, vol. 29, no. 8, p. e4046, 2017.

[77] S. Wienke, P. Springer, C. Terboven, and D. an Mey, "Openacc: First experiences with real-world applications," in *Proceedings of the 18th International Conference on Parallel Processing*, ser. Euro-Par'12. Berlin, Heidelberg: Springer-Verlag, 2012, p. 859–870. [Online]. Available: https://doi.org/10.1007/978-3-642-32820-6_85

[78] S. Santander-Jiménez, M. A. Vega-Rodríguez, A. Zahinos-Márquez, and L. Sousa, "Gpu acceleration of fitch's parsimony on protein data: from kepler to turing," *The Journal of Supercomputing*, pp. 1–27, 2020.

[79] F. Ronquist and J. P. Huelsenbeck, "Mrbayes 3: Bayesian phylogenetic inference under mixed models," *Bioinformatics*, vol. 19, no. 12, pp. 1572–1574, 2003.

[80] A. Stamatakis, "Raxml-vi-hpc: maximum likelihood-based phylogenetic analyses with thousands of taxa and mixed models," *Bioinformatics*, vol. 22, no. 21, pp. 2688–2690, 2006.

[81] P. Malakonakis, A. Brokalakis, N. Alachiotis, E. Sotiriades, and A. Dollas, "Exploring modern fpga platforms for faster phylogeny reconstruction with raxml," in *2020 IEEE 20th International Conference on Bioinformatics and Bioengineering (BIBE)*. IEEE, 2020, pp. 97–104.

[82] F. Pratas, P. Trancoso, A. Stamatakis, and L. Sousa, "Fine-grain parallelism using multi-core, cell/be, and gpu systems: accelerating the phylogenetic likelihood function," in *2009 International Conference on Parallel Processing*. IEEE, 2009, pp. 9–17.

[83] T. Flouri, F. Izquierdo-Carrasco, D. Darriba, A. J. Aberer, L.-T. Nguyen, B. Minh, A. Von Haeseler, and A. Stamatakis, "The phylogenetic likelihood library," *Systematic biology*, vol. 64, no. 2, pp. 356–362, 2015.

[84] D. L. Ayres, A. Darling, D. J. Zwickl, P. Beerli, M. T. Holder, P. O. Lewis, J. P. Huelsenbeck, F. Ronquist, D. L. Swofford, M. P. Cummings *et al.*, "Beagle: an application programming interface and high-performance computing library for statistical phylogenetics," *Systematic biology*, vol. 61, no. 1, pp. 170–173, 2012.

[85] D. L. Ayres, M. P. Cummings, G. Baele, A. E. Darling, P. O. Lewis, D. L. Swofford, J. P. Huelsenbeck, P. Lemey, A. Rambaut, and M. A. Suchard, "Beagle 3: improved performance, scaling, and usability for a high-performance computing library for statistical phylogenetics," *Systematic biology*, vol. 68, no. 6, pp. 1052–1061, 2019.

[86] N. Alachiotis, E. Sotiriades, A. Dollas, and A. Stamatakis, "Exploring fpgas for accelerating the phylogenetic likelihood function," in *2009 IEEE International Symposium on Parallel & Distributed Processing*. IEEE, 2009, pp. 1–8.

[87] S. Zierke and J. D. Bakos, "Fpga acceleration of the phylogenetic likelihood function for bayesian mcmc inference methods," *BMC bioinformatics*, vol. 11, no. 1, pp. 1–12, 2010.

[88] J. Zhou, X. Liu, D. S. Stones, Q. Xie, and G. Wang, "Mrbayes on a graphics processing unit," *Bioinformatics*, vol. 27, no. 9, pp. 1255–1261, 2011.

[89] C. C. Chang, C. C. Chow, L. C. Tellier, S. Vattikuti, S. M. Purcell, and J. J. Lee, "Second-generation plink: rising to the challenge of larger and richer datasets," *Gigascience*, vol. 4, no. 1, pp. s13 742–015, 2015.

[90] S. Purcell, B. Neale, K. Todd-Brown, L. Thomas, M. A. Ferreira, D. Bender, J. Maller, P. Sklar, P. I. De Bakker, M. J. Daly *et al.*, "Plink: a tool set for whole-genome association and population-based linkage analyses," *The American journal of human genetics*, vol. 81, no. 3, pp. 559–575, 2007.

[91] Y. Tang, Z. Li, C. Wang, Y. Liu, H. Yu, A. Wang, and Y. Zhou, "Ldkit: a parallel computing toolkit for linkage disequilibrium analysis," *BMC bioinformatics*, vol. 21, no. 1, pp. 1–8, 2020.

[92] C. Zhang, S.-S. Dong, J.-Y. Xu, W.-M. He, and T.-L. Yang, "Poplddecay: a fast and effective tool for linkage disequilibrium decay analysis based on variant call format files," *Bioinformatics*, vol. 35, no. 10, pp. 1786–1788, 2019.

[93] N. Alachiotis and P. Pavlidis, "Scalable linkage-disequilibrium-based selective sweep detection: a performance guide," *GigaScience*, vol. 5, no. 1, pp. s13 742–016, 2016.

[94] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh, "Basic linear algebra subprograms for fortran usage," *ACM Transactions on Mathematical Software (TOMS)*, vol. 5, no. 3, pp. 308–323, 1979.

[95] S. Hammarling, J. Dongarra, J. Du Croz, and R. Hanson, “An extended set of fortran basic linear algebra subprograms,” *ACM Transactions on Mathematical Software*, vol. 14, no. 1, pp. 1–32, 1988.

[96] J. J. Dongarra, J. Du Croz, S. Hammarling, and I. S. Duff, “A set of level 3 basic linear algebra subprograms,” *ACM Transactions on Mathematical Software (TOMS)*, vol. 16, no. 1, pp. 1–17, 1990.

[97] K. Goto and R. A. v. d. Geijn, “Anatomy of high-performance matrix multiplication,” *ACM Transactions on Mathematical Software (TOMS)*, vol. 34, no. 3, pp. 1–25, 2008.

[98] http://www.openblas.net, 2021.

[99] F. G. Van Zee and R. A. Van De Geijn, “Blis: A framework for rapidly instantiating blas functionality,” *ACM Transactions on Mathematical Software (TOMS)*, vol. 41, no. 3, pp. 1–33, 2015.

[100] N. Alachiotis and G. Weisz, “High performance linkage disequilibrium: Fpgas hold the key,” in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2016, pp. 118–127.

[101] M. Kimura, “The number of heterozygous nucleotide sites maintained in a finite population due to steady flux of mutations,” *Genetics*, vol. 61, no. 4, p. 893, 1969.

[102] 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.

[103] F. L. J. W. Xian-Yu, L. C.-X. B. Hai-Nan, and Z. J.-S. Lai, “Fast computing of linkage disequilibrium on gpu,” in *GPU Technology Conference*. Citeseer, 2013.

[104] C. Theodoris, N. Alachiotis, T. M. Low, and P. Pavlidis, “qld: High-performance computation of linkage disequilibrium on cpu and gpu,” in *2020 IEEE 20th International Conference on Bioinformatics and Bioengineering (BIBE)*. IEEE, 2020, pp. 65–72.

[105] C. Theodoris, T. M. Low, P. Pavlidis, and N. Alachiotis, “quickld: an efficient software for linkage disequilibrium analyses,” *Molecular Ecology Resources*, 2021.

[106] Y. Wang, G. Liu, M. Feng, and L. Wong, "An empirical comparison of several recent epistatic interaction detection methods," *Bioinformatics*, vol. 27, no. 21, pp. 2936–2943, 2011.

[107] K. Van Steen, "Travelling the world of gene–gene interactions," *Briefings in bioinformatics*, vol. 13, no. 1, pp. 1–19, 2012.

[108] X. Wan, C. Yang, Q. Yang, H. Xue, X. Fan, N. L. Tang, and W. Yu, "Boost: A fast approach to detecting gene-gene interactions in genome-wide case-control studies," *The American Journal of Human Genetics*, vol. 87, no. 3, pp. 325–340, 2010.

[109] M. L. Calle, V. Urrea Gales, N. Malats i Riera, K. Van Steen *et al.*, "Mb-mdr: model-based multifactor dimensionality reduction for detecting interactions in high-dimensional genomic data," 2008.

[110] J. Piriyapongsa, C. Ngamphiw, A. Intarapanich, S. Kulawonganunchai, A. Assawamakin, C. Bootchai, P. J. Shaw, and S. Tongsima, "iloci: a snp interaction prioritization technique for detecting epistasis in genome-wide association studies," in *BMC genomics*, vol. 13. Springer, 2012, pp. 1–15.

[111] H. J. Cordell, "Epistasis: what it means, what it doesn't mean, and statistical methods to detect it in humans," *Human molecular genetics*, vol. 11, no. 20, pp. 2463–2468, 2002.

[112] L. Wienbrandt, J. C. Kässens, M. Hübenthal, and D. Ellinghaus, "1000$\times$ faster than plink: Combined fpga and gpu accelerators for logistic regression-based detection of epistasis," *Journal of Computational Science*, vol. 30, pp. 183–193, 2019.

[113] J. González-Domínguez, S. Ramos, J. Touriño, and B. Schmidt, "Parallel pairwise epistasis detection on heterogeneous computing architectures," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 8, pp. 2329–2340, 2015.

[114] L. Wienbrandt, J. C. Kässens, J. González-Domínguez, B. Schmidt, D. Ellinghaus, and M. Schimmler, "Fpga-based acceleration of detecting statistical epistasis in gwas," *Procedia Computer Science*, vol. 29, pp. 220–230, 2014.

[115] G. Pfeiffer, S. Baumgart, J. Schröder, and M. Schimmler, "A massively parallel architecture for bioinformatics," in *International Conference on Computational Science*. Springer, 2009, pp. 994–1003.

[116] J. González-Domínguez, L. Wienbrandt, J. C. Kässens, D. Ellinghaus, M. Schimmler, and B. Schmidt, "Parallelizing epistasis detection in gwas on fpga and gpu-accelerated computing systems," *IEEE/ACM transactions on computational biology and bioinformatics*, vol. 12, no. 5, pp. 982–994, 2015.

[117] G. Hemani, A. Theocharidis, W. Wei, and C. Haley, "Epigpu: exhaustive pairwise epistasis scans parallelized on consumer level graphics cards," *Bioinformatics*, vol. 27, no. 11, pp. 1462–1465, 2011.

[118] L. S. Yung, C. Yang, X. Wan, and W. Yu, "Gboost: a gpu-based tool for detecting gene–gene interactions in genome–wide case control studies," *Bioinformatics*, vol. 27, no. 9, pp. 1309–1310, 2011.

[119] M. Wang, W. Jiang, R. C. W. Ma, and W. Yu, "Gboost 2.0: A gpu-based tool for detecting gene-gene interactions with covariates adjustment in genome-wide association studies," in *2016 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*. IEEE, 2016, pp. 1437–1437.

[120] 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.

[121] Y. Kim and W. Stephan, "Detecting a local signature of genetic hitchhiking along a recombining chromosome," *Genetics*, vol. 160, no. 2, pp. 765–777, 2002.

[122] 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.

[123] B. Charlesworth, M. Morgan, and D. Charlesworth, "The effect of deleterious mutations on neutral molecular variation." *Genetics*, vol. 134, no. 4, pp. 1289–1303, 1993.

[124] 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.

[125] B. F. Voight, S. Kudaravalli, X. Wen, and J. K. Pritchard, "A map of recent positive selection in the human genome," *PLoS biology*, vol. 4, no. 3, p. e72, 2006.

[126] P. C. Sabeti, D. E. Reich, J. M. Higgins, H. Z. Levine, D. J. Richter, S. F. Schaffner, S. B. Gabriel, J. V. Platko, N. J. Patterson, G. J. McDonald *et al.*,

"Detecting recent positive selection in the human genome from haplotype structure," *Nature*, vol. 419, no. 6909, pp. 832–837, 2002.

[127] 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.

[128] 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.

[129] 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.

[130] P. Danecek, A. Auton, G. Abecasis, C. A. Albers, E. Banks, M. A. DePristo, R. E. Handsaker, G. Lunter, G. T. Marth, S. T. Sherry *et al.*, "The variant call format and vcftools," *Bioinformatics*, vol. 27, no. 15, pp. 2156–2158, 2011.

[131] R. R. Hudson, "Generating samples under a wright–fisher neutral model of genetic variation," *Bioinformatics*, vol. 18, no. 2, pp. 337–338, 2002.

[132] G. K. Chen, P. Marjoram, and J. D. Wall, "Fast and flexible simulation of dna sequence data," *Genome research*, vol. 19, no. 1, pp. 136–142, 2009.

[133] M. Kimura, "The number of heterozygous nucleotide sites maintained in a finite population due to steady flux of mutations," *Genetics*, vol. 61, no. 4, p. 893, 1969.

[134] D. V. Zaykin, A. Pudovkin, and B. S. Weir, "Correlation-based inference for linkage disequilibrium with multiple alleles," *Genetics*, vol. 180, no. 1, pp. 533–545, 2008.

[135] J. J. Dongarra, J. Du Croz, S. Hammarling, and I. S. Duff, "A set of level 3 basic linear algebra subprograms," *ACM Transactions on Mathematical Software (TOMS)*, vol. 16, no. 1, pp. 1–17, 1990.

[136] F. G. Van Zee and T. M. Smith, "Implementing high-performance complex matrix multiplication via the 3m and 4m methods," *ACM Transactions on Mathematical Software (TOMS)*, vol. 44, no. 1, pp. 1–36, 2017.

[137] T. M. Low, F. D. Igual, T. M. Smith, and E. S. Quintana-Orti, "Analytical modeling is enough for high-performance blis," *ACM Transactions on Mathematical Software (TOMS)*, vol. 43, no. 2, pp. 1–18, 2016.

[138] S. W. Keckler, W. J. Dally, B. Khailany, M. Garland, and D. Glasco, “Gpus and the future of parallel computing,” *IEEE micro*, vol. 31, no. 5, pp. 7–17, 2011.

[139] T. M. Smith, R. Van De Geijn, M. Smelyanskiy, J. R. Hammond, and F. G. Van Zee, “Anatomy of high-performance many-threaded matrix multiplication,” in *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. IEEE, 2014, pp. 1049–1059.

[140] N. Whitehead and A. Fit-Florea, “Precision & performance: Floating point and ieee 754 compliance for nvidia gpus,” *rn (A+ B)*, vol. 21, no. 1, pp. 18 749–19 424, 2011.